BrunOS®

Un sistema operativo .. diferente!

Karpovsky - Martinez - Mesa

Presentado el

[21/05/2012]

Introducción

El trabajo práctico especial propuesto por la cátedra consistió en realizar la implementación de un sistema operativo que muestre las características del modo protegido de los microprocesadores de Intel y que hagan uso de las nociones de multi-procesamiento, y scheduling.

El presente informe tiene por objetivo introducirle al usuario todas las herramientas y comandos con los que cuenta el sistema operativo BrunOS® como así también comentar los detalles de la implementación del mismo.

Interfaz gráfica

La mayoría de los sistemas operativos que no cuentan con un entorno gráfico como el de Windows y corren sólo bajo modalidad consola, no cuentan con una buena interfaz gráfica implementada. Debido a ésto, nuestra propuesta de desarrollo no viene a innovar en cuanto a comandos ofrecidos para el usuario, pero sí trae una amigable interfaz de usuario en la que éste podrá sentirse cómodo.

El usuario tendrá a la vista todos los parámetros de sistema en forma constante, implementados como header de cada terminal. Ésto le permitirá poder saber cuestiones como por ejemplo la distribución de teclado activa o la terminal en la que está escribiendo, sin la necesidad de ejecutar un comando.

Grub

Apenas uno inicia el ordenador con el disco de BrunOS® insertado, si su PC está configurada para bootear desde la lectora de CD, el Grub cargará el Sistema Operativo y usted verá una pantalla como la siguiente:

Tendrá 30 segundos para seleccionar qué sistema operativo desea iniciar y, en caso de que usted no presione ninguna tecla, se iniciará BrunOS® por defecto.

Visualización inicial

Una vez dentro del SO, usted visualizará una interfaz como la siguiente:

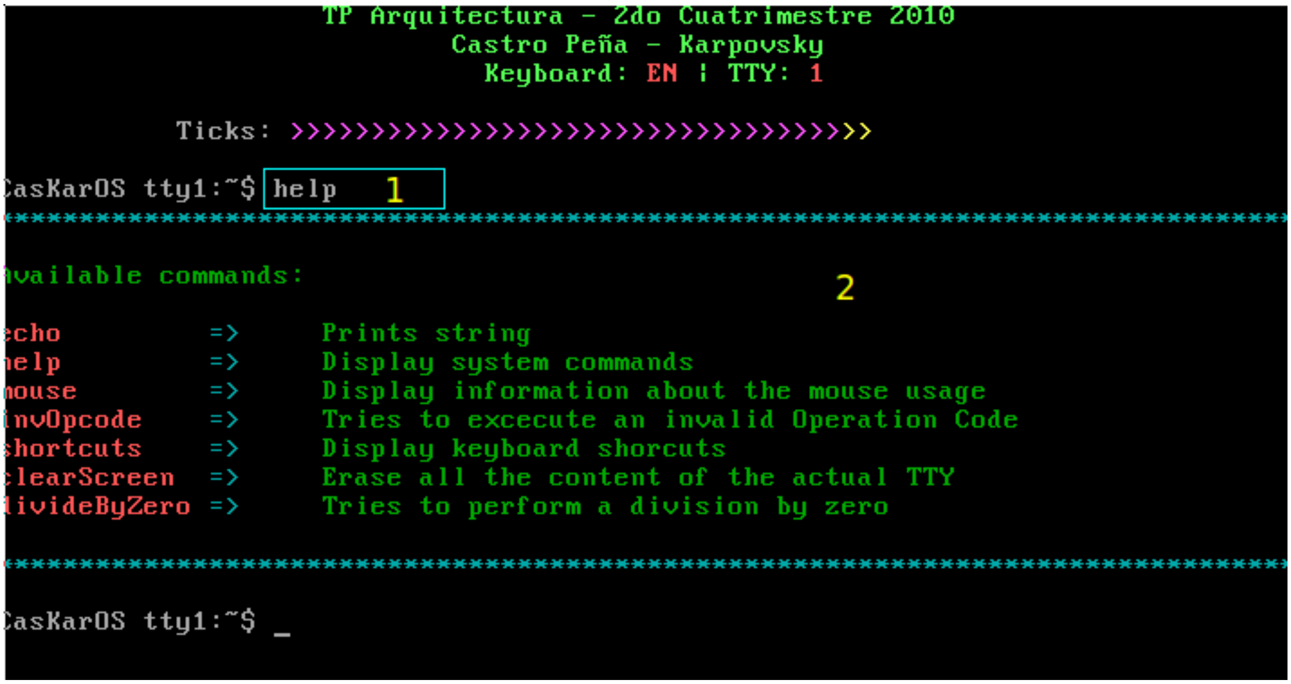
1- Encabezado con el nombre de la materia y del equipo desarrollador.
2- Distribución de teclado sobre la cual usted está trabajando *(ES: Española / EN: Inglesa)*.
3- Número de TTY: Indica en cuál de las 5 TTY's usted se encuentra actualmente.
4- Se muestra de una manera gráfica las interrupciones generadas por el TimerTick. Cuanta más frecuencia tengan las interrupciones, más veloz correrá la señalización (>>>>>>>>>>)
5- Prompt: Aquí se le repite al usuario la TTY actual por una cuestión de comodidad.

Comandos de Consola

A continuación se enumerarán todos los comandos que pone a su disposición BrunOS®. Para ejecutarse, los mismos deben ser ingresados en la consola y luego se debe presionar la tecla **Enter**.

- **Help**

El comando **Help** nos perimte visualizar una lista de todos los otros comandos y opciones disponibles del sistema operativo. Al ejecutarlo usted verá la siguiente pantalla:



1- Comando **Help** escrito al lado del promtp
2- Salida del comando **Help**: En rojo se visualizan otros comandos disponibles y próximos a ellos, en verde, se detalla una pequeña descripción de su funcionalidad.

- **Echo**

El comando **echo** escribe en la salida estandar el texto que se le pase como parámetro.
Por ejemplo, si ejecutamos: ***echo Arquitectura 2010***, en la salida estandar veremos:

- **Mouse**

El comando **mouse** nos da información útil sobre las funcionalidades del Mouse en nuestro sistema operativo. En éste caso, el mouse nos sirve para incrementar y/o decrementar las interrupciones por segundo generadas por el TimerTick.

Al presionarse el **Botón Izquierdo**, el PIT es reprogramado y la frecuencia de interrupción del mismo aumenta en 10hz.
Por el contrario, al presionarse el **Botón Derecho**, la frecuencia de interrupción será menor.

Cabe aclararse que existe una limitación por software de la frecuencia mínima y máxima que usted podrá utilizar. Por lo que no debe gastarse en quedarse días haciendo click con el mouse, pensando que va a aumentar hasta el infinito la frecuencia de interrupción, sino que la misma llegará con algunos clicks al máximo disponible.

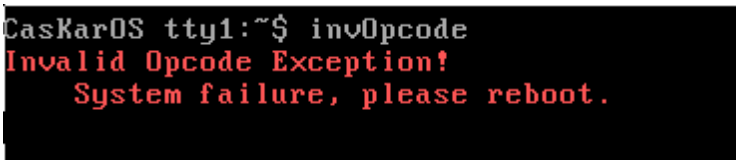
La manera en la que usted verá el cambio en dicha frecuencia es a través del señalizador ubicado en la parte superior de su pantalla:

Apenas comience a aumentar la frecuencia, lo notará dado que la “flecha” se moverá hacia la derecha con mayor velocidad.

