DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

SYSTEM PROGRAMMING

Organización del Computador II

Segundo Cuatrimestre a Distancia 2020

Integrante	LU	Correo electrónico
María Belén Páez	78/19	bpaez2@gmail.com
Federico Hernán Suaiter	37/19	fsuaiter@dc.uba.ar
Tomás Sujovolsky	113/19	tsujovolsky@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. Ejercicio 1

A) En el archivo gdt.c, agregamos los distintos segmentos pedidos. Por fuera de la lista de descriptores de segmento, definimos a cada uno con los valores entre 10 y 13, ya que habían 10 que ya estaban reservados (0 hasta 9, inclusive). En el caso de los segmentos de código, los definimos como execute, readable y no conforming (no vamos a usar puertas de llamada y usarlo nos generaría problemas en el ejercicio 7). En cuanto a los de datos, los definimos como readable y writable, ya que queremos poder escribir en ellos. A estos cuatro segmentos, les definimos la granularidad en 1, ya que como su tamaño es de 201 MiB, excede a un MiB y por ende, debemos definir el límite en páginas de 4 KiB.

B) En el archivo kernel.asm, desactivamos las interrupciones (ya que no tenemos definido cómo atenderlas todavía), activamos A20, cargamos la gdt, usando el descriptor de la gdt (tomado como externo desde el archivo gdt.c, donde está definido) y activamos el bit PE del registro CR0, que indica si estamos en modo protegido. Luego hacemos el salto lejano a modo protegido, accediendo con la base en el segmento de código nivel 0, creado en el ejercicio a. (usando una etiqueta con el valor del selector del segmento pedido) Dicho selector es el que tiene indice 10, el indicador de tabla 0 (está en la gdt) y el RPL en 0.

Luego definimos que trabajamos en 32 bits y ubicamos la etiqueta modoProtegido, a la cual saltamos. Desde allí, establecemos los registros de segmento que son de datos y pila (ds, es, fs, gs y ss) con el valor del selector de segmento de datos de nivel 0, definido en el ítem a (al cual le corresponde el índice 12). Establecemos la base y el tope de la pila en 0x25000, que es la dirección dada.

C) En el archivo gdt.c, agregamos un descriptor de segmento (con valor 14) que corresponde al área de pantalla en memoria y que por ello, debe ser de datos(con posibilidad de escritura). Como ocupa menos de 1 MiB (exactamente 8000 bytes, ya que son 80 x 50 x 2 bytes de pantalla), podemos usar granularidad en 0 y medir el límite en bytes.

D) En el archivo kernel.asm, antes de saltar a modo protegido, definimos a 'fs' como el selector de segmento de video. Una vez hecho esto, pintamos a la pantalla como vemos en la sección dicha. Es decir, seteamos en 0 al caracter del primer byte de cada pixel (lo limpiamos para que no figuren datos anteriores impresos en pantalla), y definimos el color en el segundo byte. A su vez, seteamos la parte baja de la pantalla, donde se ve el puntaje, junto con los caracteres que no se modificaran durante el transcurso del juego.

2. Ejercicio 2

A) Las entradas de la IDT tienen en común el selector de segmento, el cual es el de código de nivel 0, y los bits p, dpl y d que son 1, 0 y 1, respectivamente. Esto es así, ya que el segmento está presente, el nivel de privilegio en el que estamos es 0 y estamos en modo protegido, por lo que estamos trabajando con 32 bits. De esta manera, en el archivo idt.c, definimos la macro idt_entry con los datos dichos. Luego, los inicializamos en la tabla idt, tomando como entrada a los enteros entre 0 y 19, ya que las excepciones del procesador son 20.

En el archivo isr.asm, agregamos a la macro de _isr## la llamada a la función (definida en idt.c) que imprime el número de tal excepción en la esquina izquierda de la pantalla. Luego, definimos las excepciones entre 0 y 19 usando esta macro, las tomamos como globales y las declaramos en el archivo isr.h, como funciones void.(Esto lo modificamos en luego en el ejercicio 8).

B) Para que el procesador utilice la IDT creada anteriormente, modificamos el archivo kernel.asm. Allí llamamos a la función que inicializa la idt (que es la recién creada) y la cargamos en el registro idtr, usando el descriptor (definido también en idt.c). Luego, hacemos posibles las interrupciones externas remapeando PIC (para que el selector apunte al índice correcto) y habilitándolo, junto con las interrupciones.

