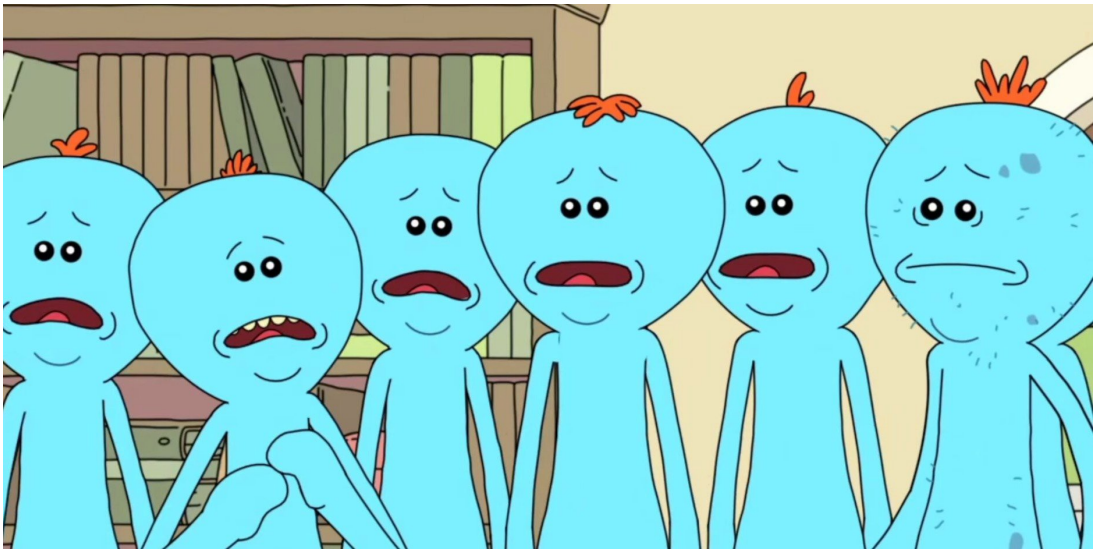
Trabajo Práctico 3

System Programming

Segundo Cuatrimestre 2020

Low Power



0.1 Objetivo

Este trabajo práctico consiste en un conjunto de ejercicios en los que se aplican de forma gradual los conceptos de *System Programming* vistos en las clases teóricas y prácticas. Los ejercicios están inspirados en la serie *Rick y Morty*, siendo el sistema que desarrollarán un multiverso en "memoria".

En este universo viven un par de personajes o jugadores denominados *Rick* y *Morty*. Cada jugador tiene en su poder una caja de Mr Meeseeks que utilizará para capturar `Mega Semillas` sobre un mapa. Los Mr Meeseeks, son curiosos personajes mágicos que aparecen al presionar el único botón que una caja de Mr Meeseeks tiene. La vida de un Mr Meeseeks esta limitada a cumplir una orden clara y concisa. Una vez que cumplen su cometido en la vida, es decir cumplir su orden, desaparecen así como aparecieron.

Como no queremos perder control del juego y destruir el universo, la cantidad de Mr Meeseeks que pueden vivir simultáneamente en el juego está limitada. Los jugadores utilizarán la caja y les ordenarán a sus Mr Meeseeks, buscar y recolectar `Mega Semillas`.

Es importante recordar, que si bien este trabajo práctico esta ilustrado por un juego, los ejercicios proponen utilizar los mecanismos que posee el procesador para la programación desde el punto de vista del sistema operativo.

0.2 Introducción

Para este trabajo se utilizará como entorno de pruebas el programa *Bochs*. El mismo permite simular una computadora IBM-PC compatible desde el inicio, y realizar tareas de debugging. Todo el código provisto para la realización del presente trabajo está ideado para correr en *Bochs* de forma sencilla.

Una computadora al iniciar comienza con la ejecución del POST y el BIOS, el cual se encarga de reconocer el primer dispositivo de booteo. En este caso dispondremos de un *Floppy Disk* como dispositivo de booteo. En el primer sector de dicho *floppy*, se almacena el *boot-sector*. El BIOS se encarga de copiar a memoria 512 bytes del sector, a partir de la dirección `0x7C00`. Luego, se comienza a ejecutar el código a partir esta dirección. El boot-sector debe encontrar en el *floppy* el archivo `KERNEL.BIN` y copiarlo a memoria. Éste se copia a partir de la dirección `0x1200`, y luego se ejecuta a partir de esa misma dirección. En la figura 2 se presenta el mapa de organización de la memoria utilizada por el *kernel*.

Es importante tener en cuenta que el código del *boot-sector* se encarga exclusivamente de copiar el *kernel* y dar el control al mismo, es decir, no cambia el modo del procesador. El código del *boot-sector*, como así todo el esquema de trabajo para armar el *kernel* y correr tareas, es provisto por la cátedra.

Los archivos a utilizar como punto de partida para este trabajo práctico son los siguientes:

- `Makefile` - encargado de compilar y generar el *floppy disk*.
- `bochsrc` y `bochsdbg` - configuración para inicializar Bochs.
- `diskette.img` - la imagen del *floppy* que contiene el *boot-sector* preparado para cargar el *kernel*.
- `kernel.asm` - esquema básico del código para el *kernel*.
- `defines.h` y `colors.h` - constantes y definiciones.
- `gdt.h` y `gdt.c` - definición de la tabla de descriptores globales.
- `tss.h` y `tss.c` - definición de entradas de TSS.
- `idt.h` y `idt.c` - entradas para la IDT y funciones asociadas como `idt_init` para completar entradas en la IDT.
- `isr.h` y `isr.asm` - definiciones de las rutinas para atender interrupciones (*Interrupt Service Routines*).
- `sched.h` y `sched.c` - rutinas asociadas al *scheduler*.
- `mmu.h` y `mmu.c` - rutinas asociadas a la administración de memoria.
- `screen.h` y `screen.c` - rutinas para pintar la pantalla.
- `a20.asm` - rutinas para habilitar y deshabilitar A20.
- `print.mac` - macros útiles para imprimir por pantalla y transformar valores.
- `idle.asm` - código de la tarea `Idle`.
- `game.h` y `game.c` - implementación de los llamados al sistema y lógica del juego.
- `syscalls.h` - interfaz a utilizar desde las tareas para los llamados al sistema.
- `kassert.h` - rutinas para garantizar invariantes en el kernel.
- `prng.c`, `prng.h` y `seed.c` - generacion de numeros aleatorios.
- `taskRick.c` - código de las tareas del jugador Rick.
- `taskMorty.c` - código de las tareas del jugador Morty.
- `i386.h` - funciones auxiliares para utilizar *assembly* desde C.
- `pic.c` y `pic.h` - funciones `pic_enable`, `pic_disable`, `pic_finish1` y `pic_reset`.

