DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming

Organización del Computador II

Segundo Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Patricio Damián Bruno	62/19	pdbruno@gmail.com
Gonzalo Martín Bustos	56/19	bustosgonzalom@gmail.com
Martín Yoel Saied	58/19	martinsaied@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. Introducción

Para el presente trabajo práctico desarrollamos un sistema al que denominamos Ángela Kernel, capaz de ejecutar el juego temático de la serie Rick y Morty ”Low Power”.

En este informe vamos a dar una descripción del funcionamiento del sistema, yendo por cada archivo documentando su cometido y explicando la implementación de sus funciones y rutinas.

2. Kernel

En el archivo kernel.asm se define la secuencia de pasos necesaria para arrancar el sistema(pasar a modo protegido, activado de paginación, etc) para luego pasar a ejecutar las tareas

Lo primero que nos encontramos en el mismo es el pasaje a modo protegido, donde fue necesario habilitar el pin A20 del procesador (se utilizó la instrucción habilitarA20 provista por la cátedra) para poder direccionar más de 1 MB, a diferencia que lo que sucede en modo real. Luego, se cargo la dirección base de la GDT en el registro GDTR mediante la instrucci´on lgdt Se puso en 1 el bit menos significativo del registro CR0 para activar el modo protegido y se realizó un salto a la primera instrucción utilizando el selector de segmento de código de nivel 0. Una vez en modo protegido, se cargaron los registros de segmento de la siguiente manera: ds, es, gs, ss: selector de datos de nivel 0 fs: selector de video (datos de nivel 0). Además, se estableció la base de la pila del kernel en la dirección 0x25000 cargando la misma en los registros esp y ebp..

Luego procederemos a inicializar y limpiar por primera vez la pantalla usando el registro de segmento fs. Acto seguido, inicializamos el manejador de memoria, el directorio de páginas del kernel y cargamos el puntero a ese directorio en cr3. Después habilitamos paginación, hacemos el init de todas las tss, asi como tambien de la idt y el scheduler configuramos el controlador de interrupciones cargamos la tarea inicial, pintamos la pantalla con el mapa del juego y comenzamos con el. Por último activamos las interrupciones y saltamos a la tarea idle.

3. MMU

El archivo de la MMU es uno corto pero que lleva el peso de la asignación y mapeo de páginas para tareas, por lo que es esencial para el correcto funcionamiento de la distribución de memoria. Para la inicialización de la MMU lo único que hacemos es asignar a la variable global que definimos como *proxima_pagina_libre* el puntero a la primer dirección de páginas libres, para poder usarlo después. Siguiendo con una descripción de las diferentes funciones definidas en este archivo, explicamos:

- mmu_next_free_kernel_page: simplemente la utilizamos para conseguir la próxima página libre del kernel, tal como lo indica su nombre. Ésto lo hacemos devolviendo el valor de la variable de *proxima_pagina_libre* y actualizando su valor a la siguiente página (sumándole el tamaño de una página).

- mmu_init_kernel_dir: es la encargada de realizar el mapeo con *identity mapping* de los primeros 4Mb de memoria que son dedicados al kernel. Esto lo hacemos agarrando un puntero a la dirección del Page Directory y de la Page Table dados por la cátedra en el enunciado. Inicializamos en 0 las 1024 entradas de cada una y seteamos la base y los atributos necesarios en la primer entrada del PD, siendo que va a apuntar al PT que mencionamos arriba. También, seteamos los atributos de todas las entradas de la PT. Notar que como estamos haciendo *identity mapping*, armamos un ciclo que hace corresponder la direccion fisica asociada como base, con el índice de cada entrada en la PT. Finalmente, retornamos el puntero a la PD.

- mmu_map_page: esta función hace el mapeo de una dirección virtual a una dirección física, utilizando el CR3 para determinar la dirección de la PD, que los recibimos por parámetro. Ésto lo hacemos primero obteniendo los bits de la dirección de memoria virtual que representan la dirección del PD, y obteniendo también el índice a la PT correspondiente. Con el CR3 recibido por parámetro y el *directoryIdx*, logramos llegar al descriptor de tabla de páginas, chequeamos su bit de present, que si está en cero, nos va a indicar que tenemos que ir a buscar una nueva página para

asignarle una nueva PT, la cual inicializamos con todas sus entradas en 0, y al descriptor de tabla de páginas le seteamos el bit present en 1. Nuevamente, asignamos los atributos necesarios. Notar que no nos preocupamos por los niveles de privilegio porque confiamos en que los parámetros que ingresan son correctos. Finalmente, accedemos a la entrada de la tabla de páginas, y asignamos a la entrada correspondiente, dada por el table index que obtuvimos antes, la dirección física pasada por parámetro, junto con sus atributos. Para terminar con el mapeo, hacemos un tlbflush (proporcionado por la cátedra).