Para probarlo, realizamos una división por cero. Al ocurrir esto nos muestra la interrupción (excepción) con número 00, que es el valor correcto para este error.(Esto luego lo quitamos de la entrega final ya que no es pertinente).

3. Ejercicio 3

A) Utilizando la macro IDT_ENTRY, inicializamos las ints. de reloj y teclado, con los valores 32 y 33, respectivamente; y las syscalls de 88, 89, 100 y 123. Agregamos estas llamadas también a isr.h con el formato void _isr##(), donde ## indica el numero correspondiente. También agregamos en isr.asm las etiquetas _isr## y las declaramos como globales.

(Posteriormente, utilizamos la macro IDT_ENTRY_syscall para las syscalls 88, 89, 100 y 123. A su vez, a las exepciones que pushean un código de error les asignmos una macro distinta (ISR_EC)).

B) Escribimos la rutina de atención en el mismo archivo donde se define next_clock, basándonos en la función vista en las clases prácticas.

C) Escribimos la rutina de atención de interrupción del teclado en el archivo isr.asm. Desde allí llamamos a la función declarada como externa (printScanCode) la cual detecta si la tecla que se presionó fue un número del 0 al 9 del teclado alfanumérico. De ser así, devuelve el número exacto que se presionó en la parte superior derecha de la pantalla.

Esta función printScanCode está ubicada en el archivo screen.c, y se encuentra declarada en el archivo screen.h.

D) Para lograr esto, colocamos las declaraciones de las interrupciones _isr88, _isr89, _isr100 y _isr123 en el archivo isr.h.

En el archivo idt.c declaramos bajo la definición de IDT_ENTRY (como se mencionó en un ejercicio pasado) las siguientes entradas: IDT_ENTRY(88), IDT_ENTRY(89), IDT_ENTRY(100) e IDT_ENTRY(123).

En el archivo isr.asm, escribimos lo que deben realizar cada una de estas interrupciones (Estas de momento modifican eax según corresponda). No hacemos pushad y popad, ya que esto mantendría el valor de eax.

(Posteriormente, utilizamos la macro IDT_ENTRY_syscall para las syscalls 88, 89, 100 y 123).

4. Ejercicio 4

A) En el archivo mmu.c definimos la función mmu_init_kernel_dir que se encarga de agarrar las direcciones 0x25000 y 0x26000 e inicializar allí el "Directorio de paginas de Kernel" y la "Tabla de paginas Kernel" respectivamente.

Casteamos tales direcciones como un puntero a un page_directory_entry y un page_table_entry respectivamente y los rellenamos con ceros.

Guardamos en la primer entrada de la PD la dirección de la PT creada, y en la PT guardamos los primeros 4MB de páginas de forma tal de cumplir Identity Mapping.

B) Para activar paginación llamamos desde el kernel.asm a la función mmu_init la cual asigna la primer página libre. Luego llamamos a la función mmu_init_kernel_dir que hace lo descrito anteriormente.

Cargamos el registro CR3 con la dirección que nos devuelve mmu_init_kernel_dir. Por último, activamos paginación seteando el bit PD de CR0 en 1.

C) Para lograr imprimir nuestras libretas por pantalla utilizamos la función print_text_pm brindada por la cátedra, junto con el mensaje y el largo, que fueron creados por nosotros en el archivo kernel.asm.

5. Ejercicio 5

A) La función mmu_init definida en el archivo mmu.c se encarga de tomar la primera página libre del área libre del kernel (por primera y única vez).

B) **I- mmu_map_page(uint32_t cr3, uint32_t virtual, uint32_t phy, uint32_t attrs)**

La función mmu_map_page definida en el archivo mmu.c toma como parámetros al CR3, una dirección virtual y una física, y los atributos. Gracias a la CR3 y la dir_virtual accede a la entrada de directorio

correspondiente y si el bit de presente esta en 0, crea una nueva tabla de página con los atributos pasados por parámetro. El rango de atributos va del 0 al 7, ya que nos basamos en los bits US, RW y P. De esta manera, un nivel 0 podría ser pasando los atributos 0 o 1 y un nivel 3, pasando 6 o 7. Luego de acceder a la entrada de la tabla de página correspondiente (es decir, de la recién creada o de la que ya existía) guardamos allí la dirección física pasada por parámetro y sus atributos.

