DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Tierra Pirata

Trabajo Práctico III - Programación de sistemas

7 de diciembre de 2017 — Organización del Computador II

Grupo: Ariane 5

Integrante	LU	Correo electrónico
Greco, Luis	150/15	luifergreco@gmail.com
Hertzulis, Nicolás	811/15	nicohertzulis@gmail.com
Ramos, Ricardo	841/11	rikigerman@yahoo.com.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Introducción

Un sistema operativo es el conjunto de programas de un dispositivo informático (laptop, celular, etc.) que maneja los recursos de hardware y provee servicios a los programadores de aplicaciones mediante interrupciones de software. Los sistemas operativos se ejecutan en modo privilegiado, aunque puede que una parte se ejecute en modo usuario.
El núcleo o kernel es la parte más fundamental del sistema operativo, que se encarga de hacer los primeros pasos para el arranque del sistema y de proveer las rutinas de atención a interrupciones de software, que utilizan los programadores de aplicaciones.
El kernel se define como la parte del sistema operativo que se ejecuta en modo privilegiado.
En este trabajo educativo nos proponemos hacer un sistema operativo muy básico para entender los fundamentos de la programación de sistemas operativos.
Para ello completamos diversos fragmentos de un esqueleto de kernel provisto por los docentes.

2. Desarrollo

2.1. Ejercicio 1

2.1.1. Segmentos de código, datos y de pantalla

El sistema operativo utiliza dos segmentos de código de nivel 0 y 3, dos segmentos de datos de nivel 0 y 3 y un segmento que describe el área de la pantalla de video. Declaramos los cinco segmentos en la Tabla de Descriptores Globales (GDT), cada uno con su correspondiente descriptor. La GDT es un arreglo en memoria cuyas entradas ocupan 64 bits y tienen una estructura particular (ver figura 1). Para ingresar los datos con comodidad utilizamos un struct de C. La primer entrada está ubicada en la posición con índice 8. Todos los segmentos excepto el de video direccionan los primeros 500 MB de la memoria, es decir que se superponen.

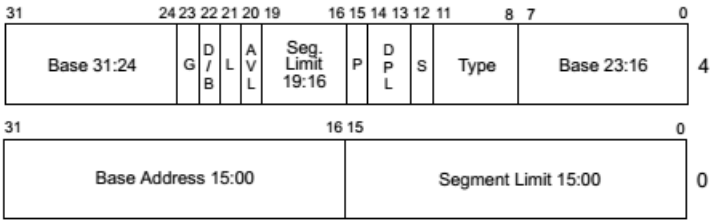

Figura 1: Descriptor de segmento.

- Base, límite y granularidad: La base es 0x0, el límite es 0x1F3FF y la granularidad es 1. La granularidad activada hace que tratemos a la memoria como bloques de 4K, por lo tanto el límite con granularidad corresponde a la cantidad de bloques de 4K menos uno. Para direccionar los primeros MB, necesitamos la base en cero. El límite es 0x1F3FF porque en decimal es 127999, ya que necesitamos 128000 bloques de 4KB para llegar a 500 MB. Todo esto es para los segmentos de código y datos. Para descriptor de segmento de pantalla debemos especificar como base `0xB8000` y para el límite necesitamos resolver (80 * 50 * 2) - 1, donde 80 es la cantidad de celdas, 1 celda == 2 bytes, en fila de pantalla y 50 es cantidad de celdas en columna. Es decir que el límite nos la cantidad de bytes que necesitamos para escribir en cualquier parte de la pantalla. Luego el valor del límite es `0x1F3F`. La granularidad está cero ya que tiene menor tamaño (no necesita bloques de 4KB para direccionar toda su memoria).

- Tipo: 0x0A en los segmentos de código (lectura, ejecución) y 0x02 en los segmentos de datos y de video (lectura, escritura).
- Nivel de privilegios: 0x0 en los segmentos de datos y código que utiliza en kernel, así como en el segmento de video (que también es de uso exclusivo del kernel). Estos son los segmentos de nivel cero mencionados anteriormente. 0x03 en los segmentos de código y datos que utilizan las tareas, o sea el usuario, que corresponden a los segmentos de nivel tres mencionados anteriorente.
- Sistema: 0x1 (que significa desactivado) pues no son segmentos de sistema sino de código y datos.
- Presente: 0x1.