Todos los archivos provistos por la cátedra **pueden** y **deben** ser modificados. Los mismos sirven como guía del trabajo y están armados de esa forma, es decir, que antes de utilizar cualquier parte del código **deben** entenderla y modificarla para que cumpla con las especificaciones de su propia implementación. Tengan en cuenta que el código provisto por la cátedra puede **no** funcionar en todos los casos, y depende del desarrollo de cada uno de los trabajos prácticos.

A continuación se da paso al enunciado, se recomienda leerlo en su totalidad antes de comenzar con los ejercicios. El núcleo de los ejercicios será explicado en las clases, dejando cuestiones cosméticas y de informe para que realicen por su cuenta.

0.3 Historia

Todo comenzó con una pelea entre Rick y Morty, ya que Morty debía llegar rápidamente a la Tierra para participar de una cita con Jessica. Nuestros personajes se encontraban en medio de una aventura, otra vez salvando a la galaxia, cuando la batería del auto se averío nuevamente. Hacia un tiempo, Rick había remplazado a su universo dentro de su batería[1] por un combustible más convencional, las `Mega Semillas`. Estas se utilizan para producir energía a través de un mecanismo similar al de un RTG[2].

Lamentablemente, las `Mega Semillas` no son tan simples de encontrar, así que Rick desafío a Morty y lo invito a buscar las `Mega Semillas`. El que más `Mega Semillas` encontrara, podría elegir cual sería la próxima aventura (en otras palabras, a donde ir con el auto reparado). Si bien Rick es muy inteligente, no es muy trabajador, así que tanto *Rick* como *Morty*, dispondrán de una caja de Mr Meeseeks para hacer el trabajo duro.

Los Mr Meeseeks son otras tareas en el juego, creadas por los jugadores y que viven para cumplir una orden, en este caso, buscar `Mega Semillas`.

0.4 Juego

El juego en sí consiste en dos jugadores `Rick` y `Morty` que enviarán cada uno como máximo 10 Mr Meeseeks en simultáneo a buscar `Mega Semillas`. Los Mr Meeseeks harán la tarea que Rick o Morty les indiquen (moverse por el mapa en busca de `Mega Semillas`). Cuando las encuentren, las asimilarán y así se contarán como puntos para un jugador o el otro. En el momento en que un Mr Meeseeks asimile una `Mega Semillas`, este desaparecerá junto con la semilla, dejando un slot

[1] The Ricks must be Crazy (T02E06)

[2] `https://en.wikipedia.org/wiki/Radioisotope_thermoelectric_generator`

libre para llamar a un nuevo Mr Meeseeks. Cada `Mega Semilla` otorgará exactamente 425 puntos al jugador que la encuentre. Por capturar cualquier `Mega Semilla` se obtiene la misma cantidad de puntos.

El mapa o mundo donde se encuentran las `Mega Semillas` va a ser modelado por un rectángulo de 80x40 celdas, con la semillas distribuídas al azar.

Cada celda podrá tener una `Mega Semilla`, o podrá alojar a uno o varios Mr Meeseeks (superposición cuántica).

La figura 1 muestra un ejemplo de como se presentará el juego en pantalla. Lamentablemente no tenemos control de una placa de vídeo y estamos programando en ASM y C, por lo tanto los gráficos resultantes serán un poco más rústicos que los presentados en la figura. No obstante, se presenta toda la información necesaria para el juego, los puntajes de los jugadores, y las posiciones de los Mr Meeseeks y de todas las `Mega Semillas`.

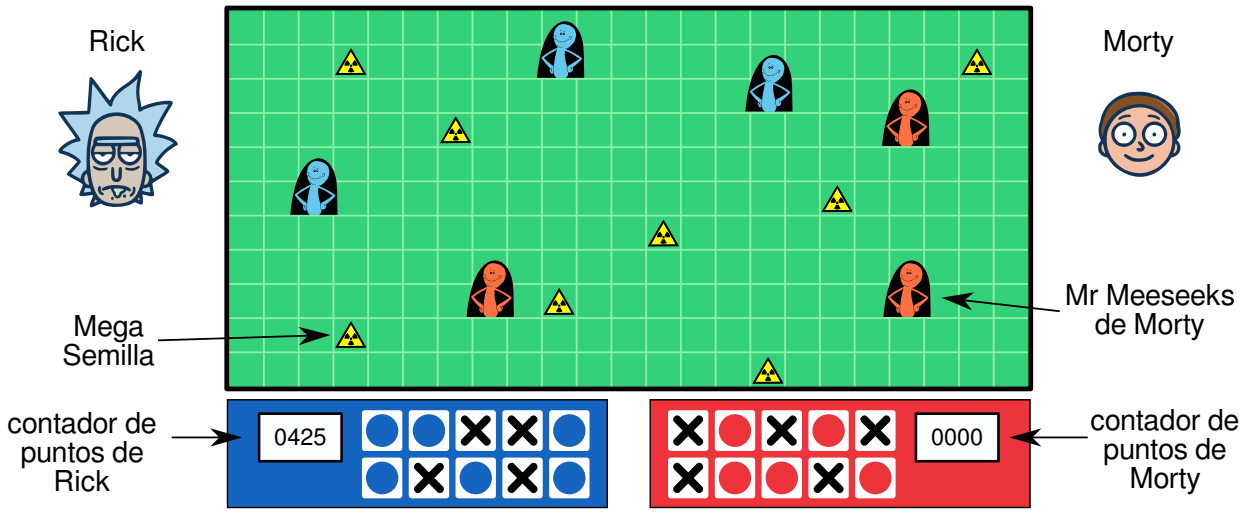


Figura 1: El Juego

Las tareas que ejecutará el sistema serán o bien los jugadores o tareas Mr Meeseeks controladas por alguno de los jugadores. Cada tarea Mr Meeseeks ocupará exactamente 8KiB de memoria (es decir, dos páginas), mientras que los jugadores, o tareas `Rick` y `Morty`, ocuparán exactamente 16KiB. Cada celda del mapa ocupa 8KiB, y permite contener a cualquier tarea Mr Meeseeks. Por otro lado, las tareas jugadores se ubicarán en un espacio de memoria física diferente.

Para crear una tarea Mr Meeseeks y indicarle qué hacer, el jugador debe avisarle al sistema que desea crear una tarea Mr Meeseeks por medio de un llamado al sistema. En este llamado, el jugador pasa como parametro un buffer con el código que la tarea Mr Meeseeks debe ejecutar y la posición del mapa donde desea crearla.