II- mmu_unmap_page(uint32_t cr3, uint32_t virtual)

La función mmu_unmap_page definida en el archivo mmu.c toma como parámetros al CR3 y una dirección virtual. Gracias a la CR3 y la dir_virtual accede a la entrada de directorio correspondiente y si el bit de presente esta en 1, accede a la entrada de la tabla de página correspondiente, pone 0 en la dirección física y limpia el bit de presente.

C) La función mmu_init_task_dir definida en el archivo mmu.c se encarga de inicializar el esquema de paginación de una tarea, cualquiera sea. Toma como parámetros el CR3, la dirección de donde arranca el código en el kernel, la dirección virtual y la física donde queremos mapearlo, y la cantidad de páginas que ocupa la tarea. Comienza pidiendo una página libre para el directorio de la tabla de paginas y la inicializa en 0. Luego, mapea con Identity Mapping los primeros 4MB para poder tener acceso al Kernel, donde se encuentra el código de la tarea. Luego, mapeamos en el mapa del kernel la dirección virtual a la física, pasadas por parámetro, si exceden el rango mapeado. Una vez que tengo acceso a la dirección física donde quiero guardar el código de la tarea, lo copio allí. Guardamos las páginas en las tabla de páginas creada (habiendo inicializado el directorio correctamente) y desmapeamos las páginas del mapa del kernel. (Los cuatro primeros MB no los desmapeamos).

D)

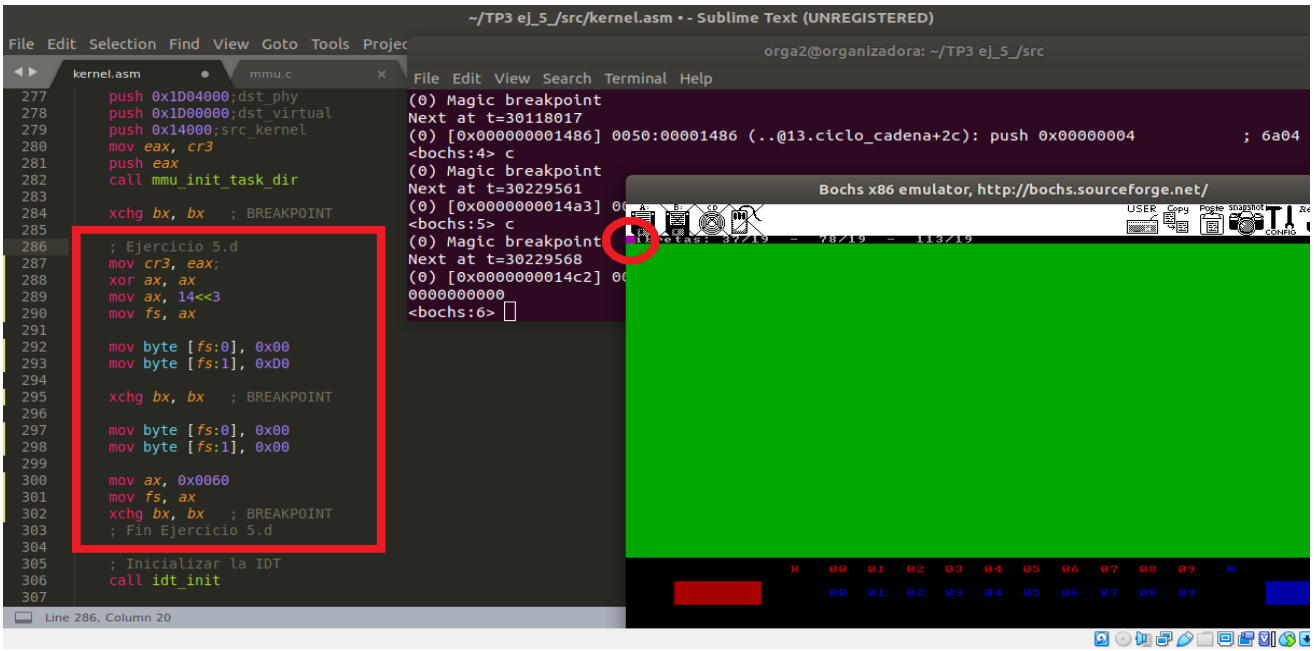

Figura 1: Cambio del color del primer caracter tras construir un mapa de memoria.