2.1.2. Modo protegido y pila del kernel

Realizamos los siguientes pasos:

- Habilitar A20 para poder acceder a direccioner superiores a 2^{20} bits.
- Inicializar la GDT, cuyas entradas fueron ingresadas a partir de una dirección de memoria arbitraria, mediante la instrucción lgdt.
- Activar el bit menos significativo de CR0.
- Saltar a segmento:modoprotegido, donde segmento corresponde al índice del segmento de código de nivel 0 en la GDT corrido 3 ceros a la izquierda y modoprotegido corresponde a la dirección de memoria donde arranca el código que se ejecutará a continuación (ya en modo protegido).
- Establecer los registros selectores de segmento datos (ds, es, gs y ss) en el índice del segmento de datos de nivel cero en la GDT corrido 3 bits a la izquierda.
- Establecer el registro selector de segmento de video (fs) en el índice del segmento de video en la GDT corrido 3 bits a la izquierda.
- Establecer la base de la pila en 0x27000.

2.1.3. Limpieza e inicialización de la pantalla

Para limpiar la pantalla hicimos una función en ensamblador que utiliza el segmento de video y otra en C que utiliza el segmento de datos. La función en ensamblador recorre byte por byte y la función en C recorre celda por celda (2 bytes) y llama a la función screen_pintar en cada iteración. Ambas funciones dejan la pantalla en negro.
Para inicializar la pantalla llamamos a la función screen_inicializar que fue completada por nosotros, llamando a la función para limpiar la pantalla y luego a la función screen_pintar_rect para las distintas secciones.

2.2. Ejercicio 2

2.2.1. Rutinas de atención a las excepciones del procesador

Cada interrupción debe tener su entrada en la Tabla de Descriptores de Interrupción (IDT). Las primeras 32 posiciones de esta tabla (índices 0 a 31) corresponden a las excepciones del procesador.
Así como en la GDT, para las entradas de la IDT utilizamos un struct de C, cuyos campos y valores son los siguientes:

- Selector de segmento: 0x40 que corresponde al índice del segmento de código del kernel en la GDT corrido 3 bits a la izquierda (índice 8).
- Desplazamiento: La dirección donde comienza el código de la rutina de atención. Es una dirección distinta para cada rutina. Las rutinas están definidas con una macro en ensamblador cuyas etiquetas están declaradas como símbolos globales para poder referenciarlas luego desde C.
- Atributo: Es un campo que agrupa varios valores. En nuestro caso el valor es 0x8E00. El 8 es un valor hexadecimal que en binario es 1000. El 1 es el bit de presente, los dos bits siguientes son el DPL (nivel 00 por ser excepciones del procesador) y el bit menos significativo debe estar en cero para interrupciones. La E también es un valor hexadecimal que corresponde al *gatetype* y en este caso significa puerta de interrupción de 32 bits. Así, el valor de atributo es 0x8E y luego se agregan dos ceros a la derecha porque debe estar corrido siempre 8 bits (los ceros son fijos).

Las entradas de la IDT se cargan desde una dirección de memoria arbitraria. Una vez cargadas, inicializamos la IDT utilizando la instrucción lidt (load IDT).
Las rutinas solo muestran el número de excepción (que corresponde al índice de la IDT) y luego quedan en un ciclo infinito.

2.3. Ejercicio 3

2.3.1. Paginación

La paginación es un mecanismo que permite tratar a la memoria en bloques de tamaño fijo, que en este caso son de 4KB y asignar las direcciones físicas de cada página a un rango de mismo tamaño de memoria virtual.
Un rango de direcciones de tamaño de una página de memoria virtual puede estar asignado a distintas páginas de memoria física, cada asignación en un directorio de páginas distinto.
Un directorio de páginas tiene 1024 entradas que corresponden a 1024 tablas de páginas. Una tabla de páginas tiene 1024 entradas que corresponden a 1024 páginas.

