DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

System Programming - Infectados

Organización del Computador II
Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Borgna, Agustín	079/15	aborgna@dc.uba.ar
Corleto, Alan	790/14	corletoalan@gmail.com
Lancioni, Franco	234/15	glancioni@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Ejercicio 1

1.1. Inicializar la GDT

Configuramos la GDT para tener bien configurada la segmentación al pasar a modo protegido.

Dejamos la primer entrada de la tabla vacía, por convención de la arquitectura, y las tres siguientes, que quedan reservadas para la práctica.

Además, creamos otras cinco entradas para nuestro uso:

- Un segmento flat (con base 0 y límite de toda la memoria) de código con nivel de privilegio 0
- Un segmento flat de código con nivel de privilegio 3
- Un segmento flat de datos con nivel de privilegio 0
- Un segmento flat de datos con nivel de privilegio 3
- Un segmento de datos con nivel de privilegio 0, con base en $0xB8000$ y límite $80 * 50 * 2 - 1 = 7999 = 0x1F3F$

1.2. Pasar a modo protegido

Para pasar al modo protegido debemos habilitar primero A20 con la rutina provista por la cátedra, luego cargar la GDT con la instrucción $LGDT$, habilitar el bit PE del registro CR0 que setea el modo protegido propiamente dicho, y saltar a una sección del código programada en 32 bits.

```
; Habilitar A20
call habilitar_A20

; Cargar la GDT
lgdt [GDT_DESC]

; Setear el bit PE del registro CR0
mov eax, cr0
or eax, 1
mov cr0, eax

; Saltar a modo protegido
jmp GDT_CODE_0_DESC:mp
```

1.3. Imprimir el mapa inicial

La memoria de video que mapeamos anteriormente representa un mapa de caracteres de 80 columnas y 50 líneas, con cada caracter representado como un byte con su valor ASCII y un byte con sus atributos (color de letra, color de fondo y un flag de blink).

Para inicializar el mapa recorremos el array y escribimos los valores que queremos para obtener la figura 1