El sistema se ocupará de crear la nueva tarea (si hay algún slot disponible) y agregarla al scheduler para que luego sea ejecutada. Todas las tareas Mr Meeseeks de cada jugador utilizarán el mismo mapa de memoria que el del jugador. Esto permitirá que las tareas de un mismo jugador se puedan comunicarse entre si a través de memoria compartida.

Una vez que son creadas, las tareas pueden llamar a servicios del sistema a fin de reconocer donde se encuentra alguna `Mega Semilla` y moverse hacia la misma, también pueden usar la pistola de portales para mover a un Mr Meeseeks enemigo a otra posición del mapa.

El juego termina cuando se asimilan todas las `Mega Semillas`, gana el jugador que más puntos tiene. Si la tarea principal de un jugador es desalojada del sistema cuando aún quedan semillas, ese jugador pierde automáticamente y el juego finaliza.

0.4.1 Tareas

El sistema dispondrá de cuatro servicios `move`, `look`, `meeseeks` y `use_portal_gun`. Estos se atenderán con un número de interrupción diferente para cada uno. Los parámetros de cada servicio se describen a continuación:

- Syscall `move` int 123

Parámetros	Descripción
`in EAX=x`	Desplazamiento en `x`
`in EBX=y`	Desplazamiento en `y`
`out EAX=worked`	0: No se desplazo, 1: Se desplazo

  Toma como parámetros de entrada en `EAX` y `EBX` los valores de `X` e `Y` respectivamente. Las coordenadas representan la cantidad de casillas que se quiere mover la tarea, notadas como un número con signo, donde los positivos representan hacia abajo y a la derecha, mientras que negativos representan hacia arriba y a la izquierda. El mapa además es circular: una vez que se llega a un limite, se continúa del lado opuesto. Es decir, si se intenta moverse por sobre el límite superior del mapa, el movimiento se continuará en la misma columna desde el límite inferior y viceversa. Análogamente, si el movimiento horizontal supera un límite izquierdo o derecho, se continuará en el opuesto en la misma fila.

  El servicio retorna en `EAX` si se pudo desplazar o no a la celda esperada.

  Nota: si se quiere utilizar la función módulo de C para implementar esto, tener en cuenta que % no es equivalente al resto de la división entera como estamos acostumbrados ya que puede devolver valores negativos. Es necesario salvar estos casos para que el movimiento circular funcione correctamente.

  Si el movimiento implica poner a la tarea Mr Meeseeks en un casillero donde se encuentre una `Mega Semilla`, este la asimilará y ambos desaparecerán del mapa. En otro caso, mover a un Mr Meeseek implica copiar su código hacia la nueva ubicación en el mapa y mapear sus dos páginas virtuales a las dos nuevas páginas fisicas correspondientes. En caso de haber un Mr Meeseeks en el destino, su código será pisado, pero el sistema debe continuar con normalidad.

  Las tareas Mr Meeseeks, como sabemos, no pueden vivir mucho tiempo, ya que a medida que pasa el tiempo, su objetivo en la vida pierde sentido. Por esta razón, las tareas Mr Meeseeks deben encontrar lo más rápido posible una `Mega Semilla`, antes de perder la calma.

  Inicialmente, la syscall permite moverse a 7 celdas de distancia, contando estas como la suma de los valores absolutos de cada coordenada (distancia Manhattan). Por cada tick de reloj sobre una tarea Mr Meeseeks, su capacidad de moverse se ve degradada a razón de 1 celda por cada dos ticks de reloj, hasta un mínimo de 1 casillero por tick.

  Por otro lado, los jugadores `Rick` y `Morty` no pueden moverse sobre el mapa, y por lo tanto no deben llamar a este servicio. Si intentan llamarlo, se deberá generar un error y matar a la tarea que lo llamó.

– Syscall `look` int 100

Parámetros	Descripción
`out EAX=x`	Desplazamiento en `x`
`out EBX=y`	Desplazamiento en `y`

Retorna en `EAX` y `EBX` los valores de `X` e `Y` respectivamente, de las coordenadas **relativas** desde la posición de la tarea Mr Meeseeks a la `Mega Semilla` más cercana. Siempre encuentra una, ya que de no existir, el juego tendría que haber terminado. Por simplicidad, no hace falta que el calculo de la distancia tenga en cuenta que el mundo es circular.

Si este servicio es llamado por los jugadores `Rick` o `Morty` debe retornar -1 en ambas coordenadas.

– Syscall `meeseeks` int 88

Parámetros	Descripción
`in EAX=code`	Código de la tarea Mr Meeseeks a ser ejecutada.
`in EBX=x`	Columna en el mapa donde crear el Mr Meeseeks.
`in ECX=y`	Fila en el mapa donde crear el Mr Meeseeks.

Este servicio puede ser llamado solamente por un jugador, `Rick` o `Morty`. El mismo se ocupa de crear una tarea. Las tareas vivirán dentro del espacio virtual del jugador, para esto se debe disponer de algún slot donde cargar la nueva tarea (maximo 10 Mr Meeseeks simultaneos por jugador). Si no hay espacio, la tarea no será cargada y el servicio retornará un 0 en `EAX`. En el caso contrarío, retornará en `EAX` la dirección virtual en donde fue cargada exitosamente la tarea. El primer parámetro de este servicio es un puntero a un buffer de 4KB correspondiente al código de la tarea que debe ser cargada. Tener en cuenta que este código debe estar declarado dentro del espacio de memoria de usuario de la tarea.

La tarea será creada en la posición del mapa pasada por parámetro, si es válida. En caso de haber una semilla en esa posición, será asimilada automáticamente (sin necesidad de crear la tarea, pero solo si hay slots disponibles), y la función retornará 0 en `EAX`

Las tareas Mr Meeseeks comparten espacio de memoria virtual entre ellas y con la tarea de su respectivo jugador. Al momento de crear una tarea Mr Meeseek, se mapearán dentro del espacio virtual dos páginas de memoria consecutivas que no estén en uso en el rango virtual 0x08000000 - 0x08014000, apuntando a la correspondiente página física del mapa, y la tarea Mr Meeseeks comenzara a correr desde el comienzo de la primera de esas dos páginas. Su stack estará al final de la segunda página.

Cuando la tarea Mr Meeseeks sea desalojada del sistema (ya sea porque asimiló una `Mega Semilla` o cometió una excepción), sus páginas virtuales deberán ser desmapeadas y una nueva tarea Mr Meeseeks podrá ser creada en su lugar.[3]

[3]Se recomienda reutilizar la página asignada a la pila de nivel cero de la tarea que fue desalojada.

– Syscall `use_portal_gun` int 89

  Este servicio no toma ningún parámetro y solo puede ser llamado una única vez por cada tarea Mr Meeseeks que sea creada.