- mmu_unmap_page: seguimos la misma lógica que el map, pero seteamos el bit de present de la PTE en 0.
- mmu_init_task_dir: la utilizamos para crear el esquema de paginación de una tarea. Lo que hacemos es tomar una nueva página que representa un nuevo Page Directory, cuyas entradas inicializamos todas en 0, y al igual que en *mmu_map_page*, seteamos la primer entrada con sus atributos, apuntando además a la tabla de páginas del kernel. Utilizamos *mmu_map_page* para mapear el codigo de la tarea en el directorio de la tarea, y también hacemos *identity mapping* para el área de kernel. Copiamos la memoria de codigo desde la dirección de *code_start* que recibimos por parámetro, a la dirección física apuntada por la virtual que obtuvimos antes para realizar el *identity mapping*, y finalmente desmapeamos éste area que ya no necesitamos mantener mapeada. Para terminar, usamos un tlbflush al igual que en las funciones anteriores.

4. Estructuras de datos del sistema

4.1. GDT y descriptores de segmento

La GDT (Global Descriptor Table) es la estructura de datos que utiliza el sistema para cargar los descriptores de segmentos que organizaran la memoria en ese primer nivel. Así como también contiene los descriptores de TSS que indican dónde se encuentran dichos TSS.

Primero, los descriptores de segmento:
Los descriptores de segmento son parte de la unidad de segmentación, que se utiliza para traducir una dirección lógica a una dirección lineal. Los descriptores de segmento describen el segmento de memoria al que se hace referencia en la dirección lógica. Al ser un sistema con segmentación flat con protección, utilizamos 4 segmentos los cuales mapean los primeros 201 MB de memoria cada uno (que haría del total de memoria del sistema). Estos 4 segmentos se dividen en 2 segmentos de código y 2 segmentos de datos. Otros campos del descriptor que vale la pena mencionar y se repiten en estos 4 segmentos son:

- base_15_0 y base_23_16: Estos bits juntos representan la dirección en la que comienza dicho segmento, que para los nuestros están seteados en 0x0 ya que ocupan desde el principio de la memoria hasta el final(en nuestro caso 201MB)
- limit_15_0 y limit_19_16: Estos indican el límite hasta el cual se extienden los segmentos. En nuestro caso esta seteado en 0xC900, el cual es efectivamente 201 MB
- s: system, indica que no es una estructura de datos del sistema
- g: granularity, indica con qué precisión se mide el límite del segmento, 1 byte o 4kb.
- type: indica el tipo del segmento, los de código son de tipo code/execute y los de datos son data/read/write.
- dpl: indica el nivel de privilegio requerido para acceder a dicho segmento. En nuestro caso poseemos 1 segmento de datos con DPL 0 y otro con DPL 3, mismo para los dos segmentos de código.

Utilizando estos segmentos somos capaces de dividir entre código Kernel ejecutado en nivel 0, el cual utiliza pilas de nivel 0 y ejecutar tambien código de usuario (tareas) las cuales utilizan los segmentos de nivel 3 tanto de datos como de código. Esto permite proteger al código, los datos y la pila del kernel de

exponer información sensible mientras que a la vez reducimos al mínimo el uso de segmentación (aunque aún mantenemos un par más de segmentos para lograr esta protección) para apoyarnos fuertemente en la paginación como método de organización de memoria.

En la GDT no solo guardamos estos descriptores de segmento sino que también guardamos descriptores de TSS. En gdt.c definimos de manera estática los descriptores de segmentos de las 22 TSS (2 para Rick y Morty y 20 para sus meeseeks) aunque sin sus direcciones bases (estas son completadas en el tss_init dentro de tss.c). Todos estos descriptores de TSS son de DPL 0, evitando así que la tarea sea capaz de modificar su propio TSS o saltar de una tarea a otra sin pasar por el Scheduler.