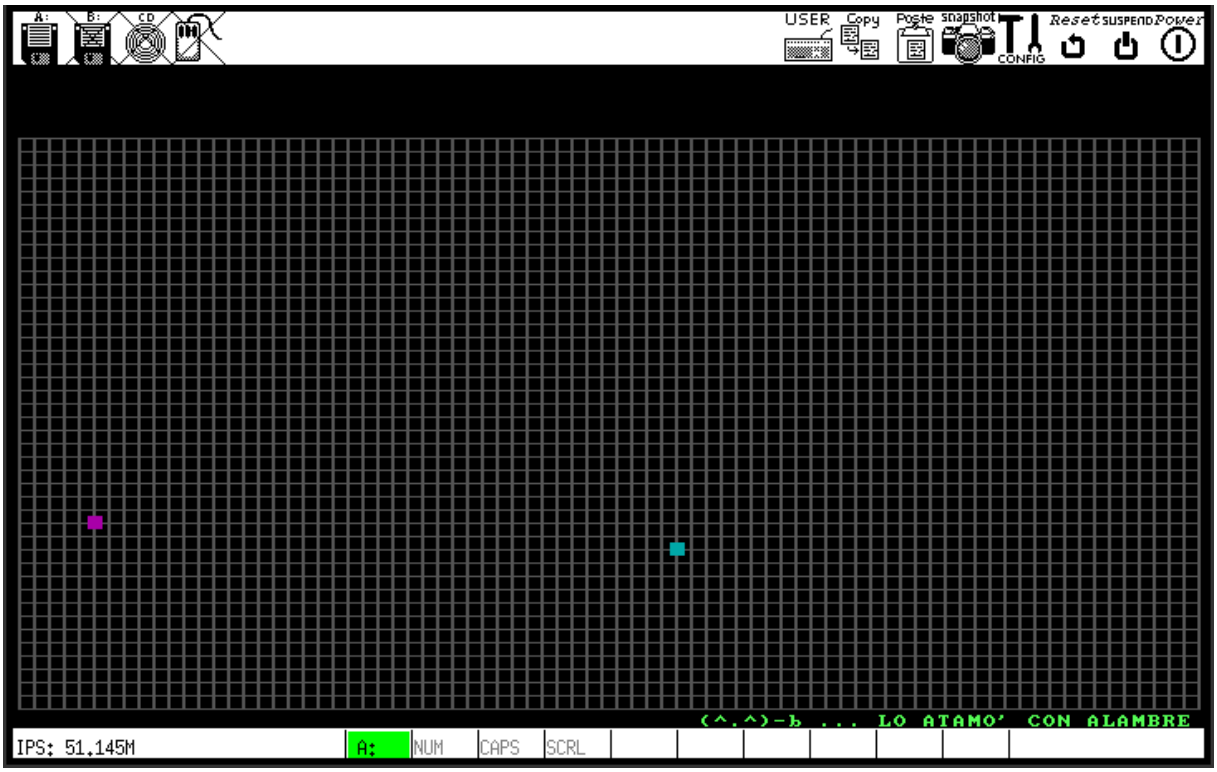

Figura 1: Mapa inicial

2. Ejercicio 2

La IDT se corresponde con un arreglo de tamaño 255 de idt_entry. Un idt_entry es un struct que representará cada gate dentro de la tabla indicando selector de segmento de código donde se almacena la ISR, offset en dicho segmento y atributos del descriptor (present gate, tipo de gate y dpl).

```
typedef struct str_idt_entry_fld {
    unsigned short offset_0_15;
    unsigned short segsel;
    unsigned short attr;
    unsigned short offset_16_31;
} __attribute__((__packed__, aligned (8))) idt_entry;
```