  Los Mr Meeseeks pueden hacer un único uso de la pistola de portales, pero debido a la falta de práctica, no saben usarla con precisión. Al usar la pistola de portales, el sistema moverá una tarea Mr Meeseeks del jugador contrario elegida al azar (si es que hay) a una posición al azar dentro del mapa.

Por otro lado, ninguno de los servicios debe modificar ningún registro, a excepción de los indicados anteriormente. Tener en consideración que luego del llamado a cualquiera de los servicios, la tarea en ejecución "pierde el turno". Es decir, que es desalojada del scheduler, y debe esperar hasta que pueda ser ejecutada nuevamente. En el tiempo entre que la tarea es desalojada y llega una interrupción de reloj para ejecutar la próxima tarea, el sistema debe ejecutar a la tarea `Idle`. Esta última, no tiene más propósito que mantener al procesador realizando alguna tarea, a la espera del próximo punto de sincronismo.

0.4.2 Organización de la memoria

El primer MiB de memoria física será organizado según indica la figura 2. En la misma se observa que a partir de la dirección `0x1200` se encuentra ubicado el *kernel*; inmediatamente después se ubica el código de las tareas Rick y Morty, y a continuación el código de la tarea Idle. El resto del mapa muestra el rango para la pila del kernel, desde `0x24000` a `0x25000` y a continuación la tabla y directorio de páginas donde inicializar paginación para el kernel. La parte derecha de la figura muestra la memoria a partir de la dirección `0xA0000`, donde se encuentra mapeada la memoria de vídeo y el código del `BIOS`.

En un rango más amplio, fuera del primer MiB, memoria física estará dividida en cuatro áreas: *kernel*, *área libre kernel*, *mapa* y tareas Rick y Morty. El área *kernel* corresponderá al primer MiB de memoria, el *área libre kernel* a los siguientes 3 MiB de memoria, el área *mapa* a los siguientes 25 MiB y el area para tareas Rick y Morty serán los siguientes 32 KiB

La administración del área libre de memoria será muy básica. Se tendrá un contador de páginas a partir del cual se solicitará una nueva página. Este contador se aumentará siempre que se requiera usar una nueva página de memoria, y nunca se liberarán las páginas pedidas.

Las páginas del *área libre kernel* serán utilizadas para datos del kernel: directorios de páginas, tablas de páginas y pilas de nivel cero. Las páginas del *mapa* corresponderán a toda la memoria de las celdas donde se crearan las tareas Mr Meeseeks. Se almacenarán los códigos, datos y pila de las tareas vivas, excepto para los jugadores.

Los jugadores, por su parte, estarán ubicados luego de la última dirección física del mapa, según se indica en la figura 2.

La memoria virtual de cada una de las tareas tiene mapeado el *kernel* y el *área libre kernel* con *identity mapping* en nivel 0. Sin embargo, el área *mapa* no está mapeada en ninguna tarea. Esto obliga al *kernel* a mapear el área del *mapa*, cada vez que quiera escribir en el. No obstante, el *kernel* puede escribir en cualquier posición del *área libre kernel* desde cualquier tarea sin tener que mapearla.

El código, pila y datos de las tareas Mr Meeseeks estará compartido en un área de 8 KiB. Para construir una nueva tarea se debe realizar una copia del código que fue pasado por parámetro desde el jugador a el área de memoria del *mapa* en una posición dada.

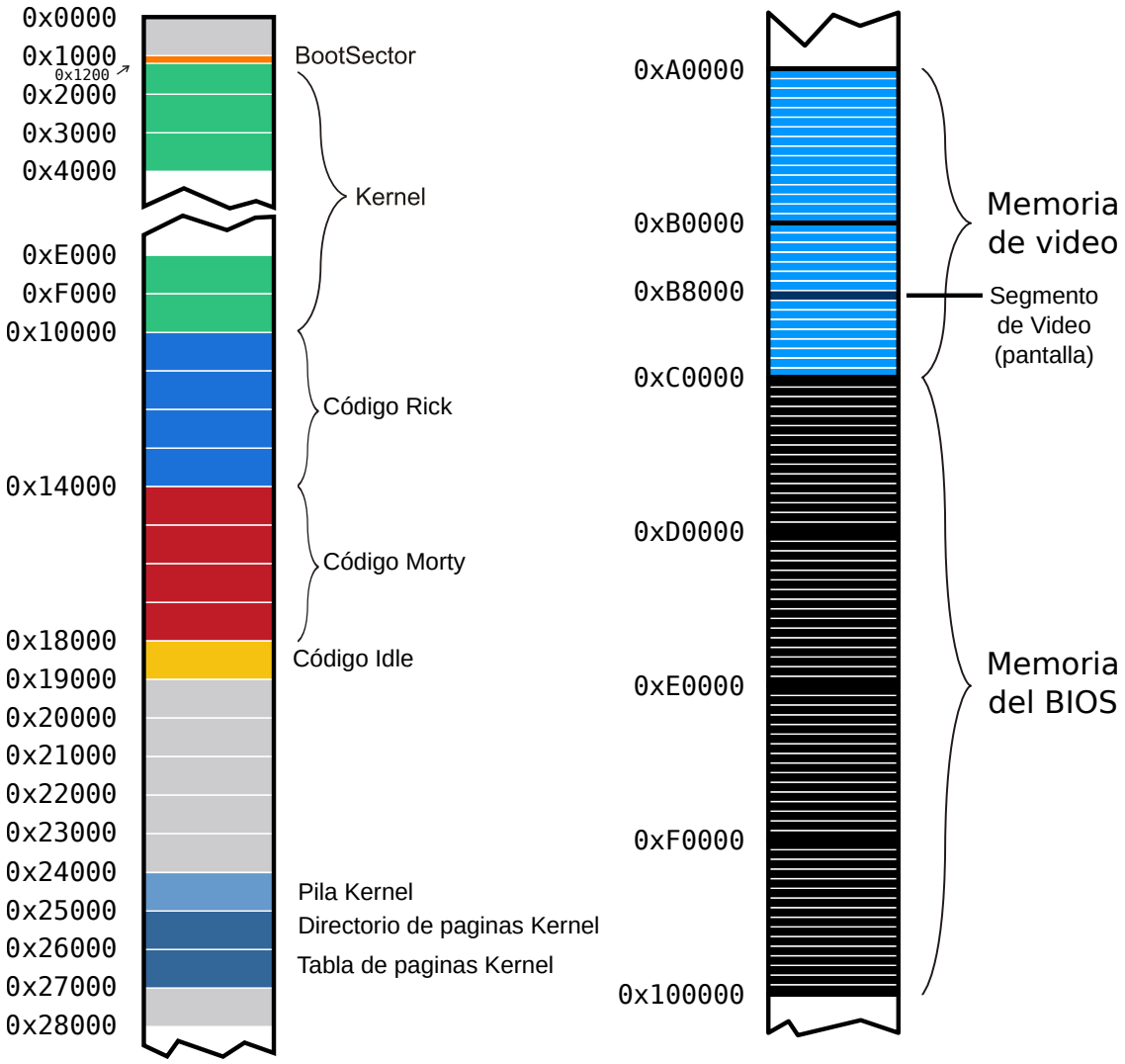


Figura 2: Mapa de la organización de la memoria física del *kernel*

El código de las tareas por su parte estará mapeado en nivel 3 con permisos de lectura/escritura según indica la figura 3.