6. Ejercicio 6

A,C y D) Comenzamos el ejercicio 6 definiendo en gdt.c las entradas "necesarias" en la GDT: la Inicial, el Idle, Rick, Morty y los 20 Mr.Meeseeks. La base de todas las tareas las seteamos en 0, ya que las definiremos en tiempo de ejecución, usando la función tss_init. Los límites los seteamos en $67h$, ya que no tiene ningún mapa de bits adicional, y todos serán de sistema (S = 0). El bit Busy de todas las entradas será 0, ya que ninguno arrancará en ejecución y el nivel será 0 para el inicial y Idle, y 3 para los jugadores y Mr. Meeseeks.

B) Inicializamos la tss de la Idle en el tss.c, usando la pila del kernel y la dirección del código como eip. Dado que tanto la pila como el código de la tarea Idle pertenecen al área del Kernel y a su área libre, y que estas están mapeadas con identity mapping en los esquemas de paginación del kernel y todas las tareas, no hace falta volver a mapearlos. Seteamos los flags como 0x202, ya que seteamos el flag de interrupciones, para que estén habilitadas dentro de la tarea. Ponemos como registros de código y de datos de nivel 0 a los definidos en la gdt, utilizando el selector que los apunta, con RPL = 00b. El resto lo seteamos en 0, ya que, por ejemplo, no tiene pila adicional de nivel 0 a la que saltar en caso de ser interrumpida por una de mayor nivel (por ser Idle de nivel 0).

E) En el kernel.asm, movemos a ax el selector de la entrada de la gdt de la tarea inicial y lo cargamos en TR usando la instrucción LTR, definida en i386.c. Luego hacemos un jump far usando como selector al que apunta a la entrada de la tarea Idle en la gdt y offset 0.

F) Para el fácil manejo de la TSS, decidimos tener una lista que contiene las direcciones correspondientes a dónde se hayan declaradas las tss de cada tarea, según el orden de la GDT. La posición en la lista de un elemento define su task_id.

Para inicializar la TSS creamos la función tss_init que toma un task ID, el CR3, el eip (donde arranca el código) y el lugar donde arranca el stack. Luego, la función pide una pagina libre para la pila de nivel 0 y completa la TSS con los datos pasados por parámetro. Además, completa el SS0 con un selector de código de nivel 0 y al resto de los registros de segmento con los selectores de segmento de nivel 3. Por último, utilizamos la función gdt_tss_init para inicializar la base en el descriptor correspondiente de la GDT.

```
void gdt_tss_init(uint32_t task_id){
    gdt[task_id+15].base_15_0 = (uint16_t) ((uint32_t)(tss_list[task_id]));
    gdt[task_id+15].base_23_16 = (uint8_t) (((uint32_t)(tss_list[task_id])) >> 16);
    gdt[task_id+15].base_31_24 = (uint8_t) (((uint32_t)(tss_list[task_id])) >> 24);
}
```

Figura 2: Inicializador de las bases de la gdt. Toma de una lista, castea, shiftea y luego vuelve a castear.

7. Ejercicio 7

Para llevar a cabo las tares del scheduler, definimos las siguientes estructuras:

-rick_tasks y morty_tasks: Estos arrays contienen los índices de la GDT de las TSS de las tareas de Rick, Morty y sus respectivos Ms. Meeseeks. Ambos arrays son de 11 valores uint16_t, donde la primera posición es la tarea principal y las otras 10 son sus respectivos Meeseeks. Ya están definidos en el código para evitar tener que definirlas en tiempo de compilación o de ejecución. Tenemos una lista de Meeseeks donde definimos cuales son las tareas activas (arrancan todas en 0). Las entradas en la GDT de la TSS de las tareas Meeseeks están inicializadas (con los campos de base en 0). Estos se modifican al llamar a la funcion ïnicializar_meeseeks", donde también se definen las TSS de cada una (con los esp, ebp y eip en 0, ya que se definen al activar la tarea, al usar la syscall Meeseeks).