Listing 1: Struct de las gates

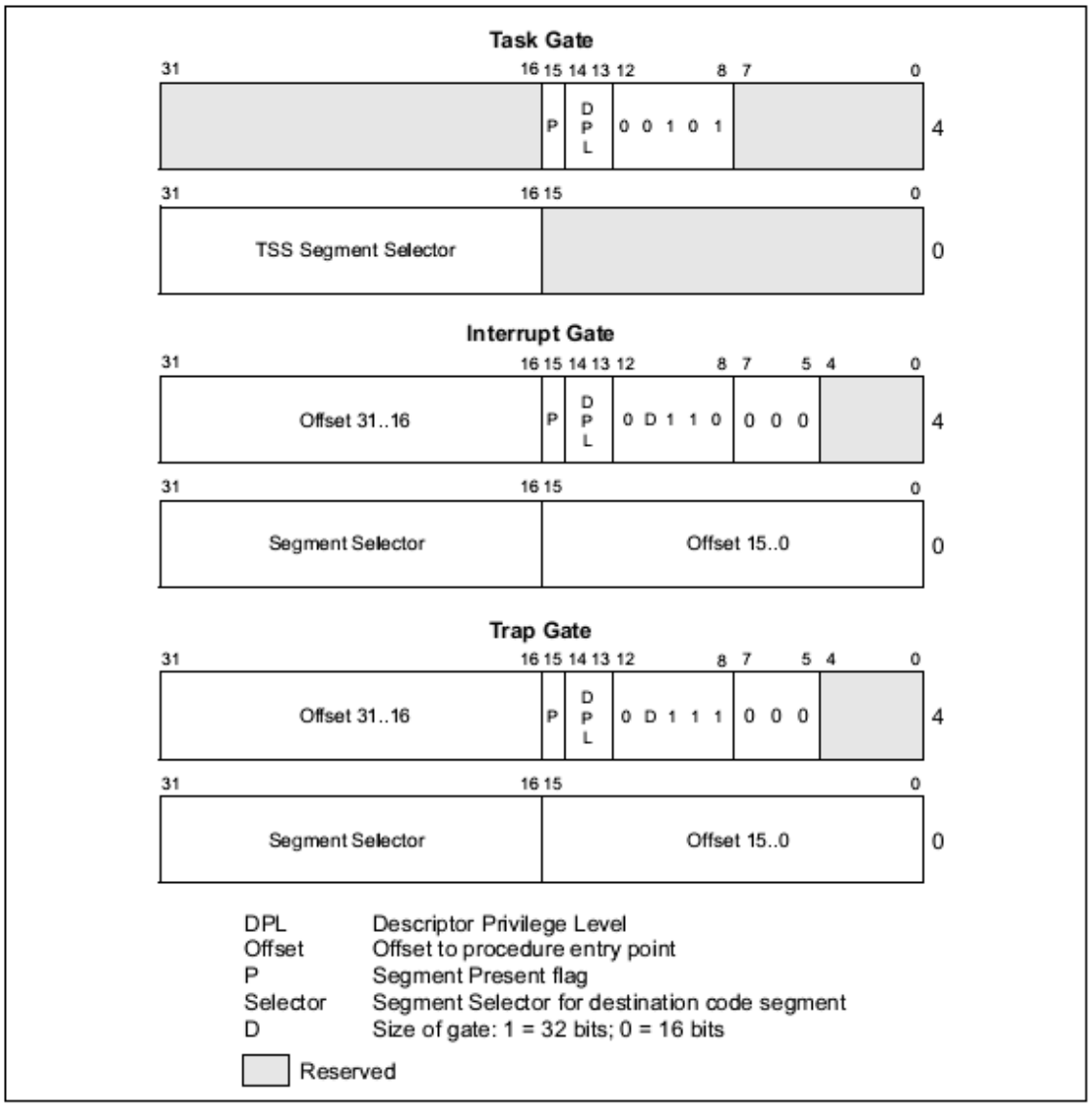

Figure 6-2. IDT Gate Descriptors

Figura 2: Formato de las gates en IDT

2.1. Inicializar la IDT

Para inicializar la IDT usamos una serie de macros tanto en C (para las gates) como en ASM (para las ISR) para facilitar la definición de las mismas. De los macros generados usamos particularmente 3 de estos.

IDT_ENTRY_INTERRUPT para las interrupciones de tipo 0-19, 32,33 y 40. Se setea con el descriptor de nuestro segmento flat de código de nivel 0, y offset (dividido de 0 a 15 y 16 a 31 bits por cuestiones de arquitectura) con puntero a una rutina genérica del 0-19, y específica para reloj (PIT y RTC) y teclado en interrupciones 32, 40 y 33 respectivamente.
Los atributos corresponden a nivel de privilegio 0, bit de present en 1 y tipo = 0xE >> 2

IDT_ENTRY_DEFAULT para las interrupciones reservadas por Intel según manual (20-31) y aquellas ”libres”para el usuario que no definimos (por lo tanto, no deberían ser accedidas durante la ejecución del sistema).
De esta forma podemos enterarnos, agregando un breakpoint en la rutina, si salta una interrupción que no estamos atendiendo.

IDT_ENTRY_INTERRUPT_USER para syscalls int 0x66 con privilegios de nivel usuario para llamar a SOY, DONDE y MAPEAR.

Originalmente las interrupciones no-default se encargan de imprimir por pantalla su número de interrupción con un define que se hace dentro del macro de la isr.

```
%macro ISR 2
global _isr%1

interrupt_msg_%1 db          %2
interrupt_msg_%1_len equ     $ - interrupt_msg_%1
```

Listing 2: Mensaje en el macro de isr

Todo esto quedó encapsulado en una función llamada $idt\_inicializar$ que es ejecutada luego de cargar en IDTR un descriptor con la dirección donde se encuentra la IDT y su límite, empaquetados en el struct $idt\_descriptor$ de la cátedra.

3. Ejercicio 3

3.1. Inicializar las páginas del kernel con identity mapping

Para inicializar el directorio de páginas del kernel creamos una función $mmu\_inicializar\_dir\_kernel$ que hace lo siguiente:

- Setea en cero todos los page directory entry menos el primero, que tiene como base la dirección de su primera tabla y los bit de present y write en 1.
- Mapea todas las entradas dicha página con identity mapping y setea los bits de present y write en 1, logrando mapear así los primeros 4MB de memoria.

Más adelante extendimos la misma función para que mapee la página de la tarea idle, también con identity mapping.

3.2. Activar paginación

Para activar paginación movimos en eax el contenido de CR0, seteamos en 1 el bit de paginación (bit 31) y devolvimos a CR0 el contenido de eax.

4. Ejercicio 4

4.1. Contador de páginas

Se declara en mmu.c una variable global $proxima\_pagina\_libre$ de tipo void* (que apunta a una dirección de memoria) y se inicializa apuntando a la dirección 0x100000 (el inicio del área libre, mapeada por identity mapping). Cuando pidamos memoria en el área libre con $mmu\_proxima\_pagina\_fisica\_libre$ se retornará esta dirección para uso libre de kernel (a modo pseudo-malloc, aunque no es posible recuperar el espacio pedido) y se incrementará en 2^{12} (es decir, apuntará a la próxima página en memoria).