0.4.3 Scheduler

El sistema va a correr tareas de forma concurrente, durante un tiempo fijo denominado *quantum*. Este será determinado por un *tick* de reloj. Para lograr este comportamiento se va a contar con un *scheduler* minimal que encargará de desalojar una tarea del procesador para intercambiarla por otra en intervalos regulares de tiempo.

Adicionalmente, objetivo es repartir el tiempo equitativamente entre los dos jugadores, por esta razón se ejecutará por cada tick de reloj una tarea de un jugador distinta al del ciclo anterior.

Las tareas serán ejecutadas una a continuación de la otra, respetando la regla anterior. Este proceso se repetirá indefinidamente. Siempre existirán tareas a ejecutar en el sistema, mínimamente se debe ejecutar a los jugadores, tareas `Rick` y `Morty`. El orden de ejecución de las tareas puede ser cualquiera, pero se debe garantizar que nunca sean ejecutadas dos tareas seguidas del mismo jugador y que todas las tareas sean ejecutadas al menos una vez.

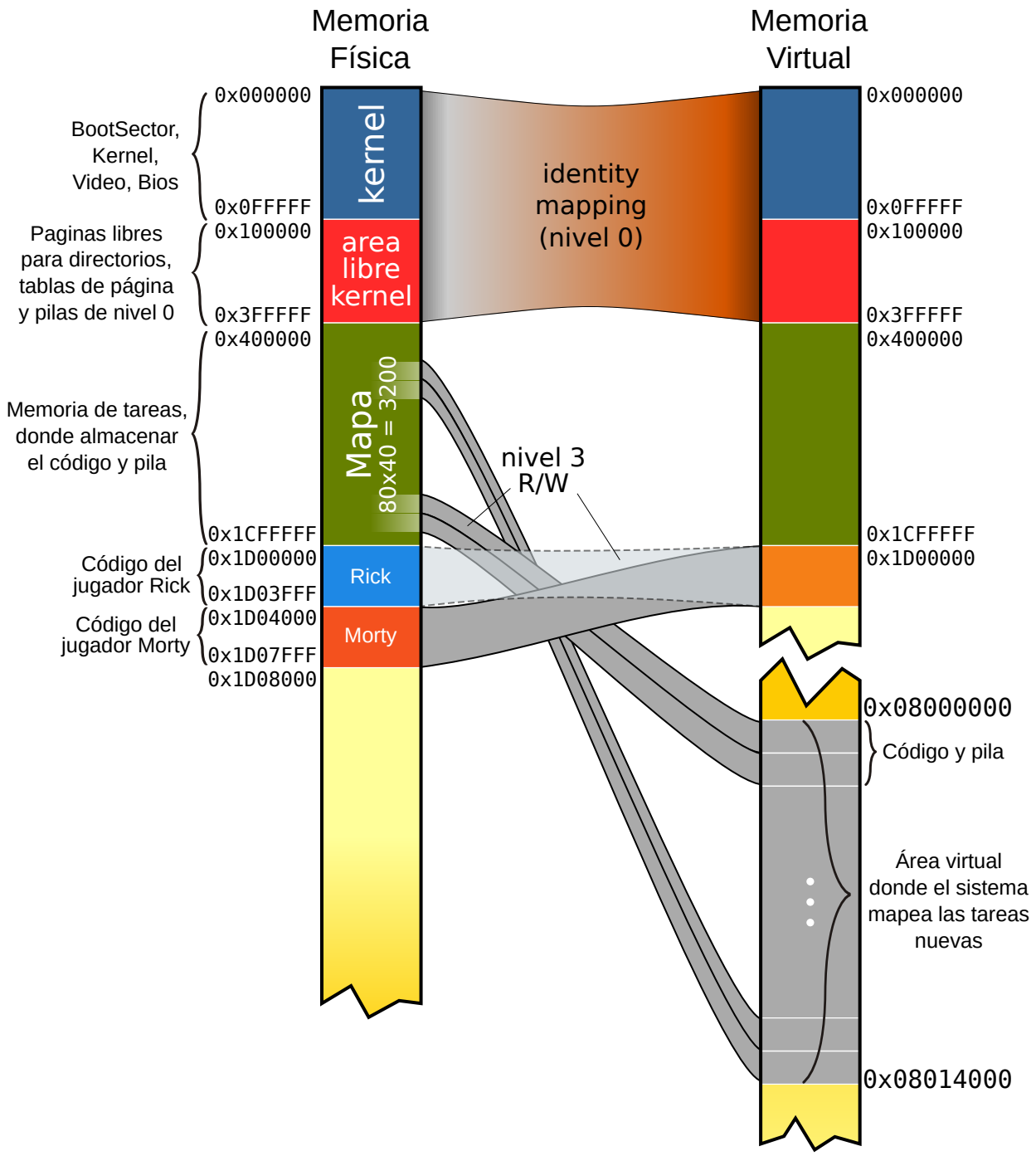


Figura 3: Mapa de memoria de la tarea

Las tareas pueden generar problemas de cualquier tipo, por esta razón se debe contar con un mecanismo que permita desalojarlas para que no puedan correr nunca más. Este mecanismo debe poder ser utilizado en cualquier contexto (durante una excepción, un acceso inválido a memoria, un error de protección general, etc.) o durante una interrupción ocasionada por haber llamado de forma incorrecta a un servicio. Cualquier acción que realice una tarea de forma incorrecta será penada con el desalojo de dicha tarea del sistema (esto en el contexto del juego equivale a que la tarea muera).