2.3.2. Directorio y tablas de páginas del kernel

El kernel tiene su propio directorio de páginas ubicado en la dirección física 0x27000. Este se va a encargar de las páginas de la zona del kernel, que corresponde a las direcciones físicas 0x00000000 a 0x003FFFFF. Estas direcciones serán asignadas a otras direcciones de memoria virtual, que casualmente en este caso coinciden con las direcciones físicas, por eso decimos que es la asignación identidad, pues la función de asignación es la función identidad.
Para asignar la memoria debemos calcular cuántas páginas necesitamos. Para eso hacemos la siguiente cuenta: (0x003FFFFF + 0x1) / 0x1000 que es igual a 0x400. 0x1000 es el tamaño de una página (en decimal es 4096 pues una página es de 4KB). El resultado 0x400 en decimal es 1024. Por lo cual necesitamos 1024 páginas para completar la zona del kernel. Casualmente 1024 es el tamaño de una tabla de páginas, por lo cual el directorio de páginas del kernel tendrá todas las entradas vacías excepto la primera, que hará referencia a una tabla que estará completamente llena. La tabla de páginas estará en la dirección física 0x28000.
La función que se encarga de inicializar el directorio del kernel es la función *mmu_inicializar_dir_kernel* y realiza los siguientes pasos:

- Llenar las 1024 entradas del directorio en cero. En particular el bit de presente está en cero, que en nuestro sistema operativo significa que la memoria correspondiente a esa tabla no está asignada.
- Modificar la primera entrada con los siguientes valores: 1) Bit de presente activado. 2) Bit de lectura-escritura activado. 3) Base en 0x27 que es la dirección física de la tabla (0x27000) corrida 12 bits a la derecha.

- Para cada entrada de la tabla, poner una entrada con los siguientes valores: 1) Bit de presente activado. 2) Bit de lectura-escritura activado. 3) Base en el valor i donde i es un número entero entre 0 y 1023. La dirección física es en realidad (i << 12), pero como luego hay que volverla a correr 12 bits para ingresarla a la estructura, (i << 12) >> 12 es equivalente a i.

2.3.3. Activación de paginación

Para tener el mecanismo de paginación funcionando, realizamos los siguientes pasos:

- Realizar las asignaciones de memoria deseadas (como mínimo la zona del kernel) tal como explicamos en la sección anterior. En nuestro sistema operativo lo hacemos llamando a *mmu_inicializar_dir_kernel*.
- Poner la dirección física del directorio en el registro de control 3 (CR3).
- Activar el bit más significativo del registro de control 0 (CR0). Este es el paso que activa el mecanismo.

Durante la ejecución del sistema operativo se pueden modificar las asignaciones de memoria del directorio en uso o de otros directorios.

2.4. Ejercicio 4

2.4.1. Manejador de memoria

Tenemos un manejador de memoria al que le podemos pedir una página libre. El mecanismo es muy básico. Tenemos una variable inicializada en la posición física 0x100000. Cuando alguien llama a la función para pedir la dirección de una página libre, el manejador devuelve el valor de esta variable y luego la incrementa en 0x1000 que es el tamaño de una página, así en la próxima llamada a la función se devuelve la página que está inmediatamente más abajo, que se asume libre. Con este mecanismo, si borramos una página física de todos los directorios la misma no podrá volver a ser asignada nuevamente, porque el manejador no se enterará que está libre, ya que la variable de la próxima página libre siempre crece.

2.4.2. Asignación de páginas

La función mmu_mapear_pagina se encarga de asignar un rango de direcciones de memoria virtual a memoria física en un directorio determinado, siendo ese rango el correspondiente a una página. Las direcciones físicas asignadas serán, por lo tanto, el rango [A, A+0xFFF] donde A es la dirección de comienzo de la página (múltiplo de 0x1000) y 0xFFF es 0x1000 - 0x1. No se puede asignar un bloque de memoria menor a una página. Si una página no es suficiente, se puede llamar a esta función tantas veces como sea necesario con distinta páginas. Notar que no es necesario que las páginas físicas sean continuas, pues solo importa que las direcciones virtuales sean contiguas. Esta función, entonces, recibe los siguientes parámetros:

- Dirección física de comienzo de la página (múltiplo de 0x1000).
- Dirección virtual con la que se desea hacer la asignación.
- Dirección virtual del directorio de páginas en donde se desea hacer la asignación.

Pasos que realiza la función para hacer la asignación:

- Tomar los 10 bits más significativos de la dirección virtual que corresponden al número de tabla en el directorio de tablas.
- Tomar los 10 bits siguientes de la dirección virtual que correspondenal número de página en la tabla de páginas.