Por último vale la pena aclarar algunos descriptores específicos que fueron completados como decía la consigna

- El descriptor de segmento de video
- El descriptor de TSS de la tarea INIT y de la tarea IDLE. El primero nos indica dónde se encuentra la tss de la tarea INIT necesaria para el mecanismo de conmutación de tareas de intel y el segundo nos dice donde se encontrará la TSS de esta tarea que usaremos siempre que el procesador no tenga ninguna otra tarea que procesar.
- El descriptor nulo en la posición 0

4.2. IDT y descriptores de interrupciones

Para las entradas 0 a 33 de la Interrupt Descriptor Table definimos descriptores donde el selector de segmento es el correspondiente al de código con DPL 0 (el que se encuentra en el índice 10 de la GDT), el handler es la función con nombre _isrX donde X es dicho número del 0 al 33. Esta funcion se encuentra implementada en el archivo isr.asm usando las dos macros que definimos en el mismo, y los atributos son los correspondientes a un descriptor de interrupciones de 32 bits con DPL de supervisor.

Para las entradas 88, 89, 100 y 123 definimos descriptores análogos, pero con DPL 3 y selector de segmento de DPL 3.

4.3. TSS

En este archivo se encuentran las definiciones iniciales de las TSS de las tareas INIT, IDLE, Rick y Morty. Así como también está definido de manera global 2 arrays en los que se encuentran las TSS de los Mr. Meeseeks de Morty y otro para los Mr. Meeseeks de Rick. Este último array tiene una correspondencia directa con los arrays de tareas definidas en el Scheduler. Donde el índice en el array del schduler corresponde al mismo índice - 1 en los arrays de TSS.
La definición de las primeras 4 tareas mencionadas ocurre en tiempo de compilación para casi toda sus campos.

Luego en *tss_init*, la cual es ejecutada en el kernel, nos encargamos de cargar los valores que son obtenidos dinámicamente y de completar todas las TSS de los Mr. Meeseeks.
Dentro de los campos que completamos de esta manera están:

- El CR3 el cual es obtenido a partir de inicializar el mapa de memoria de las tareas Rick y Morty. Luego dichos CR3 son utilizados en las tareas meeseeks también ya que comparten mapas de memorias.
- La pila de nivel de 0 para todas las tareas, para la cual pedimos una página libre del kernel.

Algo para aclarar, debido a que las TSS de cada tarea se encuentran definidas en este archivo, decidimos completar la base de los índices de la GDT de las tareas en tss_init. Obteniendo la GDT es fácil modificar esta información, ya que los índices de los meeseeks corresponden a sus indice en la GDT + un índice base (el índice en la gdt) de Rick o Morty respectivamente. El resto de los campos, como dijimos, en su mayoría fueron completados a partir de la consigna.
Una última aclaración, en tss.c se puede cómo creamos las tss de los Mr Meeseeks con EIP y ESP en 0. Esto se debe a que luego, en game.c a la hora de efectivamente crear una nueva tarea Mr. Meeseeks

seteamos el EIP y el ESP mapeando el código que ejecuta la tarea desde el área virtual de los Mr. Meeseeks (organizado a partir de sus índices) a la celda en la que se encontrara en el mapa. Daremos más detalles a la hora de explicar game.c.

5. Scheduler

En este archivo primero que nada, nos definimos 2 arreglos (inicializados en 0) de 11 elementos cada uno, llamados *tasks_rick* y *tasks_morty*, los cuales se encargarán de almacenar estructuras de datos con información necesaria para todas las tareas de Rick y de Morty respectivamente. Es decir, almacenarán structs de las tareas principales de cada uno de los jugadores y sus 10 posibles *"subtareas"* MrMeeseeks.
Estas estructuras están declaradas en el archivo sched.h y proveen datos sobre:

- idx_gdt: el índice del descriptor de la tarea en la gdt.
- es_jugador: si este bit está encendido, la tarea es Rick o Morty, si esta apagado es un MrMeeseeks de alguna de las tareas anteriores.
- y,x: posiciones de la tarea sobre los ejes del mapa del juego.
- cap_movimiento: capacidad de movimiento de la tarea (válido en caso de ser MrMeeseeks).
- uso_portal_gun: indica si la tarea ya utilizó su portal gun o no.
- status: indica si la tarea es parte de las tareas activas para que el scheduler sepa si debe alocarla para que sea ejecutada o no. De no serlo, la saltea.
- isrNumber: atributo utilizado para el display del reloj de cada tarea.

Al inicializar el scheduler, seteamos los valores correspondientes a las tareas de Rick y Morty, con el bit de status en 1 para ambos, y asignándole a cada uno de los elementos del arreglo su índice correspondiente en la GDT (junto con su nivel de privilegio).