Un punto fundamental en el diseño del *scheduler* es que debe proveer una funcionalidad para intercambiar cualquier tarea por la tarea `Idle`. Este mecanismo será utilizado al momento de matar a una tarea, ya que la tarea `Idle` será la encargada de completar el *quantum* actual. La tarea `Idle`

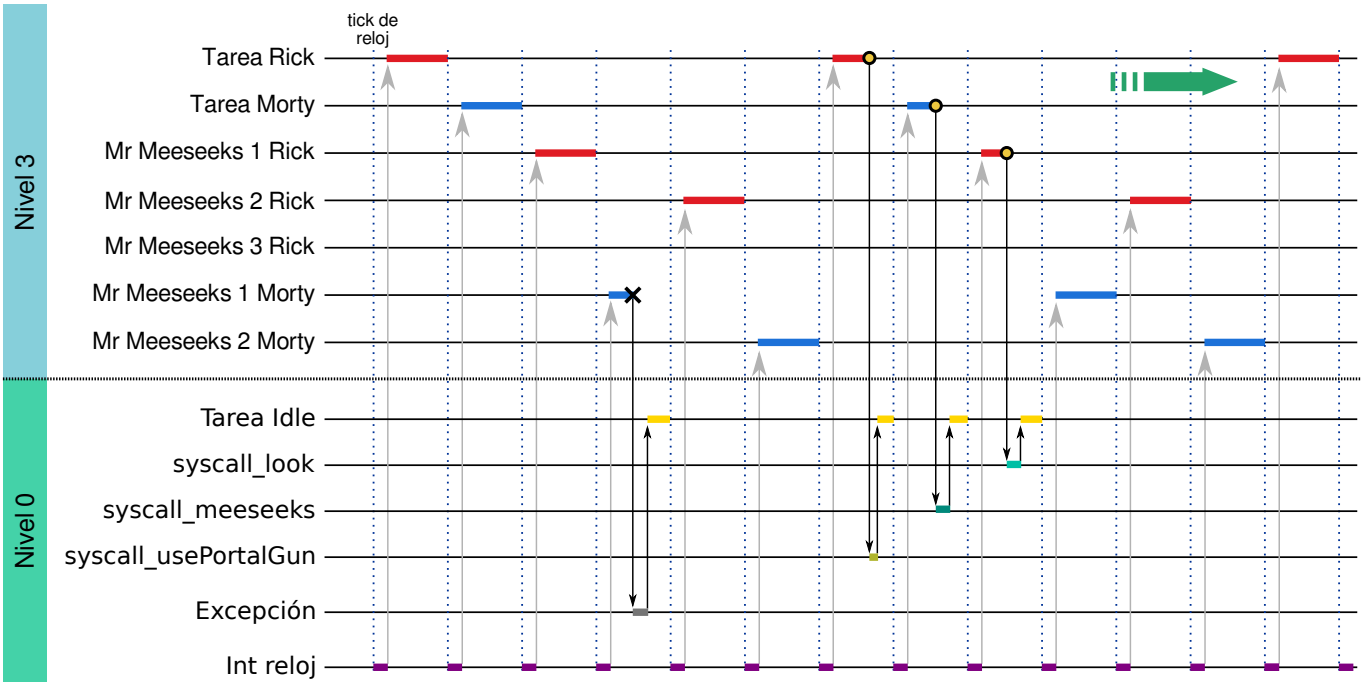


Figura 4: Ejemplo de funcionamiento del *Scheduler*

se ejecutará por el resto del *quantum* de la tarea desalojada, hasta que nuevamente se realice un intercambio de tareas por la próxima que corresponda.

En la figura 4 se presenta un ejemplo con cinco tareas Mr Meeseeks, y las tareas Rick y Morty respectivamente. Se puede observar como las tareas llaman a distintos servicios durante su tiempo en ejecución, y como son desalojadas para ejecutar la tarea `Idle` durante el resto del *quantum*.

0.4.4 Modo debug

El sistema deberá responder a una tecla especial en el teclado, la cual activará y desactivará el modo de debugging. La tecla para tal propósito es la "y". En este modo se deberá mostrar en pantalla la primera excepción capturada por el procesador junto con un detalle de todo el estado del procesador como muestra la figura 5. Una vez impresa en pantalla esta excepción, el juego se detendrá hasta presionar nuevamente la tecla "y" que mantendrá el modo debug pero borrará la información presentada en pantalla por la excepción. La forma de detener el juego será instantánea. Al retomar el juego se esperará hasta el próximo ciclo de reloj en el que se decidirá cuál es la próxima tarea a ser ejecutada. Se recomienda hacer una copia de la pantalla antes de mostrar el cartel con la información de la tarea.

0.4.5 Pantalla

La pantalla presentará el *mapa* en forma de mapa de 80×40. En este mapa se indicará la posición de todas las tareas, la cantidad de puntos acumulados y el estado de todas Mr Meeseeks. Además se deberá indicar qué tareas están corriendo y cuáles lugares están libres, dado que las tareas ya murieron.

La figura 6 muestra una imagen ejemplo de la pantalla indicando cuáles datos deben presentarse como mínimo. Se recomienda implementar funciones auxiliares que permitan imprimir datos en pantalla de forma cómoda. No es necesario respetar la forma exacta de presentar estos datos en pantalla. Se puede modificar la forma, no así los datos en cuestión.