- Los 12 bits menos significativos corresponen al desplazamiento dentro de la página, que se usan en el mecanismo de paginación para poder acceder a direcciones particulares dentro de una página. En este caso los desestimamos porque siempre se asigna la página en bloque (solo necesitamos la dirección de comienzo).
- Si la entrada en el directorio de páginas correspondiente a la tabla (accedemos con el número de tabla calculado anteriormente) tiene el bit de presente activado, obtenemos la dirección de la tabla de páginas.
- Si no está presente, establecemos como dirección de tabla una dirección de página libre que le pedimos al manejador de memoria y activamos el bit de presente. Las 1024 entradas de esta nueva tabla las inicializamos vacías (en cero).
- En cada entrada, de directorio o de tabla, seteamos los atributos *read write* y *user*. Si queremos escribir en página seteamos ambos *read write* en 1 y si queremos acceder como usuarios seteamos ambos *user* en 1.
- Ahora que tenemos la dirección de la tabla, accedemos a la entrada correspondiente a la página utilizando el número de página calculado al comienzo y pisamos los siguientes valores: activamos bit de presente y ponemos como dirección base la dirección física de la página que recibimos como parámetro.
- Finalmente llamamos a una función para invalidar la cache de traducción de direcciones.

2.4.3. Eliminación de páginas

La función mmu_desmapear_pagina se encarga de hacer que una página deje de pertenecer a un directorio determinado. Para hacer esto, simplemente separamos la dirección virtual en número de tabla y número de página como en la asignación, accedemos a la tabla desde el directorio y desactivamos el bit de presente de la página correspondiente desde la tabla. Adicionalmente, si detectamos que la tabla queda vacía (todas las entradas con el bit de presente desactivado), desactivamos el bit de presente de la tabla desde el directorio. Por último, llamamos a una función para invalidar la cache de traducción de direcciones.

2.4.4. Directorios de las tareas pirata

La función mmu_inicializar_dir_pirata se encarga de crear el directorio para una tarea pirata. Recibe como parámetros:

- Dirección física de la página donde se encuentra el código la tarea (se asume que entra en una sola página).
- Dirección física de destino del pirata (debería ser parte del mapa).
- ID del pirata.

Realiza los siguientes pasos:

- Pedir al manejador de memoria la dirección de una página libre que utilizaremos para el directorio.
- Asignar las páginas correspondientes a la zona del kernel en el nuevo directorio. Esto es para poder atender interrupciones.
- Asignar las páginas en posiciones matricialmente contiguas a la dirección de destino del código en el directorio actual pero usando una tabla de páginas compartida, lo cual hace que estas páginas sean visibles para todas las tareas automáticamente.

- Copiar el código del pirata de la dirección física de origen a la dirección física destino. Para hacer el copiado del código es necesario asignar una dirección virtual libre cualquiera que esté en la zona de kernel a la dirección física de destino del pirata y usar esta dirección temporal para copiar. Esto es necesario porque para hacer el copiado no usamos el nuevo directorio, sino el directorio actual, que puede pertenecer a otro pirata y si hacemos el copiado usando directamente la dirección virtual 0x400000 estaríamos perdiendo la asignación de ese posible pirata.
- Luego de hacer el copiado, eliminar esta página temporal del directorio actual y asignar la dirección 0x400000 en el nuevo directorio.
- Devolver la dirección física del directorio de páginas de la tarea.

2.5. Ejercicio 5

2.5.1. IDT

Inicializamos tres nuevas entradas en la *idt*: la 32, 33 y 70. Las dos primeras con campo *dpl* en $0x0$, indicando que pueden ser accedidas por el kernel solamente, y la última con *dpl* en $0x3$, para que pueda ser accedida por los usuarios.
Debemos remapear las direcciones de las interrupciones 32 y 33 porque caen en índices de excepciones del procesador. Para esto llamamos a función *resetear pic*. Luego habilitamos las interrupciones externas llamando a la función *habilitar pic* que setea flag en registro de control.
Para las rutinas de atención 32, 33 y 70 debemos resguardar los registros de uso general de manera que las interrupciones sean transparentes a la tarea interrumpida. Luego, excepto en la rutina 70, se debe comunicar al *PIC* que la interrupción será atendida. Esto se hace llamando a la función $fin\ intr\ pic1$. Al salir de la rutina se debe restaurar los registros y ejecutar la instrucción *iret*.
La interrupción del reloj solo debe invocar una función, en este caso *game tick*. En cambio la rutina del teclado debe leer del puerto $0x60$ la información de la tecla. Esto se hace con la instrucción *in* en un registro, por ejemplo $in\ ax$, $0x60$.