Creamos tambien ciertas variables globales que nos van a ayudar a conocer el estado de las tareas, la tarea actual y si el juego finalizó.

Adentrándonos más en la funcionalidad del scheduler en sí, tenemos una serie de funciones que son útiles para distintas ocasiones en el juego:

- _maybe_degradar_cap: simplemente se encarga de disminuir la capacidad de movimiento de la tarea actual, utilizando el atributo antes mencionado de *cap_movimiento*, que tal como lo indica el enunciado se disminuye cada 2 clocks sobre la tarea indicada. En el caso de que la tarea sea Rick o Morty y no uno de sus MrMeeseeks no tiene efecto.
- sched_next_task: ésta función es muy importante porque es la encargada de devolverme el índice de la gdt de la siguiente tarea en la lista de tareas del scheduler (formato round-robin). Aquí primero chequeamos si la tarea anterior a la que debemos devolver fue una de Rick o una de Morty, y hacemos un *"toggle"* de la variable global *ultimo_fue_rick* segun corresponda, y luego (tambien dependiendo del caso anterior para determinar el caso siguiente) buscamos la siguiente tarea con status=1 para saber cual es la que corresponde devolver para hacer el switch. Notamos también que como estamos en formato Round-Robin, si se llega al final de la lista de tareas, se vuelve a la primer posición del arreglo.
- reset_esp: reseteamos el esp0 de la tarea corriéndolo hasta la base de su pila.
- desalojar_tarea: usamos ésta función para desalojar la tarea actual, seteando su status en 0 (ie, sacándola de la lista de tareas disponibles para alocar con el scheduler), reseteando el esp con la funcion explicada arriba y usando el cr3 actual para desmapear el area de memoria de la tarea, haciendo el cálculo de la direccion base de la memoria virtual, sumado a 2 veces el tamaño de las páginas (ya que sabemos que cada tarea ocupa 2 páginas) y usamos el índice de la tarea para terminar de ubicar la dirección de memoria a desmapear. En caso de que la tarea a desalojar sea la de Rick o Morty, directamente devolvemos el selector de la tarea IDLE.

- desalojar_tarea_indice: se utiliza únicamente para el caso de *use_portal_gun*, desalojando una tarea cuyo índice es pasado por parametro. Se desaloja de la misma manera que se desalojaba en la funcion *desalojar_tarea*, pero en este caso siendo una funcion void y usando el índice recibido para ubicar la tarea dentro de los arreglos de tasks mencionados anteriormente.

6. ISR

Este es un archivo escrito en código de máquina, aquí definimos las rutinas de atención a interrupciones que son indicadas desde los descriptores de la IDT Así como también definimos las rutinas de atención a excepciones (que también se definen en la IDT).

El archivo se divide en 3 partes distintas donde se definen e implementan distintas rutinas. Primero vemos la definición de 2 macros, *ISR_CON_ERROR_CODE e ISR* las cuales definen la rutina de atención a las excepciones. La primera es utilizada para definir la rutina a excepciones que devuelven un errorCode en la pila a la hora de cambiar el stack por uno de privilegio 0. La segunda es utilizada para definir la rutina de excpeción que no utiliza ningún tipo de errorCode, por lo que la estructura de la pila que queda es distinta.

La única diferencia entre ambas macros es la primera instrucción donde en la utilizada para excepciones con errorCode pasamos el valor ubicado en la posición a la que apunta ESP que, como la documentación de Intel indica, es donde se encuentra el errorCode.
El resto de la implementación de las macros es exactamente igual. Basado en la variable global de modo_debug decidimos si imprimir o no la excepción en pantalla pasando por la pila como párametro todos los registros e información a mostrar. De no tener el modo debug activo salteamos este call a imprimir_excepción y pasamos a desalojar la tarea que generó la excepción. En la etiqueta error_enable se encuentra un byte que indica si hubo una excepción y luego utilizar esta información para parar la ejecución del scheduler.