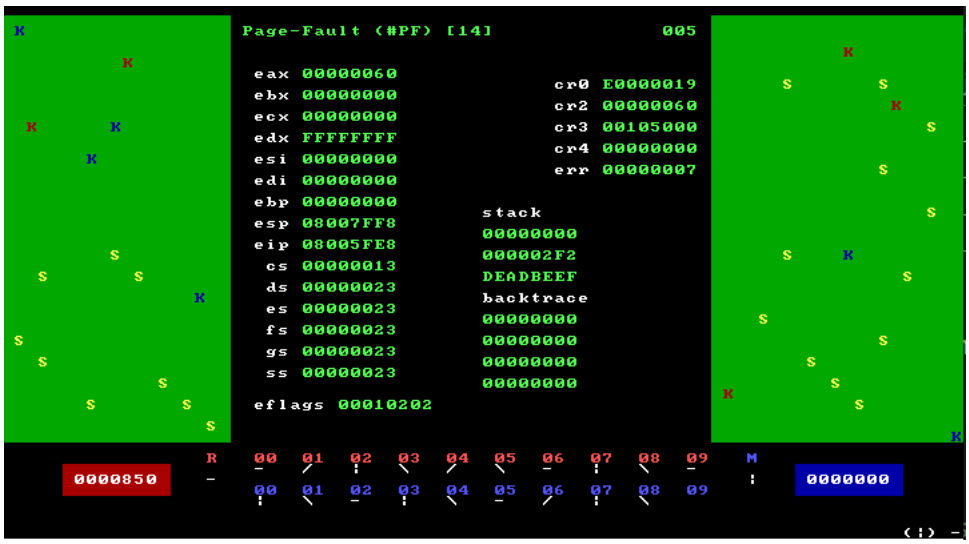


Figura 5: Pantalla de ejemplo de error

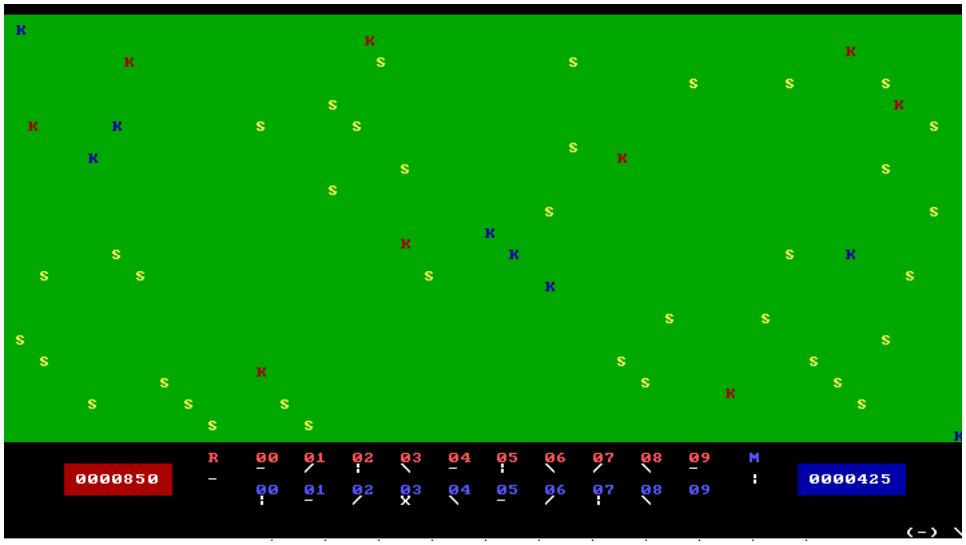


Figura 6: Pantalla de ejemplo

0.5 Ejercicios

Ejercicio 1

a) Completar la Tabla de Descriptores Globales (`GDT`) con 4 segmentos, dos para código de nivel 0 y 3; y otros dos para datos de nivel 0 y 3. Estos segmentos deben direccionar los primeros 201MB de memoria. En la *gdt*, por restricción del trabajo práctico, las primeras 10 posiciones se consideran utilizadas y por ende no deben utilizarlas. El primer índice que deben usar para declarar los segmentos, es el 10 (contando desde cero).

b) Completar el código necesario para pasar a modo protegido y setear la pila del *kernel* en la dirección `0x25000` (es decir, en la base de la pila).

c) Declarar un segmento adicional que describa el área de la pantalla en memoria que pueda ser utilizado sólo por el *kernel*.

d) Escribir una rutina que se encargue de limpiar la pantalla y pintar [4] el área del mapa con algún color de fondo, junto con las barras de los jugadores según indica la sección 0.4.5. Para este ejercicio se debe escribir en la pantalla usando el segmento declarado en el punto anterior. Es muy importante tener en cuenta que para los próximos ejercicios se accederá a la memoria de video por medio del segmento de datos.

Nota: La `GDT` es un arreglo de `gdt_entry_t` declarado sólo una vez como `gdt`. El descriptor de la `GDT` en el código se llama `GDT_DESC`.

Ejercicio 2

a) Completar las entradas necesarias en la `IDT` para asociar diferentes rutinas a todas las excepciones del procesador. Cada rutina de excepción debe indicar en pantalla qué problema se produjo e interrumpir la ejecución. Posteriormente se modificarán estas rutinas para que se continúe la ejecución, resolviendo el problema y desalojando a la tarea que lo produjo.

b) Hacer lo necesario para que el procesador utilice la IDT creada anteriormente. Generar una excepción para probarla.

Nota: La `IDT` es un arreglo de `idt_entry_t` declarado sólo una vez como `idt`. El descriptor de la `IDT` en el código se llama `IDT_DESC`. Para inicializar la `IDT` se debe invocar la función `idt_init`.

Ejercicio 3

a) Completar las entradas necesarias en la `IDT` para asociar una rutina a la interrupción del reloj y otra a la interrupción de teclado. Además crear cuatro entradas adicionales para las interrupciones de software 88, 89, 100 y 123.

b) Escribir la rutina asociada a la interrupción del reloj, para que por cada *tick* llame a la función `nextClock`. La misma se encarga de mostrar cada vez que se llame, la animación de un cursor rotando en la esquina inferior derecha de la pantalla. La función `nextClock` está definida en `isr.asm`.

[4] http://wiki.osdev.org/Text_UI

c) Escribir la rutina asociada a la interrupción de teclado de forma que si se presiona cualquiera de 0 a 9, se presente la misma en la esquina superior derecha de la pantalla.

d) Escribir las rutinas asociadas a las interrupciones 88, 89, 100 y 123 para que modifique el valor de `eax` por $0x58$, $0x59$, $0x64$ y $0x7b$, respectivamente. Posteriormente este comportamiento va a ser modificado para atender cada uno de los servicios del sistema.

Ejercicio 4

a) Escribir las rutinas encargadas de inicializar el directorio y tablas de páginas para el *kernel* (`mmu_init_kernel_dir`). Se debe generar un directorio de páginas que mapee, usando *identity mapping*, las direcciones `0x00000000` a `0x003FFFFF`, como ilustra la figura 3. Además, esta función debe inicializar el directorio de páginas en la dirección `0x25000` y las tablas de páginas según muestra la figura 2.

b) Completar el código necesario para activar paginación.

c) Escribir una rutina que imprima el número de libreta de todos los integrantes del grupo en la pantalla.

Ejercicio 5

a) Escribir una rutina (`mmu_init`) que se encargue de inicializar las estructuras necesarias para administrar la memoria en el área libre de kernel.

b) Escribir dos rutinas encargadas de mapear y desmapear páginas de memoria.

I- `mmu_map_page(uint32_t cr3, uint32_t virtual, uint32_t phy, uint32_t attrs)`
Permite mapear la página física correspondiente a `phy` en la dirección virtual `virtual` utilizando `cr3`.

II- `mmu_unmap_page(uint32_t cr3, uint32_t virtual)`
Borra el mapeo creado en la dirección virtual `virtual` utilizando `cr3`.

c) Escribir una rutina (`mmu_init_task_dir`) encargada de inicializar un directorio de páginas y tablas de páginas para una tarea, respetando la figura 3. La rutina debe mapear 4 páginas virtuales para la tarea Rick o Morty, a partir de la dirección virtual `0x1D00000`. Esta función debe encargarse de copiar el código de la tarea desde la memoria del kernel a la página física correspondiente (`0x1D00000` para Rick y `0x1D04000` para Morty). Sugerencia: agregar a esta función todos los parámetros que considere necesarios.

d) A modo de prueba, construir un mapa de memoria para tareas e intercambiarlo con el del *kernel*, luego cambiar el color del fondo del primer caracter de la pantalla y volver a la normalidad. Este item no debe estar implementado en la solución final.