-indicador_jugador: este uint32_t puede tomar los valores cero o uno. Indica el próximo jugador del cual el sched_next_task deberá dar la tarea. Si es cero, la próxima tarea a ejecutar es una de rick, y si es uno sera de morty. Otro uso que se le puede dar es para ver de qué jugador es la tarea que se esta ejecutando ahora. Por ejemplo, si es cero, como la próxima tarea que debo ejecutar es de rick, la tarea actual o es el idle o es de morty.

-indicador_rick_task y indicador_morty_task: Estos uint32_t indican la posición de la próxima tarea a leer en las listas de rick_tasks y morty_tasks, respectivamente.

-rick_meeseeks y morty_meeseeks: Estas listas de 10 entradas contienen solo unos y ceros, y las usaremos como indicadores de los meeseeks que están presentes en el juego. Es la lista nombrada en el primer ítem. Por ejemplo, si rick_meeseeks[3] = 0, esto implica que el meeseeks 3 de rick no esta en juego. Y, otro ejemplo, morty_meeseeks[9] = 1 implica que el meeseeks 9 de morty esta en juego.

El la funcion sched_init se encargara de setear en cero el indicador_jugador, indicador_rick_task, indicador_morty_task y todas las posiciones de las listas de meeseeks de cada jugador. De esta manera, el primer jugador que tomara el scheduler es rick, y la primera task a leer del sched_next_task sera la tarea rick. Y cuando se tenga que leer una tarea de morty, la primera que se leerá sera la tarea morty.

b) Explicaremos la funcion sched_next_task con el siguiente pseudo código:

```
#def sched_next_task():
 if (indicador_jugador == rick):
     indicador_jugador = morty
     proxima_tarea = rick_tasks[indicador_rick_task]
     while (proxima_tarea != Tarea_rick and meeseeks_muerto(indicador_rick_task)):
         indicador_rick_task++
         if(indicador_rick_task == len(rick_tasks):
         indicador_rick_task = 0
     proxima_tarea = rick_tasks[indicador_rick_task]
     next_task = rick_tasks[indicador_rick_task]
     indicador_rick_task++
     if(indicador_rick_task == len(rick_tasks):
         indicador_rick_task = 0
 else:
     caso análogo para morty
return next_task
```

c y d) Lo chequeamos y luego lo pisamos con el resto de las implementaciones del ejercicio 8.

e) La funcion de Desalojar la implementamos de manera tal que apaga al meeseeks a desalojar de su lista rick_meeseeks o morty_meeseeks y lo quita del mapa. Si una tarea principal es desalojada, finaliza el juego y se llama a la funcion end_game() que apaga las interrupciones y solo realiza un jmp $.

f) El MODO DEBUG requiere de dos variables globales llamadas ïndicador_debuggerz ïndicador_debugger_on_screen". La primera declara si se presionó la tecla Y y se activó el modo debug (lo cual se puede chequear en la esquina derecha, donde se imprime si se está en modo debug). La segunda variable declara si se está viendo la pantalla MODO DEBUG, lo que hace que las interrupciones de reloj no hagan conmutaciones de tareas mientras que esté prendida. Si se estaba en MODO DEBUG y se vuelve

a presionar dicha tecla, este de desactiva, al igual que la otra variable, y deja de aparecer el mensaje en pantalla. En la siguiente interrupcion de reloj, se retoma el juego.

En nuestra implementación no requerimos guardarnos la pantalla actual en memoria, ya que gracias a la función que setea en una lista (map, que está en sched.c) si en cierta posición (x,y) del mapa hay un meeseek de Rick, uno de Morty o una semilla, estos valores se preservan durante el MODO DEBUG, permitiendo ser retomados al refrescar la pantalla en el siguiente clock.

La rutina de atención de las interrupciones de teclado chequea, mediante un llamado a la función ”printScanCode”(en screen.c), si, en caso de que se haya presionado la tecla Y, el ïndicador_debugger.es 0. En ese caso, lo setea en 1 y el juego continua hasta que se genere una excepción. Cuando esta llegue, la rutina de atención de dicha excepción llamará a ïmprimoModoDebug.o ïmprimoModoDebug_ec”, según si tiene errorcode o no (en screen.c). Dicha función chequea si esta activada ïndicador_debuggerz en caso de que no lo este, continua con el desalojo de la tarea actual y la conmutación a la siguiente. Si lo está, entonces setea en uno ïndicador_debugger_on_screen.eimprime en pantalla la famosa pantalla del MODO DEBUG. Para esto, pasamos como parametros de la función a los registros de segmento, al numero de excepción y al struct ”param_pusheados_ec.o ”param_pusheados”(según si tiene errorcode). Estos últimos están definidos en screen.h y se encargan de darle nombre a aquello que se pushea en memoria al cambiar de nivel de privilegio al atender la interrupción y al hacer pushad. En el que termina con ”_eç se le agrega el lugar donde se guarda el errorcode. De esta manera, la función que imprime la pantalla tiene a mano todos los registros que necesita. Para lo del stack, tomamos los 3 últimos valores que están en la pila de nivel 3 (a la cual podemos acceder, ya que no cambiamos de contexto y su dirección esta en el struct). Para lo del backtrace, aprovechamos que tenemos, por el pushad, el valor de la ebp (la de la pila que llamó a la excepción) y chequeamos si la dirección de retorno que se pusheó antes de la dirección que apunta nuestra ebp es válida. Para ello se fija que esté en rango (dentro de las direcciones virtuales donde hay pila y codigo de cualquier Mr. Meeseek) y que su modulo 4 sea 0. En caso de que no se cumpla, imprime ceros. Esta accion la repite 4 veces ingresando en los ebp apuntados. Para el numero de tarea, impreso en la esquina derecha superior, se chequea cuál es el TR (la tarea actual) y se lo compara con el selector de la TSS que está en la GDT, de cada tarea. De esta manera, imprime el mismo índice que figura en la lista de TSS (que está en tss.c), según corresponda. Lo de la impresión de qué excepcion es, lo hacemos usando dos matrices con las que se forman el mensaje correctamente.

8. Ejercicio 8

G) La pantalla del juego la inicializamos luego de pasar a modo protegido en el kernel. Allí inicializamos el campo de juego, los recuadros de los puntajes y las diferentes ubicaciones en las cuales aparecerán el estado de los meeseeks. Para esto utilizamos funciones propias que se pueden encontrar en el kernel.asm .

Luego inicializamos las semillas. Para ello llamamos a la función set_Mega_Seeds. Esta función se encuentra en screen.c y ubica las 40 semillas en el mapa y se encarga de guardarlas en una matriz denominada semillas. Luego aparecerán las semillas en pantalla cuando ocurra la primera interrupción del reloj y se efectúe un map_update.

H) Para manejar los meeseeks creamos un struct que guarde todo la información que creíamos que podíamos requerir. Este struct se encuentra el archivo game.h. Los elementos del struct se dividen en 3: Variables de reseteo, para cuando se se necesite reutilizar el meeseeks; propiedades del meeseeks que explican cualidades de estado actual (el resto son el contexto de ejecución de la tarea); e identificadores del meeseeks. Estos son varios y podrían a llegar a ser redundantes entre si, pero sirvieron para simplificar el código. Armamos una lista de 20 structs meeseeks_t y nos referiremos a esta lista para rápido acceso de la info de los meeseeks.

En contexto de compilación, inicializamos ciertos aspectos hardcodeados de los meeseeks: su CR3, su task register, su task id, de qué jugador son, cuál número de meeseeks son, su posición de memoria virtual donde se ejecuta y su puntero en memoria virtual a su pila.

Para crear un nuevo meeseeks primero buscamos cual es el primer meeseeks disponible del jugador actual mediante la lista del sched.c rick_meeseeks o morty_meeseeks (que contiene un 0 sin el iésimo meeseeks esta libre o un 1 si esta activo). Si encontramos uno libre, lo activamos. Al activarlo, entramos en su posición en la lista de meeseeks_t y les seteamos sus propiedades con el x e y dados, movimiento en 7, turnos vivos en cero, portal habilitado y le guardamos su puntero al código que deberá ejecutar (pasado por parámetro). Luego, en ”mapeo_y_guardo_código”mapeamos sus dos paginas virtuales a las paginas indicadas por su x e y (la dirección física resultante es 0x400000 + 0x2000*x + 0x2000*y*0x50). Una vez mapeado, nos encargamos de copiar el códigos desde la dirección en el kernel (guardada en el struct de meeseeks_t) hasta la virtual recién mapeada.

Una vez que que mapeamos y copiamos el código, inicializamos su TSS. Para esto accedemos a la lista con las entradas de la tss, mediante el elemento del struct del meeseeks task_id, y le reseteamos los registros de propósito general, segmentos de código (porque si antes estuvo vivo lo desalojamos en nivel cero y hay que devolverlo a nivel 3), sus punteros a sus pilas (de nivel cero y nivel 3), su EIP y sus eflags.

I) Al llamar a la syscall move, lo primero que hacemos es identificar quien fue el que realizo el llamado. Si fue la tarea Rick o Morty, realizamos una división por cero con tal de desalojarla. Si fue un meeseeks el que hizo el llamado, primero checkeamos que efectivamente su movimiento disponible sea suficiente para realizar el move, y si no lo es la función no hace nada mas. Si puede moverse, les sumamos los offsets pasados por parámetros y les aplicamos las cuentas adecuadas para que se aproveche el mapa redondo. Si cayo en una semilla, sumamos los puntos a la tarea jugador actual, sacamos al meeseeks del mapa (limpiando dicho x,y de la lista map) y lo desalojamos. Si no cayo en una semilla, actualizamos su posición en el mapa, en el struct y realizamos la función movimiento. La función movimiento se encarga de mapear con Identity Mapping las dos paginas físicas donde se encontraba (según los x y del struct) y mapea su dirección virtual correspondiente a los nuevos x, y donde se quiere posicionar. Luego, copia aquello que tenia en sus paginas anteriores a las actuales (pisando lo que esté escrito allí).

Para aplicar la restricción de movimiento faltante (bajar el movimiento a medida que avanzan los turnos), introdujimos el siguiente mecanismo dentro del scheduler. Una vez que llamamos sched_next_task, pero antes de realizar el cambio de tarea, llamamos a la función .ªumentar_turnos_vivosço nel next_task. Esta función, si la siguiente tarea sera un meeseeks, le aumenta en uno su elemento turnos_vivos del meeseeks_t y si este es un numero impar mayor a 1 y su movimiento es mayor a 1, le baja en uno su movimiento.

J) La syscall look tiene el mismo mecanismo de arranque la syscall move, primero identifica quien la llamo, y si fue una tarea principal realiza una división por cero. Si no, se guarda la posición del meeseeks que la llamo y la compara con todas las semillas del mapa que están activas (guardadas en una lista con su estado en 1), quedándose con la que menor delta x y delta y tenga. Para devolver en EAX y EBX los resultados, estos deltas los guardamos en una lista return_look”de dos posiciones y los cargamos desde assembler en los registros con la instrucción mov.

K) Al llamar a la syscall portal gun, lo primero que hacemos es identificar quien realizo el llamado y fijarnos si tiene el portal disponible. Si lo tiene, se lo deshabilitamos, y pasamos a un contrario aleatorio, si es que hay. Si hay contrarios, se consigue uno al azar y se lo teletransporta. La función escrita en C ”teletransportar.ES un copypaste de la función de C ”move”sin restricciones de movimiento y con x e y elegidos al azar.

M) Cuando el mapa se queda sin semillas, es decir, los meeseeks obtuvieron todas las semillas que había, termina el juego. Nos aseguramos de que esto ocurre cuando en la interrupción del reloj llamamos a la función çall analizar_ganador”. Esta función se encuentra implementada en game.c y revisa si la cantidad de semillas que quedan actualmente en pantalla. Cuando esta cantidad llega a 0, revisamos el puntaje de los jugadores para ver quien gano o si hubo un empate. Luego de esto, ejecutamos la función end_game la cual se encuentra en isr.asm. Esta se encarga de apagar las interrupciones y ciclar

infinitamente, evitando así la continuidad del juego.

9. Ejercicio 9

Para probar todo realizamos varias tareas rick y morty con las cuales encontramos varios bugs y errores de nuestro código. La ultima que probamos es una que realiza todas las syscalls. Realizamos un merge request con estos dos códigos al repo de tests de la cátedra. Los dos toman la misma estrategia, cada meeseeks busca su semilla mas cercana e intenta acercarse de uno en uno. Si no se llevan la semilla intentan teletransportar un contrario y vuelven a empezar. Lo que diferencia a estos dos jugadores, es que morty pone sus meeseeks en las esquinas del mapa y rick lo hace en el medio. Como look no busca semillas teniendo en cuenta el mapa redondo, rick tendrá la ventaja ya que tiene un rango de visión de las semillas de 360 grados , mientras que morty lo tiene de 45 grados.

EL juego finaliza cuando se desaloja una tarea principal o cuando se terminan las semillas. Si terminan porque no hay semillas, el jugador con mayor cantidad de puntos gana y en caso de que tengan la misma cantidad de puntos, esta programado un empate.

10. Reentrega

Para los tests look-n-move, portal-gun, pisar y look-rick-morty el error a arreglar fue el mismo. Nuestra implementación de la syscall look tomaba ser llamada por una tarea principal como una acción ilegal, por lo cal inmediatamente desalojaba la tarea rick que la llamaba y le cedía la victoria a morty. Esto fue un error de interpretación de consigna de nuestra parte y fue simplemente cambiar el sistema de división por cero si la tarea era una principal a un retorno de -1 en eax y ebx. Es por esto que el tests move-rick funcionaba bien, ya que no utilizaba la syscall look en una tarea principal.

Para el test meeseeks-pos-valida y meeseeks-invalido, no habíamos implementado ningún mecanismo para determinar si la posición y el código eran correctos o no, habíamos tomado por supuesto que estos serian validos. Para arreglar esto, agregamos en la función meeseeks de game.c una guarda el la que si esta se cumple, algún parámetro seria invalido. Esta guarda se fija si el x e y son menores a 0 y mayores a 79 y 39, respectivamente, y esta guarda también se fija que el código pertenezca al rango de las tareas principales, es decir, entre 0x1D00000 y 0x1D00FFF.

En la TSS hicimos un pequeño arreglo, el cual consistió en modificar los nombres de nuestra lista tss_-list. Aquí había algunos meeseeks cuyo nombre daba a entender que estaban bajo el dominio de Morty, cuando en realidad debían estar bajo el dominio de Rick.

Para el test backtrace, arreglamos la forma en la que accedemos a la pila para rastrear el "$VIEJO\ EBP$" y el EIP (la dirección de retorno de la tarea que llamó). Casteamos al ebp actual como un vector y si el ebp esta dentro del rango de la pila de la tarea actual (y no en el "borde"), imprime el segundo entero de 32 bits del vector, o sea el EIP y renombramos al primer entero como ebp1_p. De esta manera, comparamos de igual manera si el ebp1_p esta dentro del rango, hasta llegar a las cuatro direcciones de retorno que se deben imprimir. Si no está en rango de la pila, se imprime 0. Al realizar dichas comparaciones nos aseguramos de que el backtrace no provoque un page fault al querer acceder a una posición errónea. Para ver que esté en rango, si la tarea actual es la Inicial, Idle, Morty o Rick, toma como el ebp viejo al que está guardado en el TSS; y si es un Meeseeks, toma el que está en su respectivo struct meeseeks_t.

A la hora de implementarlo con la optimización O0, necesitamos que las tareas se corran con dicha optimización, pero que el resto de las tareas se corran con Og, debido a que, como vimos con Marco (el 22/12), al refrescar la pantalla en cada interrupcion de reloj, la cantidad de instrucciones por segundo

que hace nuestro codigo no es la suficiente y por ende, el codigo no se ejecuta de manera correcta. (la forma que vimos con Marco fue duplicando el ips del bochs)

También le hicimos un pequeño cambio a la forma de imprimir la pila en el debugger. El mismo consiste de en analizar que la base de la pila sea distinta del tope de la pila, es decir, que tengamos material en la pila para poder imprimir. Si por algún casual no hay nada, procedemos de forma parecida al backtrace e imprimimos ceros por pantalla.

A la hora de implementarlo con la optimización O0, necesitamos que las tareas se corran con dicha optimización, pero que el resto de las tareas se corran con Og, debido a que, como vimos con Marco (el 22/12), al refrescar la pantalla en cada interrupcion de reloj, la cantidad de instrucciones por segundo que hace nuestro codigo no es la suficiente y por ende, el codigo no se ejecuta de manera correcta. (la forma que vimos con Marco fue duplicando el ips del bochs)

Detalle:
En el caso del test pisar, notamos que, si bien el juego termina y gana Rick (haciendo lo esperado por el test), en pantalla se ve impreso a el meeseek de Morty sobre el de Rick. Esto debería ser al revés y se arreglaría agregando que se actualice en pantalla la posición del meeseek que se acaba de mover.