- **invOpcode**

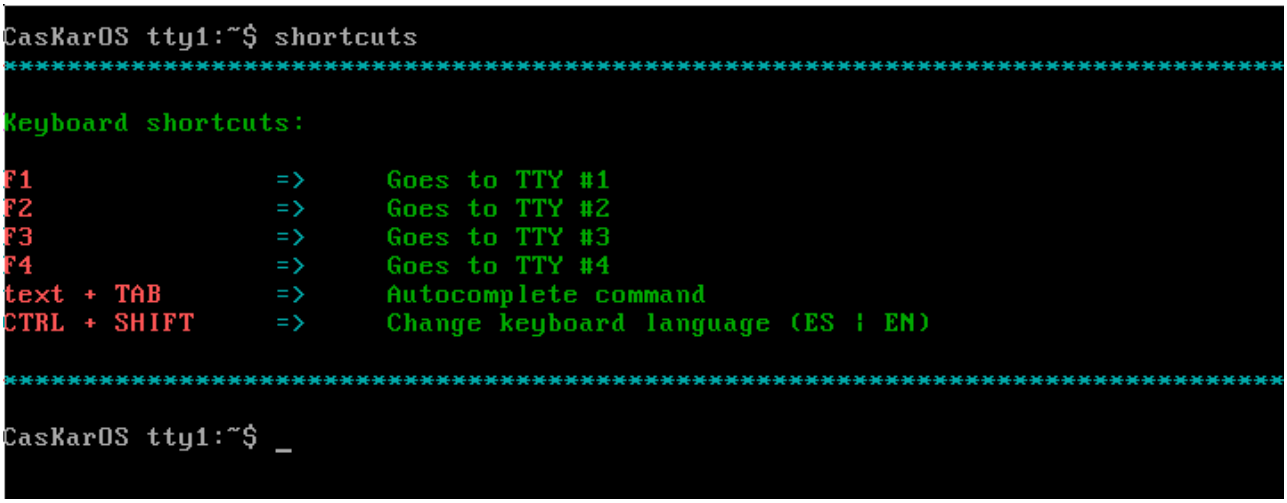
Este comando nos sirve para simular una excepción de **”invalid opcode”**. Una vez ejecutada, usted deberá reiniciar su sistema para poder seguir operando dado que la excepción es crítica y el sistema no puede auto-recuperarse de la misma.

Verá algo como:



- **Shortcuts**

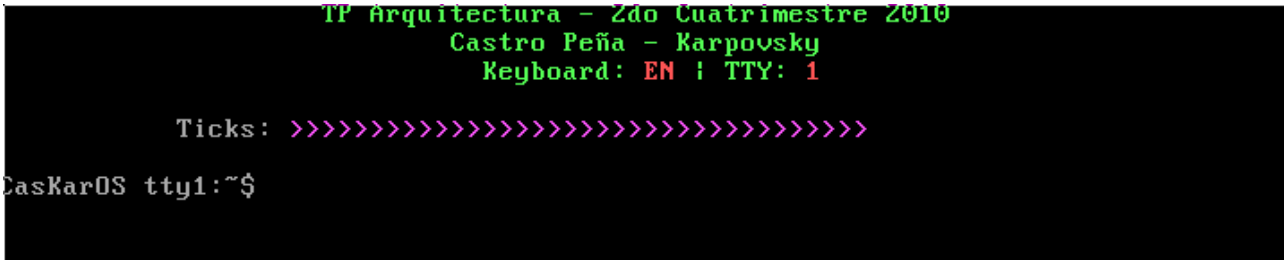
Al ejecutar éste comando, usted verá en la salida estandar una lista con todos los **Shortcuts** de teclado disponibles para ésta versión del SO. En éste caso:



Como bien aclara la imagen, al presionar **F1, F2, F3 o F4**, usted será dirigido a la TTY correspondiente. Ej, al apretar **F3,** usted irá a la **TTY #3.**

Por otro lado, el SO cuenta con el **autocompletado de comandos.** El mismo se utiliza escribiendo parte de un comando y luego presionando la tecla **TAB.** En otras palabras, si usted desea ejecutar el comando, por ejemplo, **divideByZero**, basta con que usted escriba **"di" y luego presione TAB**; el sistema autocompletará el resto del comando.

Al presionar **CTRL + SHIFT**, usted cambiará entre las distintas distribuciones de Teclado. Éste cambio podrá ser visto en el encabezado de su pantalla, como fue explicado en el apartado **1.**
Ejemplo de Trabajo sobre TTY #1

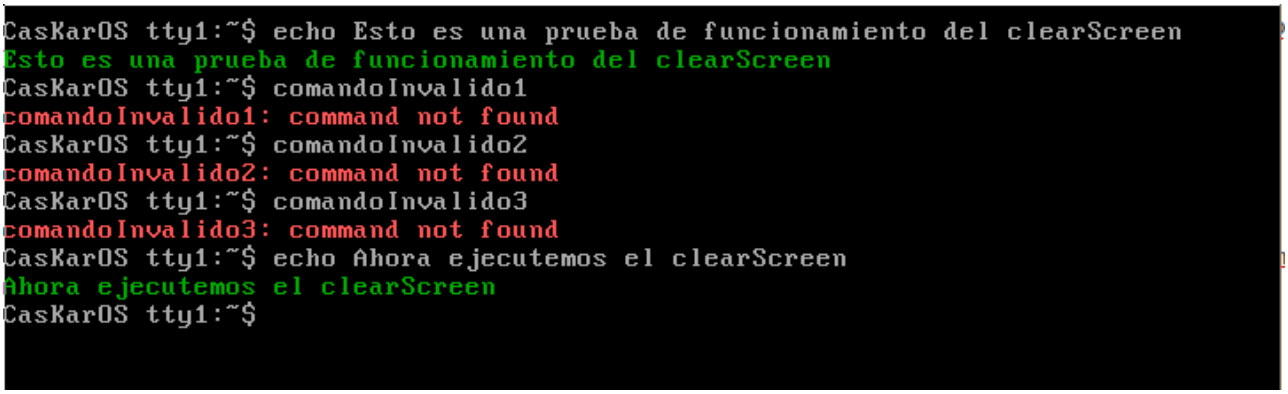


Ejemplo de Trabajo sobre TTY #2

- **Clear Screen**

El comando **clearScreen** tiene una funcionalidad básica pero no por ello poco útil. Al ejecutarse, simplemente borrará todo el contenido de la TTY en la que usted esté trabajando. Es decir, la pantalla será limpiada y usted podrá seguir ejecutando comandos sin ser entorpecido por el rastro dejando por ejecuciones de otros comandos.

Supongamos que su pantalla se encuentra en una situación como ésta:



Luego de ejecutar un **clearScreen**, su pantalla se verá así:

- **Divide by Zero**

Este comando simula una excepción por división por cero. Básicamente su funcionamiento consiste en intentar ejecutar la cuenta 1/0 (uno dividido cero). Lógicametne el microprocesador advierte del potencial error y lanza una excepción por división por cero. Ésta es atendida por el SO, por lo que se le será notificado el fallido intento de realizar ésta operación.

Al igual que en la excepción por “invalid opCode”, al ejecutarse ésta excepción, el sistema quedará tildado y usted deberá reiniciar para recuperarse.

Verá algo como:

- **kill N:** Mata al proceso con PID = N
- **imprimeUnos:** Imprime unos indefinidamente sobre el buffer de video de la TTY actual
- **TOP:** Muestra el detalle de los procesos dentro del SO (estén corriendo, waiting o terminados)

Detalles de implementación

PICs

Utilizando el Bochs para emular el booteo lo primero que se hizo fue reprogramar el PIC y programar la interrupción del teclado.

El el caso puntual del PIC maestro, lo que se realizó es moverlo para que comience en la posición 0x20; por su parte, el PIC esclavo está seteado para comenzar en la posición 0x28. En el proceso de reprogramación del PIC, se le indica al PIC maestro, entre otras cosas, dónde quedará ubicado el PIC esclavo. En nuestro caso, el PIC esclavo está colgado de la IRQ #2 del PIC maestro.

Hablando de las IRQ's, para éste trabajo práctico, se han utilizado la IRQ #0 (timer tick), IRQ #1 (teclado) e IRQ #12 (mouse PS/2). Debido a ésto, una vez movidos los PICs y seteados los handlers tanto para las excepciones como para las interrupciones, se le ha colocado una máscara a ambos PICs para que los mismos atienda sólo las interrupciones que éste trabajo requería. Dichas máscaras fueron calculadas de la siguiente forma:

1 Mascara PIC 1 = F8h:

```
/* 1 1 1 1 1 0 0 0

 * ? ? ? ? ? S K T
 T = Timer – IRQ0;   K = Keyboard – IRQ1;   S = Slave PIC - IRQ2

 */
```

2 Mascara PIC 2 = EFh:

```
/* 1 1 1 0 1 1 1 1

 * ? ? ? M ? ? ? ?
M = Mouse – IRQ12
 */
```

Para el teclado se decidió no utilizar las teclas del teclado numérico ni las flechas ni las teclas Insert, Inicio, Re Pág, Supr, Fin, Av Pág o Esc ya que se consideró que no eran significativas para la funcionalidad de la Shell.

Al tener el teclado, se implementaron las funciones putc(), getc(), printf() y scanf() muy parecidas a las de la librería estándar de C, pero con una funcionalidad menor. Dichas funciones llaman a la INT80h con los parámetros correspondientes de lectura/escritura, la cual recibe estos parámetros junto con otros que indican donde deben leer o escribir y realiza dicha acción.

Una vez implementadas estas funciones se empezó a programar la Shell.

Terminales

Ante la consigna de tener varias terminales, se decidió crear estructuras para representar cada una de las terminales y hacer al código más facil de entender. Cada terminal tiene una pantalla y un teclado, cada uno de estos con su buffer correspondiente. Además cada pantalla de la terminal tiene una posición de escritura y cada teclado sus flags de Shift, Caps, Alt, Ctrl, Num y Scroll, y otras variables necesarias para su correcto funcionamiento.

Las terminales en nuestro SO son procesos comunes y corrientes iguales a todos los demás procesos del sistema operativo. La principal diferencia es que las mismas no tienen un *parent* y que ellas son *parent* de todos los procesos que en ellas se ejecutan.

A continuación mostramos la declaración de las estructuras antes mencionadas:

```
typedef struct tty_t{
        int ttyNumber;
        ttyScreen_t * screen;
        keyboard_t * keyboard;
        shellLine_t * lineBuffer;
} tty_t;
typedef struct ttyScreen_t{
```

```
        int wpos;
        char buffer[SCREEN_COLS*SCREEN_ROWS*2];
} ttyScreen_t;
 typedef struct keyboard_t{
        int head;
        int tail;
        int shift_state;
        int alt_state;
        int ctrl_state;
        int caps_state;
        int escaped_key;
        int dead_key;
        int num_state;
        int scroll_state;
        int lang;
        char buffer[K_BUFFER_SIZE];
} keyboard_t;
```

Shell

Para implementar un proceso Shell para el usuario se creó una funcion shellLoop() que llama a la función shell() en un ciclo infinito. Ésta función a su vez imprime un prompt e inicia un ciclo en el cual recibe teclas del teclado imprimiendolas en pantalla y guardándolas en un buffer hasta que el usuario presiona Enter. Entonces el buffer es parseado, eliminándose los espacios y separando el comando de sus parámetros, los cuales se guardan en una estructura con la siguiente forma:

```
static struct {
        char name[LINEBUF_LEN];
        char args[LINEBUF_LEN - 2];
} command;
```

Luego se compara el comando con una lista de comandos válidos, y si es uno de esos comandos lo ejecuta. La información de los comandos y los punteros a sus respectivas funciones son guardados en una estructura definida de la siguiente manera:

```
static struct {
   char* name;
   char* description;
   commandFnct exec;
} commands[NUM_COMMANDS];
```

Es importante destacar que los "comandos" en este SO crearán procesos. Por ejemplo, al ejecutar el comando *top* lo que se hará es crear un nuevo proceso "TOP" cuyo padre será el proceso "SHELL" en el que fue ejecutado y cuya prioridad será igual a la de esta última.

Luego se borra el buffer de teclado y se sale de la función, la cual es llamada de nuevo desde shellLoop()

Mouse y Timer Tick

La consiga del trabajo práctico especial especificaba que se debía programar a éste “sistema operativo” para que, al recibir órdenes de los botones del mouse, se generen más o menos interrupciones por segundo en la IRQ #0.

Más específicamente, al presionar botón derecho, se debía aumentar la frecuencia de dichas interrupciones y al presionar el botón izquierdo, se debían decrementar.

Lo que se realizó fue programar un handler para el mouse que recibía los 3 paquetes que éste envía. De todas formas, el paquete de interés para éste trabajo en particular era el primero de ellos ya que no nos interesaba disparar ninguna acción con el desplazamiento del mouse y/o el scroll. El handler tomaba el paquete recibido y se fijaba qué bits tenía encendidos. En base a ésto y luego de aplicarle una máscara, se llamaba a una función que incrementaba o decrementaba el Timer Tick, según el botón que se haya presionado.

El PIT es un chip conectado a la IRQ #0 cuyo objetivo es interrumpir a la CPU con una frecuencia definida por el usuario (entre 18.2Hz y 1.1931 Mhz). Él PIT es el método primario utilizado para implementar el clock del sistema y es el único método disponible para implementar multitasking (cambiar de proceso cuando existe una nueva interrupción).

Este chip tiene un clock interno que oscila a 1.1931 Mhz; ésta señal de clock está alimentada por un divisor de frecuencia, para modular la frecuencia de salida final. Tiene 3 canales, cada uno de ellos con su propio divisor de frecuencia.

En nuestro caso, el único canal que nos interesa es el Channel 0 y es el que su salida está conectada al IRQ #0. Para lograr éste cambio de frecuencia pedido en la consigna, lo que se debe hacer es reprogramar el *Programmable Internal Timer.* Simplemente se le envía un byte por uno de sus puertos (el puerto de comando) para informarle que uno desea reprogramarlo y luego se le manda un divisor de

la frecuencia que se desea obtener.