Nota: Por construcción del *kernel*, las direcciones de los mapas de memoria (`page directory` y `page table`) están mapeadas con *identity mapping*. En los ejercicios en donde se modifica el directorio o tabla de páginas, se debe que llamar a la función `tlbflush` para que se invalide la *cache* de traducción de direcciones.

Ejercicio 6

a) Definir las entradas en la GDT que considere necesarias para ser usadas como descriptores de TSS. Mínimamente, una para ser utilizada por la *tarea inicial* y otra para la tarea Idle.

b) Completar la entrada de la TSS de la tarea Idle con la información de la tarea `Idle`. Esta información se encuentra en el archivo `tss.c`. La tarea `Idle` se encuentra en la dirección `0x00018000`. La pila se alojará en la misma dirección que la pila del kernel y será mapeada con *identity mapping*. Esta tarea ocupa 1 página de 4KB y debe ser mapeada con *identity mapping*. Además, la misma debe compartir el mismo CR3 que el *kernel*.

c) Completar la entrada de la GDT correspondiente a la *tarea inicial*.

d) Completar la entrada de la GDT correspondiente a la tarea `Idle`.

e) Escribir el código necesario para ejecutar la tarea `Idle`, es decir, saltar intercambiando las TSS, entre la *tarea inicial* y la tarea `Idle`.

f) Construir una función que complete una TSS con los datos correspondientes a una tarea. Esta función será utilizada más adelante para crear una tarea. El código de las tareas se encuentra a partir de la dirección `0x00010000` ocupando cuatro páginas de 8kb cada una según indica la figura 2. Para la dirección de la pila se debe utilizar el mismo espacio de la tarea, la misma crecerá desde la base del código de la tarea. Para el mapa de memoria se debe construir uno nuevo utilizando la función `mmu_init_task_dir`. Además, tener en cuenta que cada tarea utilizará una pila distinta de nivel 0, para esto se debe pedir una nueva página del área libre de kernel a tal fin.

Nota: En `tss.c` están definidas las `tss` como estructuras `tss_t`. Trabajar en `tss.c` y `kernel.asm`

.

Ejercicio 7

a) Construir una función para inicializar las estructuras de datos del *scheduler*.

b) Crear la función `sched_next_task()` que devuelve el índice en la GDT de la próxima tarea a ser ejecutada. Construir la rutina de forma devuelva una tarea por jugador por vez según se explica en la sección 0.4.3.

c) Modificar el código necesario para que se realice el intercambio de tareas por cada ciclo de reloj. Cargar las tareas Rick y Morty y verificar que se ejecuten (escriben su reloj cuando estan activas, ejecutar system calls de prueba).

El intercambio se realizará según indique la función `sched_nextTask()`.

d) Modificar las rutinas de interrupciones 88, 89, 100 y 123, para que pasen a la tarea idle cuando son invocadas. Verificar que la tarea idle se ejecuta (escribe su reloj).

e) Modificar las rutinas de excepciones del procesador para que desalojen a la tarea que estaba corriendo y ejecuten la próxima. En caso de que se desaloje una tarea principal, el juego finaliza.

f) Implementar el mecanismo de debugging explicado en la sección 0.4.4 que indicará en pantalla la razón del desalojo de una tarea.

Ejercicio 8

g) Inicializar la pantalla del juego, distribuyendo `Mega Semillas` por el mapa en posiciones aleatorias.

h) Implementar la lógica de la rutina de interrupciones `meeseeks (88)` para crear un nuevo Meeseeks. Verificar que el mapeo se realice en las direcciones correctas, que funcione la lógica de asimilación de `Mega Semillas` y que este esté agregado al scheduler. Actualizar el mapa.

i) Implementar la lógica de la rutina de interrupciones `move (123)` para mover a un Meeseeks. Verificar que el remapeo y asimilación de semillas se realice de forma correcta. Comenzar implementando sin restricciones de movimiento.

j) Implementar la lógica de la rutina `look (100)` para buscar `Mega Semillas`.

k) Implementar la lógica de la rutina `use_portal_gun (89)` para usar el arma de portales.

l) Agregar restricciones de movimiento a la syscall mover.

m) Implementar la lógica de finalización del juego.

Ejercicio 9 (optativo)

a) Crear un par de tareas Rick y Morty que respete las restricciones del trabajo práctico, de no hacerlo no podrán ser ejecutados en el sistema implementado por la cátedra.

   Debe cumplir:

   - No ocupar más de 16 KiB (tener en cuenta la pila)
   - Tener como punto de entrada la dirección cero
   - Estar compilado para correr desde la dirección `0x1D00000`
   - Utilizar los servicios del sistema correctamente

   Explicar en pocas palabras la estrategia utilizada.

b) Si consideran que su tarea puede hacer algo más que completar el primer ítem de este ejercicio, y se atreven a enfrentarse en batalla interdimensional por las `Mega Semillas`, entonces pueden enviar el **binario** de sus tareas a la lista de docentes indicando el nombre de la tarea.

   Se realizará una competencia a fin de cuatrimestre con premios a definir para los primeros puestos.

0.6 Entrega

Este trabajo práctico esta diseñado para ser resuelto de forma gradual. Dentro del archivo `kernel.asm` se encuentran comentarios que muestran las funcionalidades que deben implementarse para resolver cada ejercicio. También deberán completar el resto de los archivos según corresponda.

A diferencia de los trabajos prácticos anteriores, en este trabajo está permitido modificar cualquiera de los archivos proporcionados por la cátedra, o incluso tomar libertades a la hora de implementar la

solución; siempre que el producto final resuelva el ejercicio y cumpla con el enunciado. Parte de código con el que trabajen está programado en ASM y parte en C, decidir qué se utilizará para desarrollar la solución, es parte del trabajo.

Se deberá entregar un informe que describa **detalladamente** la implementación de cada uno de los fragmentos de código que fueron construidos para completar el kernel. Se espera que el informe tenga el detalle suficiente como para entender el código implementado en la solución entregada. Considerar al informe como un documento que ayuda al docente corrector en la tarea de entender su trabajo práctico. En el caso que se requiera código adicional también debe estar descripto en el informe. Cualquier cambio en el código que proporcionamos también deberá estar documentado. Se deberán utilizar tanto pseudocódigos como esquemas gráficos, o cualquier otro recurso pertinente que permita explicar la resolución. Además se deberá entregar en soporte digital el código e informe; incluyendo todos los archivos que hagan falta para compilar y correr el trabajo en Bochs.

La fecha de entrega de este trabajo es **26/11**. Deberá ser entregado a través de un repositorio GIT almacenado en `https://git.exactas.uba.ar` respetando el protocolo enviado por mail y publicado en la página web de la materia. El informe de este trabajo debe ser incluido dentro del repositorio en formato `PDF`.

Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes.