DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

Organización del Computador II

Integrante	LU	Correo electrónico
Daniel Claverino	273/10	dclave@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 y base teórica

Este trabajo tiene como objetivo inicial mostrar la diferencia de performance en tiempo real entre dos implementaciones del Juego de Conway[1], utilizando SSE y registros de propósito general (PG).

Para esto, se tomó como base el microkernel desarrollado como TP3 en la materia. Se modificó la MMU para que soportara memoria dinámica con *malloc()* y *free()*, reciclando todos los recursos utilizados cuando se termina un proceso, de forma que no haya pérdida de memoria. A su vez, el scheduler fue reescrito para almacenar los procesos en una lista enlazada, también reciclando los recursos usados cuando se decide terminar alguno.

Se agregaron algunos drivers. El modo gráfico se setea usando VESA, lo que permite acceder a resoluciones que VGA no soporta. También hay un driver encargado de atender la interrupción del teclado y bufferear los códigos, y un par de drivers para acceder en modo lectura a un disco virtual formateado con FAT16.

Finalmente, la tarea `IDLE` implementa una consola que permite navegar el disco virtual y ejecutar una tarea, o dos en simultáneo, en cuyo caso les cede el control hasta que mueren por tirar una excepción o son terminadas manualmente.

Los detalles de implementación aparecerán en las secciones correspondientes. Mucha de la información teórica e ideas de implementación fue sacada de OSDev.org[2].

2. Organización

Los archivos del TP están organizados de la siguiente forma:

- **./src**: el TP propiamente dicho, con sus fuentes y el Makefile.
  - **./common**: Archivos compartidos por las tareas y el sistema.
    - **./utils.***: Funciones útiles tanto para el Kernel como para las tareas, como escribir texto formateado en pantalla, funciones de strings, etc.
    - **./types.h**: Definición de algunos tipos de datos
    - **./system/utils.asm**: Implementación de las llamadas al sistema para el Kernel.
    - **./tasks/utils.asm**: Implementación de las llamadas al sistema para las tareas.
  - **./drivers**: Los drivers implementados.
    - **./fat16_driver.***: Driver del FileSystem.
    - **./system/graphic***: Driver gráfico.
    - **./tasks/hdd_driver.***: Driver de disco.
    - **./tasks/keyboard.***: Driver de teclado.
  - **./hdd**: Archivos relacionados a la imagen del disco que usa el Kernel.
    - **./disk.img**: Imagen vacía de disco.
    - **./data/***: Archivos y directorios a copiarse a la imagen del disco. El Makefile se encarga de compilar las tareas a **/src/hdd/data/conway/** como "sse.tsk" y "pg.tsk", según la versión del Juego de Conway (SSE/código en C).

[1] http://en.wikipedia.org/wiki/Conway's_Game_of_Life
[2] http://wiki.osdev.org/Main_Page

- **./kernel**: Los fuentes modificades del Kernel del TP3 original (mmu, scheduling, etc).
  - **./a20.asm**: Instrucciones para habilitar/deshabilitar el redireccionamento de memoria mayor a 1MB.
  - **./gdt.***: GDT.
  - **./i386.h**: Instrucciones de ASM en C.
  - **./idt.***: Descriptor de interrupciones.
  - **./isr.***: Implementación de interrupciones.
  - **./kernel.asm**: Código del Kernel.
  - **./mmu.***: MMU.
  - **./pic.***: Funciones para controlar el PIC.
  - **./sched.***: Scheduler.
  - **./tss.***: TSS.
  - **./v16.asm**: Instrucciones para pasar de Modo Protegido a Modo Real y viceversa, cuando el Kernel ya hizo el primer traspaso a Modo Protegido.
- **./macros**: Macros utilizadas por código ASM.
  - **./graphics.mac**: Macros del driver gráfico
  - **./macrosmodoprotegido.mac**: Macros del TP3 original.
  - **./macrosmodoreal.mac**: Macros del TP3 original.
  - **./v16_macros.***: Macros para hacer interrupciones de Modo Real cuando se está ejecutando en Modo Protegido.
- **./tasks**: Tareas.
  - **./bin/***: Binarios de las tareas.
  - **./conway.***: Implementación del Juego de Conway.
    - **./gpr/***: Implementación de la transisión de estado usando Registros de Propósito General.
    - **./sse/***: Implementación de la transisión de estado optimizada con SSE.
    - **./conway.***: Funciones básicas.
    - **./conway_fn.h**: Header para la implementación SSE/GPR
    - **./conway_utils.***: Implementación de llamadas al sistema necesarias para la tarea.
    - **./init.***: Entry point de la tarea.
  - **./idle/***: Tarea Idle.
  - **./system.***: Implementación de algunas llamadas al sistema generales.
- **./bochsrc**: Archivo de configuración para el bochs.
- **./dsk_orig.img**: Imagen del disco, original del TP3.
- **./MakeFile**: El Makefile del TP.