En la segunda parte del archivo vemos las rutinas de atención a las interrupciones de teclado y clock. La rutina de interrupción de reloj primero chequea si error_enable esta en 1, como dijimos anteriormente. En caso de estarlo, detiene cualquier tipo de switch de tarea. Luego tenemos dos calls, uno para actualizar el reloj de abajo a la derecha y otro para actualizar el reloj que se encuentra debajo del índice de la tarea que se está ejecutando para ser capaces de ver el funcionamiento y el cambio de tarea sin necesidad de utilizar breakpoints.
Después de esto chequeamos si el juego finalizó, y de ser así, imprimimos un mensaje en pantalla indicando el resultado del juego y dejamos el juego en un while infinito.
Luego, en caso de que el juego no haya terminado aún, buscamos qué tarea es la próxima a ejecutarse utilizando sched_next_task, verificamos que la tarea devuelta no sea la actual. En caso de cumplirse las condiciones, cargamos el selector de segmento que apunta al descriptor de TSS dentro de la GDT para de esta forma producir el task switch.
La rutina de interrupción de teclado solo se preocupa por una cosa: una vez cargado en AL lo que se encuentro en el puerto 0x60, chequeamos si lo que fue apretado fue una ”Y”. En caso de haber presionada dicha tecla y estar en modo_debug, ejecutamos restaurar pantalla (esta función solo restaura la pantalla en caso de que haya habido una excepción, sino no hará nada) y desactivamos el error_enable.
Si no se encontraba el modo_debug activo, lo activaremos.

Por último, en el archivo tenemos las rutinas de atención a las syscalls que pueden producir las diferentes tareas. No nos detendremos a explicar qué hace cada una ya que éstas se encuentran implementadas en game.c y aquí solo nos encargamos de atender la interrupción correctamenete, preservar todos los registros y pasar los parametros a la función de C que se encargara de ejecutar la lógica.
Como añadido, en este archivo decidimos implementar el backtrace, el cual se encuentra en el final del mismo. Esta rutina toma dos parámetros: el primero es un puntero a un arreglo donde se deben guardar las direcciones que vamos encontrando. El segundo parámetro es el valor que tenía el registro ebp cuando

se empezó a ejecutar el handler de excepción. El algoritmo va buscando el base pointer anterior a donde está parado y también obtiene la return address asociada a ese stackframe y la guarda en el arreglo de salida. Esto se repite un máximo de 5 veces, pero corta si el base pointer es nulo, no es múltiplo de 4, o si está fuera del rango del código de los jugadores.

7. Game

En este archivo decidimos poner toda la lógica de funcionamiento del juego y de los servicios expuestos a las tareas. También nos vamos a encontrar con 3 variables locales: un arreglo de tamaño 40 para almacenar las posiciones de las 40 megasemillas y si fueron absorbidas, y dos variables numéricas que llevan cuenta del puntaje de cada jugador.

La función _move_meeseeks toma un directorio de páginas, un par de coordenadas x, y de origen y un par de coordenadas x, y de destino, para mover el código y pila de un Mr. Meeseeks a su nueva posición. Para lograr esto, nosotros pensamos la porción de la memoria correspondiente al mapa como una matriz de 80x40. Usando las coordenadas viejas y nuevas calculamos la posición en la memoria física que corresponde a cada celda en cuestión. Luego, habiendo calculado ambas direcciones físicas, mapeamos temporalmente 2 páginas a partir de cada dirección con identity mapping en el cr3 pasado como parámetro (esto no es un problema ya que el espacio virtual del 0x4000000 al 0x1CFFFFF no está mapeado a priori por ningún directorio) para luego copiar todo el contenido de la celda vieja a la nueva (esto es: código y pila de nivel 3) y dejar la celda vieja llena de ceros. Hecho esto desmapeamos las cuatro páginas mencionadas anteriormente. Finalmente, buscamos la tss del Mr. Meeseeks que deseamos mover y su índice dentro de su arreglo de 10 tss's de Mr. Meeseeks. Dado el índice del 0 al 9 en su correspondiente arreglo de tss's, nosotros decidimos calcular la dirección virtual de un Mr. Meeseeks dado como MEESEEKS_VIRT_START + 2 * PAGE_SIZE * índice. Llamamos mem_slot a esta variable, y usando el cr3 de la tss del Mr. Meeseeks desmapeamos las 2 páginas a partir de la dirección virtual mem_slot y luego la mapeamos usando como dirección física la nueva celda dentro del mapa donde estará la tarea.

En la función game_checkEndOfGame primero recorremos todas las semillas para ver si queda alguna presente, y nos fijamos si la tarea Rick y/o la tarea Morty fueron desalojadas. En caso de que alguna de estas condiciones se cumpla, imprimimos en la pantalla la información correspondiente y nos quedamos ciclando indefinidamente. Como esta función se ejecuta antes de sched_next_task(), en caso de que el juego termine siempre se quedará en el while y cuando haya una interrupción de reloj entraremos a esta función de vuelta antes de tener la oportunidad de cambiar de tarea, por lo que el juego quedará tildado en la pantalla de finalización.

La función move contiene toda la lógica necesaria para atender a la interrupción que lleva el mismo nombre. Primero checkea que no la hayan llamado ni Rick ni Morty. De ser así termina y desaloja la tarea actual (la que la llamo: Rick o Morty). En caso contrario calcula cuánta distancia manhattan pretende moverse el Mr. Meeseeks. Si es mayor a la que tiene permitido, el servicio termina y devuelve 0. La razón por la cual le tomamos el valor absoluto al campo de capacidad de movimiento es porque decidimos (para evitar usar más campos en nuestros structs y más variables globales, que bastantes tiene el TP) que el valor, por cada vez que la tarea sea ejecutada, se disminuya de la siguiente manera: 7, -7, 6, -6, 5, -5, ... hasta llegar a 1. Luego de este checkeo, borra la celda actual del Mr. Meeseks y actualiza los campos x e y de su task con las nuevas coordenadas. Si hay una semilla en el campo actual, la absorbe y la tarea es desalojada. Caso contrario, imprime una "k" en la nueva celda y llama a _move_meeseeks para mover el código y datos de su tarea a la nueva posición que corresponda.

La función create_mrmeeseeks toma un puntero al código de la función que controla al nuevo Mr. Meeseeks, y un par de coordenadas x, y para ubicarlo. Si el puntero al código esta fuera del rango virtual 0x1D00000, 0x1D03FFF, entonces la llamada al servicio es inválida y la tarea que la llamó sera desalojada. También sucederá esto si llama a este servicio un Mr. Meeseeks. Si las coordenadas están fuera del rango de la pantalla, el servicio termina y devuelve 0. Acto seguido, buscamos en el arreglo correspon-

diente de tasks (depende si llamó Rick o Morty al servicio) la primer tarea Mr. Meeseeks que tenga status 0. Si no encontramos ninguna disponible el servicio devuelve 0, y sino nos quedamos con el índice, al cual llamaremos i. Luego checkeamos si en la posición pasada había una semilla, y si la había procedemos tal como hicimos en move. Luego de estos checkeos, procedemos a crear el nuevo Mr. Meeseeks. Calculamos la posición de la memoria física correspondiente a la posición x, y, mapeamos dos páginas a partir de esa posición con identity mapping, y copiamos en la primer página el código a partir del puntero y seteamos la segunda página (donde estará la pila) con ceros. Hecho esto desmapeamos la dirección de la celda y hacemos el mismo procedimiento de mem_slot que describimos en move. Por último, seteamos el eip de la tss correspondiente en la primera posición de las direcciones virtuales asignadas a este Mr. Meeseeks (más conocida como mem_slot), seteamos el esp en el final de la segunda de las páginas asignadas a la tarea y seteamos todos los valores iniciales en el struct task correspondiente al Mr. Meeseeks que crearemos.

La función use_portal_gun primero checkea que no haya sido llamada por Rick, Morty, ni un Mr. Meeseeks que ya la haya usado. Para elegir un enemigo a teletransportar primero probamos con con número aleatorio y si el enemigo encontrado no era una tarea activa, hacemos una búsqueda lineal hasta encontrar el primer enemigo disponible. Luego elije un par de coordenadas válidas y checkea si hay una semilla en esa posición. Si lo hay, lamentablemente el enemigo absorberá la semilla y será desalojado. Si no hay una semilla en ese lugar, entonces llamamos a _move_meeseeks para transladar al Mr. Meeseeks enemigo a su nueva celda.

La función look toma dos punteros a enteros, donde guardará los resultados. Si quien llamó a la función es Rick o Morty, devuelve -1 en ambas variables. En caso contrario, hacemos una búsqueda lineal para encontrar la semilla a menor distancia manhattan del Mr. Meeseeks que llamó al servicio.

8. Conclusión

Como conclusión del informe, logramos crear y utilizar este básico sistema operativo que nos ayudó a aprender sobre los fundamentos de System Programming. Entendimos lo que conlleva desarrollar sistemas operativos modernos, mucho más complejos y en muchas ocaciones utilizando otros mecanismos que difieren en gran medida de lo implementado en este Trabajo Práctico.