2.6. Ejercicio 6

2.6.1. TSS

Para intercambiar tareas necesitamos una tarea inicial, en realidad sólo su estructura de *tss*, para que al switchear se guarde el contexto actual y se pueda cargar el nuevo contexto, registros generales, selectores de segmento, registros de control y flags. Entonces inicializamos la *tss* inicial generando un descriptor de *tss* de la siguiente forma:

- campo *limite* en $0x67$ (mínimamente).
- campo *base* (separado en una *word* y dos *bytes*) con dirección de estructura de *tss* inicial.
- campo *tipo* en $0x9$.
- campo *bit* system en $0x0$ (es descriptor de sistema).
- campo *dpl* en $0x0$ (nivel 0 para que tareas de nivel 3 no puedan saltar a esta).
- campo *presente* en $0x1$ (descriptor presente).
- campos *avl*, *db*, *l* y *g* en 0x0

Luego se llena otro descriptor con los mismos valores excepto la base, que apuntará hacia la *tss* de la tarea *idle*.
La *tss* de la tarea inicial no se llena ya que no se volverá a usar pero sí debemos llenar la *tss* de la *idle*. Esto es:

- campo $cr3$ con actual $cr3$ (el de kernel).
- campo *eip* en $0x16000$ (dirección de inicio del código de la *idle*).
- campo *esp* en $0x27000$ (misma que del kernel).
- campo *ebp* en $0x27000$ (misma que del kernel).
- campos es, ds ,gs, ss en $0x48$ (segmento de datos de nivel cero).
- campo cs en $0x40$ (segmento de código de nivel cero).
- campo fs en $0x60$ (segmento de video nivel 0).
- campo $esp0$ en $0x27000$
- campo $ss0$ en $0x48$ (segmento de datos de nivel cero).

Además mapeamos con *identity mapping* las direcciones $0x16000$, del código de la *idle* y la dirección $0x27000$ para la pila.
Paso seguido llenamos las *tss* de las tareas, ubicadas en array *tss jugadorA* y *tss jugadorA* de tamaño 8 cada uno. Estas se llenan de la siguiente manera:

- campo $cr3$ con dirección de directorio de tarea. Se llama a función *mmu inicializar dir pirata* con dirección de origen de código de tarea (entre $0x10000$ y $0x13000$) y posición de destino en el mapa (puerto de partida). Obtenemos una dirección que será del nuevo directorio.
- campo *eip* en $0x40000$ (dirección de inicio del código de la tarea).
- campo *esp* en $0x400ff4$ (fin de página de código dejando lugar para 3 parámetros).
- campo *ebp* en $0x400ff4$.
- campos es, ds ,fs ,gs, ss, en $0x005b$ (segmento de datos de nivel tresf).
- campo cs en $0x0053$ (segmento de código de nivel tres).
- campo $esp0$ en fin de pila 0 (se pide nueva página y se le suma el tamaño de una página a esa dirección).
- campo $ss0$ en $0x0048$ (segmento de datos de nivel cero).

Finalmente para los descriptores estas *tss* las llenamos igual que lo hicimos con la *idle* pero pasándole la dirección de su respectiva *tss*.

Para saltar a la tarea *idle* primero cargamos selector de descriptor *tss* inicial con instrucción *ltr*. Luego ejecutamos un *jump* a selector de descriptor *tss idle*. Entonces la próxima instrucción sera de código de la *idle*.

2.7. Ejercicio 7: Estructuras Y Lógica De Juego

2.7.1. Estructuras

Para armar la lógica del juego tenemos entre varias estructuras la de *pirata*, *jugador*, *sched task* y *sched*.

- La estructura de *jugador* tiene los siguientes campos: *indice* que indica si es jugador A o B, array *piratas* de tamaño ocho que contiene punteros a los piratas del jugador, dos arrays *vistas x* y *vistas y* que contienen posiciones vistas por pirata de jugador en cada turno. Luego sigue campo *puntos*, para guardar los puntos del jugador y campos *puerto x* y *puerto y*, con posiciones iniciales en el mapa de los piratas del jugador.

- La estructura de *pirata* tiene los siguientes campos: *id* que indica en que índice de la *gdt* está el descriptor de su *tss* (no es selector); *index*, con valores entre 0 y 7, que sirve para indizar en el array de piratas de jugador dueño; *jugador* que es puntero a pirata dueño; *libre* que es flag que indica si el slot en array *piratas* de jugador asociado al pirata está libre; x e y que indican la posición del pirata en el mapa; *tipo* que indica el tipo de pirata (minero o explorador) y *reloj* indicando reloj del pirata.

- La estructura *sched* tiene los siguientes campos: *tasks* que es array de estructuras *sched task t* y tiene tamaño 17, 16 tareas de jugadores más tarea idle; *current* indicando índice de tarea actual en *tasks*; campos *posiciones tesoros A* y *posiciones tesoros B* que son colas $fifo$ (first in first out) de tuplas x e y y almacenan posiciones con monedas descubiertas por cada jugador; *inicio tesoros A* y *inicio tesoros B* que indican primer lugar ocupado de las colas de posiciones anteriores (al consultar posiciones de tesoros se chequean estos índices y se sacan usando su valor; luego se incrementan en uno apuntando al siguiente elemento de la cola y en caso de llegar al final de la cola apuntará al principio); campo *prox* que indica índice en *tasks* de siguiente tarea a la actual y por último *tiempo sin cambios* indicando el tiempo de juego en que ningún jugador anotó puntos (sirve para finalizar juego).

- La estructura de *sched task* tiene los siguientes campos: *gdt index* que indica selector de *tss* asociado a tarea en ese slot de array *tasks*, *pirata* con puntero a tarea del *sched task*, flags *reservado minero* y *reservado explorador* tal que al llegar scheduler a este *sched task* indican si se debe activar la tarea y que tipo darle, *posiciOn x tesoro* y *posiciOn y tesoro* que en caso de que se deba activar minero indican a donde buscar las monedas.

2.7.2. Lógica del juego

Al iniciar juego se inicializan los jugadores asignándoles índice (A o B), se setean puntos en 0, inicializamos el mapa de cada jugador, pedimos las páginas libres para mapear las posiciones descubiertas y ser accedidas por todos los piratas de cada jugador (las usan los mineros para moverse). Luego inicializamos los piratas.
Al iniciar piratas se asignan a los piratas de cada jugador (8 para cada jugador) un número *id* que no cambia en el resto del juego. El *id* asocia a cada pirata con el índice del descriptor de su *tss* en la *gdt* (los *id* de los piratas de jugador A están en rango desde 15 a 22 y los de jugador B en rango desde 23 a 31). Entonces el scheduler puede ubicar el descriptor de *tss* del pirata a través de su *id* y usarlo para cambiar de tarea (shifteando en tres a izquierda al *id* y sumando atributos obtenemos selector en *gdt*). Es decir que inicialmente cada pirata de jugador tiene descriptor de *tss* inicializada (asociada a una *tss* vacía). Luego a los piratas se les asigna además del *id* un índice, entre 0 y 7, para iterar en array de piratas de cada jugador. También se les asigna un puntero al jugador dueño y se les setea el campo *libre* en 1 (*true*) y campo *reloj* en 0.

Luego se inicializa el scheduler, se asigna a cada posición de array *tasks* (17 en total) un *sched task* asociado a tarea, fijando en posiciones dadas por índice 1 a 8 tareas de A y posiciones en índice 9 a 16 tareas de B; en índice 0 va la tarea *idle*. Se setean atributos de cada *sched task* seteando índice en gdt de su tarea, puntero a pirata de jugador, los flags *reservado explorador* y *reservado minero* se setean en falso y parámetros de posiciones de tesoros se ponen en 100 (vacías). Como en posición 0 de array *tasks* va la *idle* al inicio el parámetro *current* de scheduler se setea en 0 (es la única activa). Luego se deben inicializar los arrays *posiciones tesoros* con tuplas de x e y en 100 indicando posiciones vacías (con un máximo de 20 posiciones disponibles). También seteamos *prox* en 500, indicando que la siguiente no es tarea de jugador.

Luego de inicializar todo la única tarea activa es la *idle*. Si queremos lanzar piratas debemos usar las interrupciones.

2.7.3. Interrupción de teclado

Al pulsar tecla de lanzar pirata ($shiftL$ o $shiftR$) se chequea en piratas de jugador si hay slot libre tal que en ese caso se debe setear en su *sched task* el campo *reservado explorador* como *true* y llenamos los campos del pirata: *libre* en falso, seteamos su posición con valores de puerto de jugador y le asignamos tipo explorador. No inicializamos su tss porque si activaramos tarea en interrupción de reloj se puede pisar código en switch de tarea (puede suceder que siguiente tarea salga del mismo puerto y no sea explorador. Luego en esa posición física del mapa se pisará el código del explorador). Sin embargo si hay algún pirata en el puerto y hay slot libre se lanzará nuevo pirata pisandose el código del primero.
Se debe chequear campo *prox* de scheduler tal que si la tarea dada por *prox* pertenece a mismo jugador que la tarea actual, la de *current*, y pirata lanzado es de jugador contrario a estos se debe pisar *prox* con índice de nueva tarea. Caso contrario (nueva tarea pertenece a jugador actual) no se toca *prox*.

En caso de pulsar la tecla *y* se activará el modo debug. Para esto usamos los flags *modo debug* y *pantalla debug activada* que nos indican si el modo está activo o no. En caso de no activo al presionar tecla *y* se setea *modo debug* en *true* y ante cualquier excepción se mostrará una pantalla con información de los registros de la tarea que murió. Estos son: registros generales, que se obtienen directamente en handler de excepción; el *eip*, se obtiene buceando en la pila, los segmentos de código, los eflags y registros de control. Todos se almacenan en estructura auxiliar llamada *debugger* tal que al tener que imprimirse por pantalla estos valores los sacamos de ahí.
Antes de mostrar la información por pantalla se debe resguardar pantalla actual de juego. Para esto usamos una matriz auxiliar de tamaño de la pantalla tal que al salir de modo debug de allí obtenemos los datos para pintar la pantalla que se mostraba justo antes de la excepción. Este modo estará activo hasta que se presione la *y* nuevamente en el juego. Durante tiempo en que está activo no se permite mover ni lanzar pirata.

2.7.4. Interrupción de reloj

En cada interrupción de reloj se consulta si el debugger está activo y si es así no se debe cambiar de tarea. Si debug no está activo llamamos *sched atender tick*. Esta función hace lo siguiente: actualiza el reloj global, el reloj del pirata actual y en caso de que *scheduler tiempo sin cambios* sea igual a 10000 se dará por terminado el juego.
Luego se chequea si hay slot de algún jugador seteado como libre (se chequean para cada jugador). Si hay slot libre buscamos tarea en espera, es decir, se debe chequear en array *posiciones tesoros* de scheduler asociado a jugador dueño de slot libre tal que si encuentra tupla con valores distinto de 100 (moneda descubierta) entonces en estructura *sched task* del pirata libre se debe setear campo *minero* en *true* y copiar valores de tupla, *x* e *y*, a *posicion x tesoro* y *posicion y tesoro* de *sched task*. Luego se restauran esos valores de *posiciones tesoros* con 100 (indicando posición vacía) para próximas posiciones a guardar. Esto se debe repetir por cada slot vacío que se encuentre, es decir chequear si hay minero en espera por cada slot vacío para cada jugador.

Si campo *prox* de scheduler está apuntando a tarea activa entonces actualizamos *current* con ese *prox* para switchear allí.

Luego buscamos siguiente tarea perteneciente al jugador reciente y actualizamos parámetro *prox* de scheduler tal que en próximo switch de tareas se use esta tarea. Si no hay debe actualizarse *prox* con siguiente de jugador contrario a jugador reciente. En caso de no haber jugadores activos *prox* no se actualiza.