- **.img_to_life**: Script auxiliar que usa OpenCV para leer una imagen monocromática y traducirla a un código, que luego puede pegarse en el fuente de la tarea del Juego de Conway para setear el estado inicial.
  - **./data/rake.bmp**: Imagen de un estado del Juego de Conway.
  - **./src/***: El script y su Makefile.
- **./compile.sh**: Archivo para compilar el TP y correrlo en BOCHS (usando un path relativo al emulador, por lo que podría no funcionar en otra máquina).
- **./qemucompile.sh**: Archivo para compilar el TP y correrlo en QEMU (supone que se puede ejecutar "qemu" desde cualquier directorio).

3. Modificaciones del Kernel original

Se fijo el intervalo de interrupcion de reloj en 1ms por tick. Las entradas libres de la GDT y TSS se obtienen por funciones (gdt=*entradaLibreGDT()*, tss=*nuevoTSS(CR3, ESP, EIP)*), y ahora se permite reciclarlas (*releaseSegment(gdt)*, *releaseSegmentAndTSS(tss)*).

3.1. Interrupciones y excepciones

3.1.1. INT 7

Tras un Task Switch, el bit 8 del CR0 se pone en 1. Cuando está seteado y se intenta usar alguna instruccion FPU/MMX/SSE, se salta a esta excepción. Con este comportamiento en mente, hubo que modificar la rutina de atención para actuar cuando el bit 8 del CR0 está en 1. En tal caso, carga el contexto correspondiente a la tarea actual (más información en la sección del Scheduler) y se pone dicho bit en 0. En caso contrario, la excepción continúa como cualquier otra.

3.1.2. INT 32 (clock)

A modo experimental, con la idea de hacer una pantalla de crasheo del sistema animada, se me ocurrió tomar mediciones de la cantidad de ciclos de clock con *rdtsc* y tener una aproximación de una unidad de tiempo.

Para eso, se calcula la diferencia en la cantidad de ciclos de clock que hubo entre la interrupción número 1 y 17, con el objeto de tener un promedio de la cantidad de ciclos por interrupcion de timer.

Al final no hice ninguna animación. De haber un crasheo, aparece un contador de la cantidad de segundos desde el crash.

3.1.3. INT 67 (sleep)

Pone a dormir una tarea tantos ticks de reloj como indique EAX. Como cada tick ocurre cada 1ms, la tarea no tomaría control hasta que pasen EAX ms.

3.1.4. INT 68 (wait)

Traba a una tarea en un loop hasta que pasen tantos ticks de reloj como indique EAX. A diferencia de INT 67, la tarea permanece "activa" y puede tomar el control del CPU como si no hubiera llamado a la interrupción, pero permanecerá trabada hasta que pasen EAX ms.

3.1.5. INT 69 (driver gráfico)

Si EAX = 1, devuelve en EAX el ancho de pixels de la pantalla.

Si EAX = 2, devuelve en EAX el alto de pixels de la pantalla.

Si EAX = 3, devuelve en EAX la posicion inicial en memoria desde donde se puede dibujar la pantalla.

Si EAX = 4, dibuja usando un solo color lo que indique la pila apuntada por EBX:
EBX + 00 = (unsigned char *) Puntero a la lista de bytes a escribir. Cada byte representa dos pixels, y cada pixel se marca con 1 o 0.
EBX + 04 = (unsigned int) Cantidad de bytes a leer.
EBX + 12 = (unsigned int) Posicion X inicial de la pantalla.
EBX + 16 = (unsigned int) Posicion Y inicial de la pantalla.
EBX + 24 = (unsigned char) Color de base a pintar.
EBX + 28 = (unsigned char) Sumar al color esta cantidad solo si el byte del pixel es distinto de cero.