4.2. Inicialización de directorios y tablas de paginación para tareas

La función $mmu\_inicializar\_dir\_tarea$ recibe el puntero de la tarea que se quiere mapear, la posición (x,y) del mapa donde se va a inicializar (el parámetro se pasa en un struct pos_t) y el directorio actual.
Comienza creando (en blanco) el page directory de la tarea que se pasó por parámetro, que se escribe en una página pedida a $mmu\_proxima\_pagina\_fisica\_libre$.
Se mapean para el kernel las páginas de la tarea y de su posición del mapa por identity mapping.

Se copia el código de la tarea a la posición del mapa pedida y se le mapea, con permisos de usuario, la celda para la tarea.
Por último se retorna el directorio, ya inicializado, de páginas de la tarea.

4.3. Mapeo y desmapeo de páginas de memoria

La función mmu_mapear_pagina recibe la dirección virtual que se desea mapear y la física a la cual se quiere mapear, junto con el directorio actual de páginas (cr3) y los atributos que queremos asignar al entry en la page table.
En caso de que el bit de present que corresponde al índice indicado por la dirección virtual en el directorio se encontrara en 0, es necesario generar un page directory para indexar esa misma dirección con permiso de escritura y usuario (esto se debe a que si el entry de directorio tiene permisos de usario, el dpl resultante es el que indique la tabla). Luego se accede a la tabla y se agrega el entry correspondiente, marcándolo como presente. Finalmente se limpia la hidden caché del directorio con tlbflush() dado que se realizaron cambios en él.

5. Ejercicio 5

Como ya definimos anteriormente (2.1), si bien seteamos rutinas 'default' para excepciones, las rutinas de atención de interrupciones de teclado y reloj son diferentes. Para estas interrupciones (controladas por el PIC) interactuamos con puertos I/O para luego hacer llamados a rutinas específicas (rtc_isr, $keyboard\_isr$, etc) que respondan a estas interrupciones.
Por lo tanto lo único que haremos dentro de las $\_isr(\#int)$ definidas en $isr.asm$ será comunicar el fin de interrupción a los puertos del pic y hacer llamados a tareas correspondientes codeadas en C.
Caso aparte es el de la int 0x66, que es la que en la lógica del juego permitirá hacer las syscalls para llamar a $DONDE$, SOY y $MAPEAR$ según un parámetro input en eax.

5.1. Habilitar interrupciones de reloj

Contamos con dos rutinas de atención a dos relojes diferentes: el RTC [1] y el PIT [2], asignados a interrupciones nº 32 y 40 respectivamente.
Si bien en el contexto del uso dado en este sistema no presentan diferencias ambos relojes, delegamos en el RTC las cuestiones referentes al scheduler y dedicamos el PIT exclusivamente para el funcionamiento del audio A.1.

Para el RTC se corresponde la siguiente rutina:

```
void rtc_isr() {
    // Read RTC_C to ack the interruption
    outb(RTC_CMD, RTC_MASK_NMI | RTC_C);
    inb(RTC_DATA);

    game_tick();
}
```

Donde la primer línea se encarga de reconocer la interrupción y se deshabilitan las NMI [3] mientras se accede el registro C del RTC (de no suceder esto, frente a determinadas interrupciones, se podría deshabilitar la interrupción del RTC). El acknowledge se completa leyendo el puerto de datos del RTC en la segunda línea.
Luego se hace un llamado a $game\_tick()$ (función de $game.c$) que llama a actualizarse al scheduler.

[1] Real-time clock
[2] Programmable Interval Timer
[3] Non-maskable interrupt, `http://wiki.osdev.org/RTC#Avoiding_NMI_and_Other_Interrupts_While_Programming`

5.2. Habilitar interrupciones de teclado

Se lee del puerto $0x60$ un scan code que luego convertimos a ASCII usando un switch statement para cada botón de la interfaz del juego (ignorando los break codes, de modo que solo se acciona una interrupción al pulsar cada tecla). A partir de esta conversión se determina si se actualiza la posición de un cursor, si se muestra la pantalla de restart, ayuda o se setea el debug mode 5.3 (entre otros).

```
unsigned char status2ASCII(unsigned char input){
    unsigned char output = 0;
    switch (input){
        case 0x01: // Esc
            output = 3;
            break;
        //Player 1
        case 0x11:
            output = 'W';
            break;
        case 0x12:
            output = 'E';
// [...]
}
```