Si la tarea a switchear (dada por *current* actualizado) estuviera reservada se debe activar pirata, inicializar su *tss* (incluye mapear su posición en mapa) y pintar. Luego se debe setear campo

de reservado en falso.
Si se reservo para minero se debe guardar en su pila de nivel 3 los valores de campo *posicion x tesoro*, *posicion y tesoro* (posiciones donde cavar) y lanzar al minero. Esto se hace mapeando una página temporal apuntando a la pila y luego de copiar los datos la desmapeamos.
Si estuviera reservado para explorador limpiamos el flag de reservado y lanzamos la tarea. Devolvemos el selector de tarea siguiente (dado por *current* recientemente actualizado).

Devuelta en handler de interrupción de reloj, con selector de siguiente tarea en ax, cargamos el selector actual con instrucción *str* en registro cx y comparamos ax con cx. Si son distintos hacemos un jump far a la siguiente tarea. Si son iguales se debe salir de la interrupción sin switchear.

2.7.5. Interrupción de software $0x46$

Se chequea que caso de syscall se requiere.

- Al mover:
  Se chequea que la posición nueva sea válida (dentro del mapa). Si no es así se mata la tarea. Si la nueva posición es válida se mueve su código a nueva posición en mapa. Para esto se mapea una página no usada por los jugadores a la anterior posición del pirata, luego se usa esta página para acceder al código de la tarea y se lo copia a la nueva posición, que ya está mapeada (hecho al inicializar *tss*). Finalmente se desmapea esta página temporal.
  Luego si tipo de pirata es explorador chequeamos si en las posiciones descubiertas válidas (3 celdas en dirección a mover) hay monedas. En caso de haber descubierto tesoro se busca tarea libre de jugador actual y en estructura *sched task* asociada al pirata se reserva como minero, guardando la posición del tesoro en campos *posicion x tesoro*, el x, *posicion y tesoro*, el y, para que el scheduler cuando llegue a ese slot lo active y lo mande a esos lugares.
  Si no hay slot disponible para lanzar minero entonces guardar posición de tesoro descubierta para cuando alguno se libere. Esto se almacena en array *posiciones tesoros* de *sched task* correspondiente a jugador llamador. El scheduler ante primer interrupción se encargará de chequear si hay lugar y entonces lanzar minero. Si no hay posiciones libres en *posiciones tesoros* las monedas descubiertas se descartan.
  Si tipo de pirata es minero se chequea que su nueva posición esté mapeada. Esto se hace comprobando que la nueva posición esté presente como dirección virtual, es decir que esté mapeada como *read only* entre las páginas mapeadas por los exploradores de un jugador. Si no es así se mata al pirata.

- Al cavar:
  Se chequea que no se invoque con parámetros inválidos (campo *id*, tipo de tarea). Luego consultamos si hay monedas en la posición actual y cuántas hay. Si hay cero entonces se debe matar la tarea. Si hay una o más monedas se incrementa puntaje de jugador dueño de minero en uno y se decrementa cantidad de monedas en esa posición.

- Consultar posición:
  Chequeamos que los parámetros sean válidos (*id* de pirata válido y número de caso en rango dado por -1 y 7). Luego retornamos la posición requerida codificada en un *uint*32. Como tenemos que devolver la posición al llamador y la devolvemos por registro eax, a la hora de desarmar el stack pointer, cuando debemos restaurar el valor de eax ignoramos el valor pusheado de eax incrementando el *esp* en 4.

En todos los casos al final del procedimiento se debe saltar a la tarea *idle*.

2.7.6. Excepciones (0 a 19)

Se mata a la tarea que produjo la excepción. Acá vienen a parar las tareas que realicen movimiento inválido, llamen a syscall con parámetros incorrectos y tiren las demás excepciones del procesador.

Cuando la tarea muere se chequea si *prox* de scheduler apunta a esta (tarea única). En ese caso se debe setear *prox* en default (500, ninguna tarea activa) y no se debe modificar campo *current* (por más que el pirata actual muera). También seteamos campo *libre* del pirata con 1 y si el modo debug está activo mostramos la información del pirata por pantalla, esto es, registros de uso general, $eflags$, registros de control y segmentos de código.
Antes de salir del handler habilitamos las interrupciones, con instrucción *sti*, y luego saltamos a la tarea *idle*. En este caso no conservamos los registros de la tarea porque se la desaloja.