Si EAX = 6, imprime un string terminado con 0 segun lo que indica la pila apuntada por EBX:
EBX + 00 = (unsigned char *) Puntero al string.
EBX + 04 = (unsigned int) Posicion X inicial de la pantalla.
EBX + 08 = (unsigned int) Posicion Y inicial de la pantalla.
EBX + 12 = (unsigned int) Color de base en formato $0|0|0|C$ o $0|R|G|B$.
EBX + 16 = (unsigned char) El formado del color de base es un solo color (0) o RGB (otro valor).
Devuelve en EAX la cantidad de pixels corridos en X donde se deberia seguir imprimiendo

3.1.6. INT 70 (memoria dinámica)

Si EBX = 0:
Reserva tantas paginas de memoria como haga falta para ocupar lo pedido en bytes en EAX. Devuelve en EAX la posicion de memoria que apunta al primer byte o NULL (0) si no se pudo obtener la memoria pedida.

Si EBX = 1:
Libera la memoria dinámica asignada a la direccion pasada en EAX.

3.1.7. INT 71 (teclado)

Si EBX = 0, pide el foco del teclado (activa el buffering si estaba desactivado).

Si EBX = 1, procesa y devuelve el siguiente código en el buffer.

Si EBX = 2, procesa y devuelve el siguiente caracter en el buffer.

Si EBX = 3, pide el último código procesado.

Si EBX = 4, pide el último caracter procesado.

3.1.8. INT 72 (file system)

Si EAX = 0:
Pide información sobre un directorio, conteniendo en EBX el string del path al mismo, y en ECX la dirección a la posición de memoria donde depositarla.

Si EAX = 1:
Pide cargar el archivo cuyo path está en EBX como un proceso y ponerlo a correr.

Si EAX = 2:
Pide cargar los archivos cuyo path están en EBX y ECX como procesos y ponerlos a correr.

3.2. MMU

3.2.1. Organización

La información que maneja la MMU se organiza en cinco estructuras, con formato de lista enlazada:

- **task_mmu_list**: Guarda la información de la memoria dinámica utilizada por un proceso, como la cantidad de espacio pedido, la memoria virtual asignada y la memoria fisica reservada.
  - `task_mmu_list` *next
  - int pid
  - uint space_used
  - `task_alloc` *alloc
  - `task_phys_stat` *pages
- **task_alloc**: Intervalos de memoria virtual usados (sin relacionarlos con la memoria física).
  - `task_alloc` *next
  - uint virtual_from
  - uint size
- **task_phys_stat**: Qué partes de una página de memoria reservada están siendo usadas.
  - `task_phys_stat` *next
  - uint page
  - uint virtual_address
  - `task_alloc_desc` *space
- **task_alloc_desc**: Intervalos de memoria virtual usados (relacionados con la memoria física por la estructura `task_phys_stat`).
  - `task_alloc_desc` *next
  - uint mem_from
  - uint space
- **mmu_recycle_list**: Direcciones de páginas de memoria.
  - `mmu_recycle_list` *next
  - uint page

Para permitir el reciclado de estas estructuras, hay una lista enlazada "de recursos libres" por cada una. Cuando se pide alguna, por ejemplo mediante *mmu_list = *get_free_task_mmu_list()*, se devuelve la primera entrada, y se hace avanzar la lista de recursos libres.

En el caso en que no tenga primera entrada (apunte a NULL), se pide una página de Kernel. Como son listas enlazadas, la memoria obtenida se parte en varias `task_mmu_list`, con la 1ra entrada apuntando a la 2da, y así hasta que se acaben los 4KB. La lista de recursos libres se apunta a la 1ra entrada y se procede como en el caso de arriba.

Si se quiere liberar la estructura, basta con llamar a *release_task_mmu_list(*mmu_list)*, que la pondrá como primera entrada en su lista "de recursos libres".

3.2.2. Pedido y liberación de memoria dinámica

La idea es trabajar siempre con la dirección de memoria virtual más chica disponible, por lo que las listas de recursos usados por el proceso se mantienen ordenadas ascendentemente. Esto permite que cuando llegue un *malloc(size)*, la MMU pueda chequear eficientemente si hay algún pedazo de memoria virtual no utilizado de tamaño mayor o igual a *size* entre el "start" (0x500000) y el 1er intervalo de memoria, o entre el 1er intervalo y el 2do, y así. Si no lo hay, trabaja con la primera dirección virtual libre.

Se piden tantas páginas físicas como haga falta para cumplir el pedido. Si en tal dirección virtual, la MMU tiene una parte de la memoria física sin uso, se la utiliza lo más que se pueda, y luego, si es necesario, se procede a pedir otras páginas.

Cuando se quiere liberar un pedazo de memoria con *free(mem)*, se busca a qué `task_alloc` pertenece y a partir de ahí, se obtiene el intervalo de la memoria virtual a liberar. Tras reciclar el `task_alloc`, se van buscando los `task_phys_stat` que referencian alguna parte del intervalo, y de sus particiones `task_alloc_desc`, se van liberando las que estén incluídas en el mismo. Si el `task_phys_stat` se queda sin particiones, se recicla, así como la página de usuario que referenciaba.

3.2.3. Recursos de una tarea

Usando `mmu_recycle_list`, se mantienen dos listas enlazadas de páginas de memoria recicladas, una de Kernel y otra de usuarios. Cuando una tarea necesita un directorio, o mapear una parte de la memoria, la MMU se encarga de manejar las páginas de Kernel que necesita. Cuando el scheduler quiera liberar los recursos de la tarea, la MMU reciclará tanto la memoria dinámica utilizada, como la memoria utilizada por el directorio y las tablas de página.

3.3. Scheduler

3.3.1. Organización

El scheduler utiliza dos estructuras para manejar su información:

- **task_list**: Hay un array de 17 elementos de esta estructura (la cantidad máxima de tareas concurrentes). Mantiene su índice en el arreglo (recycleId) para poder reciclarse eficiente-

mente. A su vez, funciona como lista doblemente enlazada, y posee la información de una tarea.

  - unsigned int recycleId
  - `task_list` *prev
  - `task_list` *next
  - `taskStr` tsk

- **taskStr**: Información necesaria para mantener el estado de una tarea. En fxData se guardan los datos de los registros XMM, MMX, FPU, etc, para recuperarlos tras un context switch si fxValid indica que los datos guardados son válidos.

  - unsigned int fxData
  - unsigned short gdtOffset
  - unsigned char fxValid
  - unsigned short pid
  - unsigned char delete_later
  - unsigned char keyCode
  - unsigned int sleepTime
  - unsigned int waitTime
  - unsigned int activeTime
  - unsigned int parent_pid
  - unsigned int dir
  - unsigned int pages
  - unsigned int stack_page

Se mantienen dos listas doblemente enlazadas, una de tareas activas y una de tareas dormidas. La tarea `IDLE` pertenece a la lista de tareas activas sólo cuando no hay otra tarea activa. La idea es tener un puntero a la `task_list` de la tarea activa en el momento, y hacer que el puntero avance al siguiente elemento de la lista cuando se tenga que cambiar de tarea.

3.3.2. Tareas activas

Una tarea está en estado "activo" cuando pertenece a la lista de tareas activas. Cuando hay una nueva tarea, o se despierta alguna que haya estado dormida, se inserta antes de la tarea actual en la lista de tareas activas. Esto permite que una tarea activa tenga garantizado el control del procesador tras un tiempo determinado, siempre que permanezca activa (puede eliminarse antes de que le toque su turno).

Cada tarea *tsk* tiene un quantum asignado (*tsk.activeTime*). Cuando la tarea actual cambia, un contador se resetea y con cada tick del clock, éste se incrementa en una unidad. La tarea actual cede su turno a la siguiente en la lista cuando cuando dicho contador es mayor o igual al quantum de la misma. Como los tick de reloj ocurren cada 1ms, *tsk.activeTime* indica la cantidad de ms que una tarea tiene control del procesador (contando los tiempos que pueden tomar las interrupciones externas).

A su vez, las tareas activas pueden estar "esperando" (**INT 68**). En este pseudoestado, toman control del procesador como cualquier tarea activa, pero están atrapadas en un loop mientras el

tiempo de espera no haya pasado. Dicho tiempo (*tsk.waitTime*) decrementa con cada tick de clock, hasta llegar a 0, en cuyo caso, cuando la tarea tenga el control, podrá salir del loop y volver a su código.

En caso de que la tarea actual sea la `IDLE` y haya expirado su quantum, la nueva tarea toma el control, y se mueve la `IDLE` a la lista de tareas dormidas, fijando el tiempo de sleep como indeterminado ($tsk.sleepTime = 0$).

3.3.3. Tareas dormidas

Una tarea está en estado "dormido" cuando pertenece a la lista de tareas dormidas, y puede entrar en este estado por su cuenta, vía **INT 67**, o porque fue marcada para ser eliminada ($tsk.delete\_later = 1$). En el primer caso, la tarea tiene definido un tiempo de sleep ($tsk.sleepTime \neq 0$), que se irá decrementando en cada tick de clock. Cuando el tiempo llega a cero, se saca de la lista de tareas dormidas y se la inserta en la de tareas activas.

Si una tarea está tiene $tsk.sleepTime = 0$, se considera que está dormida por tiempo indefinido. Este estado particular ocurre sólo para la tarea `IDLE` y las tareas que fueron marcadas para ser eliminadas.

3.3.4. Eliminación de tareas

Una tarea puede ser eliminada porque generó una excepción, o bien porque se quieren eliminar todas y se presionó la tecla especial F4. En el primer caso, se salta inmediatamente a la siguiente tarea activa, o a la `IDLE` si no hay otra. En el segundo caso, se salta inmediatamente a la tarea `IDLE`. En ambos, la o las tareas afectadas se marcan para ser eliminadas ($tsk.delete\_later = 1$).

Cuando toma control del procesador, la tarea `IDLE` se encarga de liberar los recursos de las tareas marcadas. Esto significa que si se termina una tarea, habiendo más tareas activas (y por lo tanto, la `IDLE` no pudiendo tomar el control), las eliminadas seguirán ocupando recursos.

Por otro lado, como se verá mś adelante en la sección del Driver de Disco, sólo la `IDLE` puede crear tareas, ya que comparte el *process id* con el Kernel, y tiene acceso a la memoria dinámica del mismo. En este trabajo práctico no se exploró la posibilidad de permitir que otras tareas puedan generar procesos. De intentarlo, probablemente generen una excepción.

4. Drivers

4.1. Interfaz gráfica

Como VGA está limitado en el tamaño de resolución y la profundidad del color, opté por explorar cómo superar este límite. En OSDev encontré información sobre los modos VESA, y cómo seleccionarlos.

Siguiendo las instrucciones de la página[3], se implementaron en ASM las funciones *get_vesa_info()*, *get_mode_info(mode)*, *set_vesa_mode(mode)* y *mode = find_vesa_mode(width, height, depth)*.

Para las primeras tres, fue necesario usar la interrupción 0x10 que ofrece la BIOS, por lo que hubo que pasar a modo real, hacer la interrupción con los registros en los valores correspondientes, y finalmente volver a modo protegido.

Con la 4ta función, dada una resolución y profundidad de color buscados, se puede obtener el modo VESA apropiado, si es que existe. El Trabajo Práctico está corriendo en 1024x768 con 256 colores, aunque puede pasarse a 24 o 32 bits de profundidad. Se escribieron funciones en C para manejar el dibujo de pixels y la impresión de strings en 8, 24, y 32 bits.

4.2. Teclado

Una tarea puede tomar "foco" o control del teclado (habilitando el buffering), y en tal caso, el scheduler le permite consultar sobre el keycode de las teclas que se fueron presionando, o el caracter imprimible del keycode actual.

Las interrupciones de teclado son atendidas por este driver, que va procesando los keycodes que recibe, o bien los va apilando en un buffer de 1KB. En el caso en que una tarea haya pedido control del teclado, ésta puede pedirle al Kernel con la **INT 71** (atendida por el scheduler para controlar que la tarea tenga el "foco") que procese el siguiente keycode en el buffer y recibir éste, o su caracter imprimible. También permite consultar por el último keycode o caracter sin necesidad de procesar el buffer.

Para saber el estado de cada tecla, hay un mapa de teclas presionadas. Cuando se procesa un keycode, se modifica en el mapa el estado de la tecla a la que hace referencia.

La tecla F4 es un caso especial que ignora el buffering. Al recibir una interrupción informado que se presionó, el driver avisa al scheduler que hay que eliminar todas las tareas activas, y luego continúa como siempre (buffereando o procesando el keycode).

4.3. Disco

Para poder acceder a archivos de disco, se implementaron dos drivers, uno de inicialización y lectura de sectores (hdd), y un driver FAT16 para acceder al sistema de archivos y leerlos.

El Makefile copia los archivos en el path *src/hdd/data/** a la imagen de disco rígido, por lo que es muy sencillo armar la estructura del disco virtual.

[3] http://wiki.osdev.org/Getting_VBE_Mode_Info

Al igual que con el driver gráfico, se aplicó mucha de la información y organización estructural que se encontró en OSDev.

4.3.1. Driver HDD

Para inicializar el disco, se chequea que el bus IDE tenga algo conectado y luego se utiliza el comando `IDENTIFY`[4]. Si todo marcha sin problemas, se habilita la lectura del disco con la función *read_hdd(buffer, lba, sectors)*.

Esta función se encarga de leer[5] *sectors* sectores a partir de LBA=*lba*, guardando los datos en la direccion de memoria apuntada por *buffer*.

4.3.2. Driver FAT16

Las estructuras y el código para la inicialización del file system fueron implementadas según recomendaba OSDev[6], así como la lectura de clusters. A eso se le agregó una organización archivos y directorios propia, generada a partir del parseo de los clusters del file system.

Tanto la Tabla FAT, como los buffers utilizados para leer de disco, y el árbol de directorios utilizan memoria dinámica bajo *process id* 0, por lo que la info sólo es accesible por el Kernel y la tarea `IDLE`. Como se mencionó en la sección del Scheduler, si otra tarea intenta cargar un proceso, la memoria de estos recursos estaría apuntando a otro lado, y podría provocar una excepción.

4.3.3. Driver FAT16 - Archivos y directorios

Se utilizaron dos estructuras para representar el contenido del disco:

- **fs_directory**: Contiene la información del contenido de un directorio.
  - `fs_directory` *subdirectories
  - `fs_directory` *parent
  - `fs_directory` *prev
  - `fs_directory` *next
  - `fs_file` *files
  - uint cluster
  - char *name
- **fs_file**: Contiene la información pertinente a un archivo.
  - `fs_directory` *parent
  - `fs_file` *prev
  - `fs_file` *next
  - char *name

[4] http://wiki.osdev.org/ATA_PIO_Mode#IDENTIFY_command
[5] http://wiki.osdev.org/ATA_PIO_Mode#28_bit_PIO
[6] http://wiki.osdev.org/FAT16

- uint cluster
- uint size

Siguiendo las instrucciones de OSDev[7], se implementaron dos funciones para construir el árbol con la información del file system:

La función *parse_cluster(buff, entries, lfn, parent)* se encarga de tratar lo que apunta *buff* como información del directorio *parent*. Leyendo los clusters pertinentes del root y aplicando esta función se inicializa su `fs_directory`.

Por otro lado, existe la función recursiva *parse_content(directory)*, que requiere que *directory* sea un `fs_directory` inicializado. Ésta toma cada subdirectorio de *directory* y va leyendo sus clusters y parseándolos con *parse_cluster(...)*, para luego aplicarse al subdirectorio recién inicializado. Aplicada al `fs_directory` del root, termina de construir el árbol del file system.

4.3.4. Driver FAT16 - INT 72

La INT 72 permite que una tarea pida el contenido de un directorio dado un path. Para esto, se implementó la función *get_directory_contents(path, out)* que busca el directorio ubicado en *path* y, si existe, lo recorre y copia la información a *out*, un puntero a una estructura definida de la siguiente manera:

- char **files
- char **dirs

donde *files* es un array de nombres de archivo y *dirs* es un array de nombres de subdirectorio. Ambos arreglos tienen un último elemento como `NULL`.

Por otra parte, la INT 72 también permite que una tarea cargue procesos a partir de uno o dos archivos, dados sus paths. Con la función *file = open_file(path)*, se obtiene un puntero a una posición de memoria donde se guardó el contenido del archivo leído de disco. Si *file* apunta a `NULL`, el archivo no se pudo cargar. En caso contrario, se le pide al Scheduler la creación de un proceso por cada archivo, y se fuerza la terminación del quantum de la tarea.

Como la tarea `IDLE` es la única que puede explorar el file system, dado que comparte la memoria dinámica del Kernel, la terminación de su quantum permite que recién vuelva de la interrupción cuando todos los procesos hayan sido eliminados. Al volver de la INT 72, el shell de comandos limpiaría la pantalla y continuaría normalmente.

[7] http://wiki.osdev.org/FAT16#Reading_Directories

5. Tareas

5.1. Juego de Conway

Para representar el escenario del Juego de la Vida, hay dos matrices S_0 y S_1 de $(w/2+2) \times (h+2)$, donde h y w son el alto y el ancho en pixels de la pantalla en que se dibujarán las celdas vivas/muertas, y S_1 y S_0 representan los estados nuevo y actual de las celdas.

En cada elemento de la matriz se guardan 8 bits, de los cuales 4 se destinan a una célula. Es decir, en cada S_j[x][y] hay dos células c_h y c_l. Esto permite aprovechar más la memoria pero conlleva realizar operaciones con 4 bits (nibble) en vez del mínimo de 8 que usan las instrucciones del microprocesador.

Como el Juego de Conway requiere revisar los estados de los vecinos de una célula, para evitar problemas en los bordes se incrementó en dos unidades tanto la altura como el ancho de la matriz de estados. De esta forma, la primera célula es aquella en el nibble más significativo de $S_j[1][1]$, mientras que el estado de la última célula de la fila quedaría representado por el nibble menos significativo de $S_j[w/2][1]$.

	$S_j[0][y]$		$S_j[1][y]$		...	$S_j[w/2][y]$		$S_j[w/2+1][y]$	
$F_0 =$	-	0	0	0	...	0	0	0	-
$F_1 =$	-	0	c_1	c_2	...	c_{w-1}	c_w	0	-
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$F_h =$	-	0	$c_{(h-1)*w+1}$	$c_{(h-1)*w+2}$	...	c_{h*w-1}	c_{h*w}	0	-
$F_{h+1} =$	-	0	0	0	...	0	0	0	-

Cuadro 1: Matriz de estado del Juego de Conway. Cada celda contiene 1 byte.

Como muestra el Cuadro 1, cada c_i es el estado de la célula i (1 si está viva, 0 si no). Cabe aclarar que para la primera columna, nunca se usa el nibble más significativo, así como no se utiliza el menos significativo de la última columna

Utilizando un script, se tradujo una imagen de un estado interesante del Juego de Conway[8] a instrucciones para inicializarlo en la tarea.

5.1.1. Optimización

Salvo el procedimiento de pasar de un estado a otro, tanto la versión SSE como la versión GPR son iguales. Ambas versiones implementan la función *conway_step*(s_0, s_1, *alto*, *ancho*), donde s_0 y s_1 representan la matriz de estados actual y nueva, respectivamente.

Como recordatorio de las reglas del Juego, si un célula i está viva ($c_i = 1$), la suma de sus vecinos $sumv_i$ debe ser 2 o 3 para que no muera. En caso contrario ($c_i = 0$), la suma de sus vecinos debe ser 3 para que reviva (ver Cuadro 2). Esto permite definir el estado al que pasa una célula i como $c'_i = ((c_i|sumv_i) == 3)$, donde | es la operación *or* bit a bit.

La versión GPR recorre la matriz s_0 de izquierda a derecha y de arriba hacia abajo. Para cada célula, obtiene su estado (c_i) así como la suma de los estados de sus ocho vecinas ($sumv_i$), calcula $c'i$ y lo setea en s_1.

[8] http://en.wikipedia.org/wiki/File:Rake_selection.gif

$$\begin{array}{cccc|cccc} c_a & c_b & c_c & || & 1 & 0 & 1 & \\ c_d & C & c_e & || & 0 & 0 & 0 & \rightarrow C' = 1 \\ c_f & c_g & c_h & || & 1 & 0 & 0 & \end{array}$$

Cuadro 2: La célula C, sus vecinos y un ejemplo. Si C está viva (C=1), sigue viviendo *sii* tiene 2 o 3 vecinos vivos. Si no (C=0), revive *sii* tiene tres vecinos vivos.

La version optimizada recorre la matriz s_0 de arriba hacia abajo, y de izquierda a derecha, procesando 32 células (16 celdas) en cada pasada (las que puede guardar un registro XMM). En principio, obtiene la suma superior S_{sup} y la suma media S_{med} para las primeras 32 células, cuyos estados mantiene en Cs. Es decir:

$$\begin{array}{rccc|c|cc} S_{sup} = & sums_0 & sums_1 & | & \ldots & | & sums_{30} & sums_{31} \\ S_{med} = & summ_0 & summ_1 & | & \ldots & | & summ_{30} & summ_{31} \\ Cs = & c_0 & c_1 & | & \ldots & | & c_{30} & c_{31} \end{array}$$

donde $sums_i$ es la suma de los tres vecinos en la fila superior a la célula i y $summ_i$ es la suma de los dos vecinos en la misma fila que la célula i.

(*) A continuación, obtiene los estados de las células Cs_{inf} en la fila inferior y calcula la suma inferior S_{bot} que faltaba para Cs. De esta forma, en $S_c = S_{sup} + S_{med} + S_{bot}$ quedan las sumas de los vecinos de las 32 células en Cs, y como cada c_i puede valer 0 o 1, en S_c está la cantidad de vecinos vivos. Se procede a hacer la operación $R = S_c|Cs$, y como vimos antes, para cada nibble dicha célula queda viva en el nuevo estado *sii* su nibble en R es 3.

Ahora hay que procesar Cs_{inf}, las nuevas células. Su suma superior es la S_{med} que tenemos, agregando el valor de cada célula, es decir. $S'_{sup} = S_{med} + Cs$. Su suma media S'_{med} puede precalcularse al momento de obtener S_{bot} en (*). Con setear $Cs' = Cs_{inf}$, quedamos en el mismo estado que para las primeras 32 células, pero respecto a las 32 inferiores, por lo que se puede repetir el procedimiento desde (*).

Habiendo calculado el nuevo estado para las primeras 32 células de todas las filas, se empieza nuevamente por la primera fila, 32 células a la derecha, y se repite el procedimiento hasta que ya no se puedan leer 16 celdas de corrido.

Si $w/2$ era múltiplo de 16, ya están procesadas todas las células. En caso contrario, se hace una última pasada por las últimas 32 células de la derecha, teniendo que repetir el procedimiento para algunas.

5.2. Idle

Al ser ésta la primera tarea que toma el control del procesador, fue conveniente que actúe de shell de comandos para permitir generar procesos a partir de archivos. Al ejecutarse un proceso, la tarea `IDLE` cede el control y recién lo recupera cuando todos los demás procesos se hayan terminado/eliminado.

Fue necesario poder escribir texto en pantalla en modo gráfico, por lo que se implementó la función *scr_printf*, que acepta algunos de los formatos del *printf* de `C`, así como variantes que permiten posicionar en pantalla el texto a escribir, y modificar su color.

También fue necesario leer y procesar input del teclado, tema que en su mayoría ya está resuelto

por el Kernel (`INT 71`). La tarea añade el uso la tecla `TAB` para autocompletar la última palabra escrita a un archivo o directorio.

Para explorar el disco rígido se necesitó obtener, dado un path, un listado de los archivos y subdirectorios de éste, tema también resuelto por el Kernel (`INT 72`).

Finalmente, para ejecutar tareas, sólo hace falta llamar al Kernel con el path al archivo que contiene el código crudo en binario, el cuál se copiará a memoria y se ejecutará (`INT 71`).

Como fue interesante comparar en tiempo real las soluciones optimizada y no optimizada del Juego de Conway, la tarea `IDLE` puede ejecutar dos procesos a la vez, en vez de uno (pierde el control hasta que ambos terminen o sean eliminados).

Los comandos que acepta son:

- **help**: Muestra esta lista.
  `$> help`
- **ls**: Informa el contenido del directorio actual.
  `$> ls`
- **cd [dir]**: Cambia el directorio actual al directorio con path [dir] relativo al actual.
  `$> cd conway`
- **ld [path]**: Carga y transfiere el control a la tarea en [path], relativo al directorio actual.
  `$> ld sse.tsk`
- **ld2 [t1], [t2]**: Carga simultaneamente y transfiere el control a las tareas en [t1] y [t2], relativas al directorio actual.
  `$> ld2 sse.tsk, pg.tsk`
- **clear**: Limpia la pantalla.
  `$> clear`

6. Conclusiones

Este TP partió de la idea de comparar en tiempo real dos procesos, uno optimizado y otro no. Con cada necesidad, por la naturaleza de ser el programador del Kernel, se tuvieron que ir agregando más y más cosas.

Ya sea leer y procesar los inputs del teclado, o tener que hacer interrupciones en Modo Real para setear el modo gráfico, el Kernel debe actuar de capa intermedia de muy bajo nivel para inicializar el ambiente en que corren los procesos, y asegurar que las tareas se ejecuten aisladas (salvo casos de comunicación de procesos, que este trabajo no contempla) y con un grado de simultaneidad.

Fue particularmente interesante meterse con "la magia" a bajo nivel. Tener control total del procesador. Tener que organizar la memoria para la correcta ejecución de los procesos. Pensar en cómo podría funcionar un scheduler de procesos. De qué forma una tarea podría dibujar o escribir en pantalla. Qué interfaz ofrecer para procesar los inputs del teclado.

Siendo sólo usuario de un S.O., uno no tiene idea de la complejidad que tiene atrás un Linux o un Windows. Algo que parece tan simple como dibujar una pantalla, o navegar por un sistema de archivos, tiene todo un mundo detrás, y eso es algo que pude apreciar en este Trabajo Práctico.