5.3. Debugger

Dentro de la rutina de $keyboard\_isr$, si el debugger (7.4) se encuentra en pantalla entonces solamente se atenderá la interrupción de teclado que lo desactiva (correspondiente a la letra 'Y'). Por lo tanto no se podrán mover cursores cuando se esté mostrando.
Para habilitar el modo debug se llama a la interrupción de teclado de la letra 'Y' y se habilita un flag $dbg\_enabled$ desde la función $game\_enable\_debugger$ antes de volver a la ejecución de la tarea. La próxima vez que se produzca una excepción o un error de parámetros en syscall ($game\_kill\_task$), se detendrá la ejecución y se llamará al debugger hasta que se presione nuevamente la tecla 'Y' y se salte a la tarea idle ($game\_go\_idle$) para retomar la ejecución del juego.

5.4. Habilitar interrupción $0x66$

Se define para la int $0x66$ una syscall, cuyo funcionamiento 'ingame' detallaremos más adelante (7.2). Al entrar en la isr correspondiente se produce un llamado a funciones externas de $game.c$ en función de el parámetro pasado en eax, si este parámetro no es válido se procede a terminar la tarea que hizo la llamada a sistema por prevención de que esté corrupta.

```
global _isr0x66
_isr0x66:
    pushad

    cmp eax, SYSCALL_DONDE
    jne .not_donde
        push ebx
        call game_donde
        add esp, 4
        jmp .end
    .not_donde:
;   [...]

; Syscall invalido
call game_kill_task

.end:
popad
iret
```

6. Ejercicio 6

6.1. Definir entradas en la GDT para las TSS

Para definir las entradas de la GDT para las TSS creamos una función $gdt\_setear\_tss\_entry$ que toma el offset dentro la gdt, un puntero a una TSS, y un valor de Data Privilege Level (de 0 a 3) y completa una entrada nueva en la GDT (determinada por offset) con los datos pertinentes a una tarea.

Usamos esa misma función para completar las entradas de las tareas Inicial y Idle, así como luego utilizamos lo mismo para llenar las entradas de las 25 tareas que pertenecen al juego.

6.2. Completar la entrada TSS de la tarea Idle

Completamos la información de la tarea Idle en su entrada correspondiente con los datos que se requirieron en el enunciado:

- La tarea Idle se encuentra en la dirección 0x00010000
- Su pila se encuentra en la misma dirección que la del kernel y fue mapeada con identity mapping
- La tarea misma ocupa una página de 4KB y se encuentra mapeada con identity mapping
- Su CR3 es el mismo que el del Kernel
- En los EFLAGS habilitamos el flag de interrupciones (ya que de otro modo no podría ser interrumpida para poder cambiar de tarea)

6.3. Inicializar TSS de tareas

Creamos una función llamada $tss\_inicializar\_tarea$ que dado un puntero a una entrada de TSS y un CR3, modifica la entrada pasada por parámetro para que cumpla con los requisitos del trabajo práctico.

- Se pide una nueva página al directorio de páginas del kernel para el stack del mismo perteneciente a la tarea
- Se asigna el tope de esa página como el stack pointer de nivel 0
- Su instruction pointer es asignado en la dirección virtual 0x0x08000000
- Su stack pointer es asignado al final de su primera página virtual
- En los EFLAGS habilitamos el flag de interrupciones por la misma razón que la tarea Idle
- Los selectores de segmento de datos y de código son asignados como corresponde

El CR3 pasado por parámetro viene de haber llamado previamente a la función $mmu\_inicializar\_dir\_tarea$ que toma la dirección del código original de una tarea, una posición de el_mapa y el directorio de páginas actual para así crear un nuevo directorio que corresponderá a la nueva tarea y devolver la dirección del mismo.

Con este mecanismo inicializamos la TSS de todas las tareas involucradas en el juego.

6.4. Saltar a la tarea Idle

Para ejecutar el salto hacia la tarea Idle, primero cargamos la TSS de la tarea Inicial (que puede contener cualquier cosa ya que solo la vamos a usar para saltar) en el Task Register con el siguiente código:

```
mov ax, GDT_TSS_INICIAL_DESC
ltr ax
```

Luego, para saltar a la tarea Idle llamamos a la función tss_switch_task cuyo funcionamiento es detallado más adelante en el desarrollo del ejercicio 7.

7. Ejercicio 7

7.1. El scheduler

El scheduler mantiene una lista de las tareas vivas y, cada vez que se llama a $sched_proxima_tarea$, selecciona el próximo candidato a correr.

El enunciado pedía que se devuelva un índice a la gdt con el descriptor de TSS de la tarea, pero consideramos que sería mas cómodo devolver el tipo y número de la tarea, para que luego la lógica del juego sepa fácilmente quién está corriendo en el momento.

Para decidir el siguiente elemento se realiza un algoritmo Round-Robin para los grupos y para las tareas dentro de cada uno.

7.2. Syscalls

Las tareas pueden ejecutar varios syscalls a través de la interrupción 0x66, pasando su código en eax.
Los tres syscalls disponibles son:

- DONDE (código 0x124) - La tarea pasa como parámetro un puntero a dónde el kernel escribe sus coordenadas en el mapa.
- SOY (código 0xA6A) - La tarea informa su estado actual, pasa como parámetro un código indicando si es Roja, Azul o Neutral.
- MAPEAR (código 0xFF3) - La tarea pide al kernel que le mapee una posición xy en el mapa.

  El kernel se encarga de mapear en la página virtual $0x08001000$ de la tarea la página física solicitada, con permisos de usuario y escritura.

Al finalizar la syscall, la tarea pierde su turno de ejecución.

7.3. Cambio de tareas

Cuando una tarea pierde su turno, el kernel pasa a ejecutar la tarea Idle.

Si durante la ejecución de una tarea se produce una excepción o ella intenta hacer una syscall con parámetros inválidos, el kernel la marca como muerta (le indica al scheduler que ya no la debe ejecutar) y pasa a ejecutar la tarea Idle.

Cuando interrumpe el RTC, el kernel le pide al scheduler cuál es la siguiente tarea a ejecutar y cambia a ella (si es diferente a la actual).

Para hacer estos cambios de tareas llamamos a la función tss_switch_task con el selector del segmento tss a usar. Esta función realiza un $jmpfar$ a una dirección con en el segmento especificado, lo que activa el mecanismo de cambio de tarea del procesador, que corre el siguiente código.

```
offset: dd 0
selector: dw 0

; void tss_switch_task(short descriptor)
tss_switch_task:
    movzx eax, word [esp+4]
    mov [selector], ax
    jmp far [offset]
    ret
```

7.4. Debugger

El debugger es una herramienta a la hora de programar las tareas, que nos permite ver el estado de los registros cuando una tarea se muere.

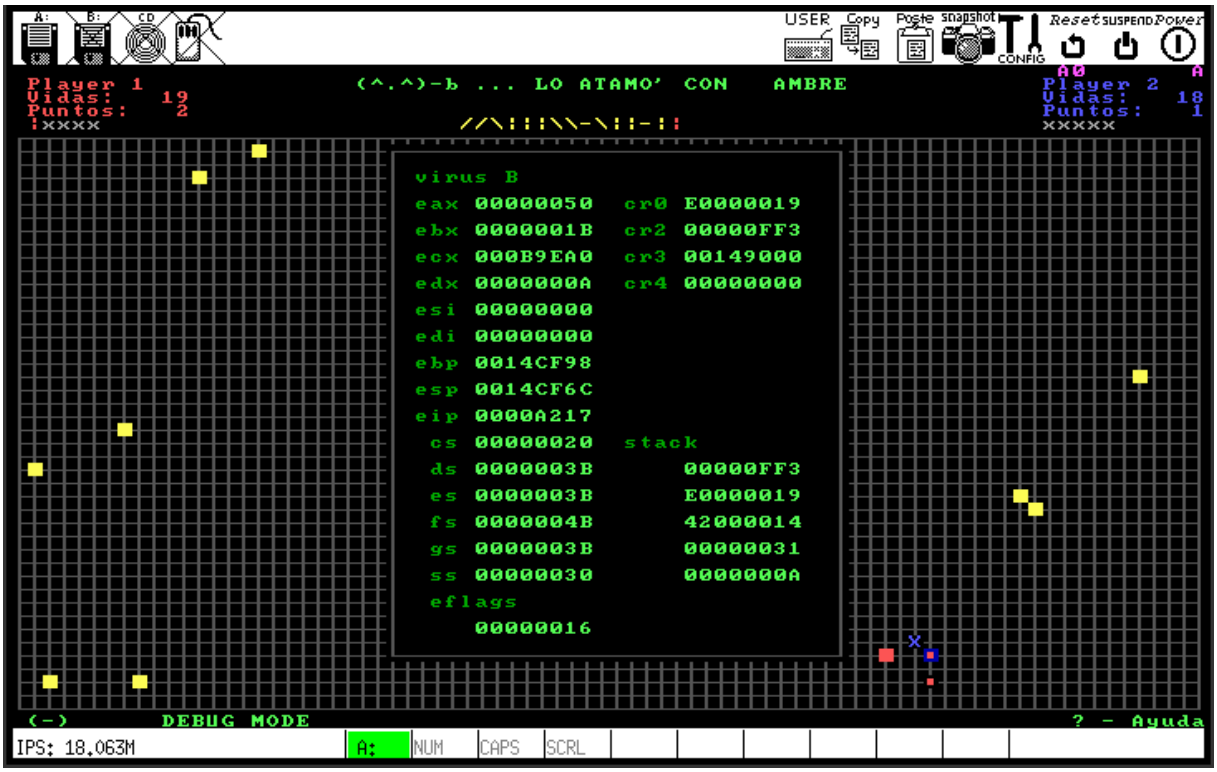

Figura 3: Cuadro de información del debugger

Cuando se mata a una tarea, si está habilitado el modo debug, se llama a $screen\_show\_debug$. Esta función recopila el estado actual de los registros y los imprime en pantalla, junto a el tope del stack.

A. Apéndice - Audio

La mayoría de las computadoras poseen un speaker conectado como periférico al procesador que permite generar sonidos de alerta.

Este speaker está directamente conectado al puerto 2 del Programmable Interval Timer (PIT). Usando el modo generador de ondas cuadradas del PIT podemos entonces generar una señal de salida audible. Ya que esta señal es siempre un a onda cuadrada con duty 50 % la calidad del sonido es bastante mala, pero pudimos usarla para reproducir pistas de audio de dos canales.

Bochs nos permite emular el speaker habilitando el plugin de la soundblaster16, si se agrega el flag $--enable-sb16$ al compilar.

A.1. Configuración de los periféricos

Habilitamos la conexión del PIT con el speaker seteando los bits 1 y 2 del puerto $0x61$ de IO.

```
tmp = inb(0x61);
if (tmp != (tmp | 3)) {
    outb(0x61, tmp | 3);
}
```

Y para configurar el PIT lo seteamos en modo 3 (square wave) y le enviamos un valor de contador. Este valor lo usará para dividir su frequencia base de 1.19MHZ, para obtener como resultado una frecuencia audible.

```
// Configure the pit, mode 3 (square wave)
outb(0x43, 0x36 | (channel << 6));

// Send the 16b count to the corresponding port
```

```
outb(0x40 + channel, (uint8_t) (divisor & 0xFF));
outb(0x40 + channel, (uint8_t) (divisor >> 8));
```

A.2. Reproducción de MIDIs

Diseñamos un formato compacto de archivo de audio compuesto por secuencias de bloques de dos bytes. El primer byte es el valor de una nota MIDI, un entero entre 1 y 127 (o 0 si es silencio) que se traduce a un valor de frecuencia con la siguiente fórmula:

$$freq = 440 * 2^{\frac{nota-69}{12}} Hz$$

Y el segundo byte es la duración en ms.

Desarrollamos un conversor de archivos MIDI a nuestro formato (src/tool/frommidi.py), que genera archivos binarios $.audio$, y usando la pseudoinstruccion $incbin$ de NASM los incluimos en el código del kernel.

Quisimos usar este sistema para hacer sonar efectos de sonidos con el movimiento de los jugadores y la acciones de las tareas, Pero nos encontramos con dos problemas:

- En la emulación de bochs, activar el canal entre el PIT y el speaker incurre un delay aparentemente aleatorio de entre 0 y 1s
- Si se deja activa la conexión pero se deshabilita el PIT, se produce un ruido audible en el speaker

Debido a estos problemas, abandonamos la idea de agregar efecto de sonido.