Una peculiaridad de ésta reprogramación es que se le debe enviar al PIT la parte ALTA y la parte BAJA de la frecuencia que se desea obtener, de forma separada. Por este motivo, la función que realiza ésta operación calcula en un entero la frecuencia total deseada y luego hace operaciones a nivel bit para separar ese número en parte alta y parte baja.

```
void cambiar_frecuencia(int frecuencia){
        int divisor = 1193180 / frecuencia;
        // Se envía el bit de comando
        _out(0x43,0x36);
        // Se hace un split del divisor en parte alta y parte baja
        char parteBaja = (char) divisor & 0xFF;
        char parteAlta = (char) ( (divisor >> 8) & 0xFF);
        _out(0x40, parteBaja);
        _out(0x40, parteAlta);

}
```

Excepciones

En nuestro kernel, hemos implementado un handler para las posibles excepciones que pudiesen surgir durante la ejecución del programa. A su vez, hemos implementado 2 funciones de prueba para hacer el testeo del correcto funcionamiento de dicho handler.
Éstas funciones son **divideByZero** y **invOpcode**. Lo que hace la primera es intentar realizar una división por cero, ejemplo 1/0; por su parte la segunda, intenta ejecutar una instrucción de assembler que no existe.

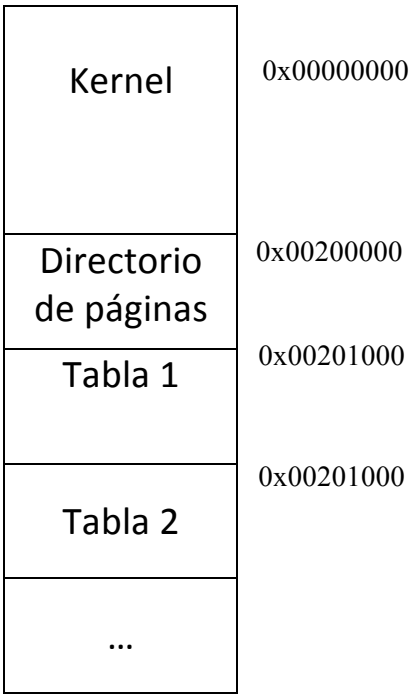
Al correr alguno de dichos comandos sobre nuestra implementación, el sistema quedará colgado y no se podrá volver a ejecutar ningún comando hasta que éste no sea reiniciado. Ésto sucede debido a que, para poder recuperarnos de la excepción, lo que deberíamos hacer es incrementar el **instruction pointer** hacia la próxima instrucción a ejecutar. Lo que sucede es que, para realizar ésta operación, deberíamos saber la longitud de la próxima instrucción. Intel contiene instrucciones de tamaño variable (a diferencia de ARM), por lo que habría que realizar un parseo de la longitud de la próxima instrucción y recién luego de ésto, calular la próxima dirección de memoria a la que deberíamos saltar.

Memory manager

Paginación

Se decidio implementar un mecanismo de un manejador de memoria que sea independiente de la memoria que se dispone.
Eso se logró haciendo una función que reciba como parametro la ultima dirección de la memoria, entonces se divide por el tamaño de una pagina (4K) y a partir de eso se calcula la cantidad de paginas que se dispone para todo el sistema.
A partir de esas paginas se calcula cuantos descriptores tendrá el directorio de paginas que será la misma cantidad de tablas de página y se realizo una paginación uno a uno.

Por el momento la memoria queda de la siguiente forma:



Luego se calcula la primera dirección después de la última tabla y a partir de ahí se destina a memoria para el usuario.

En cuanto al manejador de memoria, se implemento una lista de “element”que contienen “mem_headers” que son estructuras que describen a una pagina.

En principio, por eficiencia, se tiene una lista con un único header que describe a la primer página del sector del usuario y a medida que se van necesitando páginas, se itera por esa lista, y si todos los

headers describen a paginas llenas, se agrega un header mas.

La estructura para la lista es la siguiente:

```
typedef struct llist {
    int NumEl;   // Número de elementos en la lista
    element *  pFirst;
    element *  pLast;
    void * first_page;
    int max_pages;
} llist;

typedef llist * List;
```

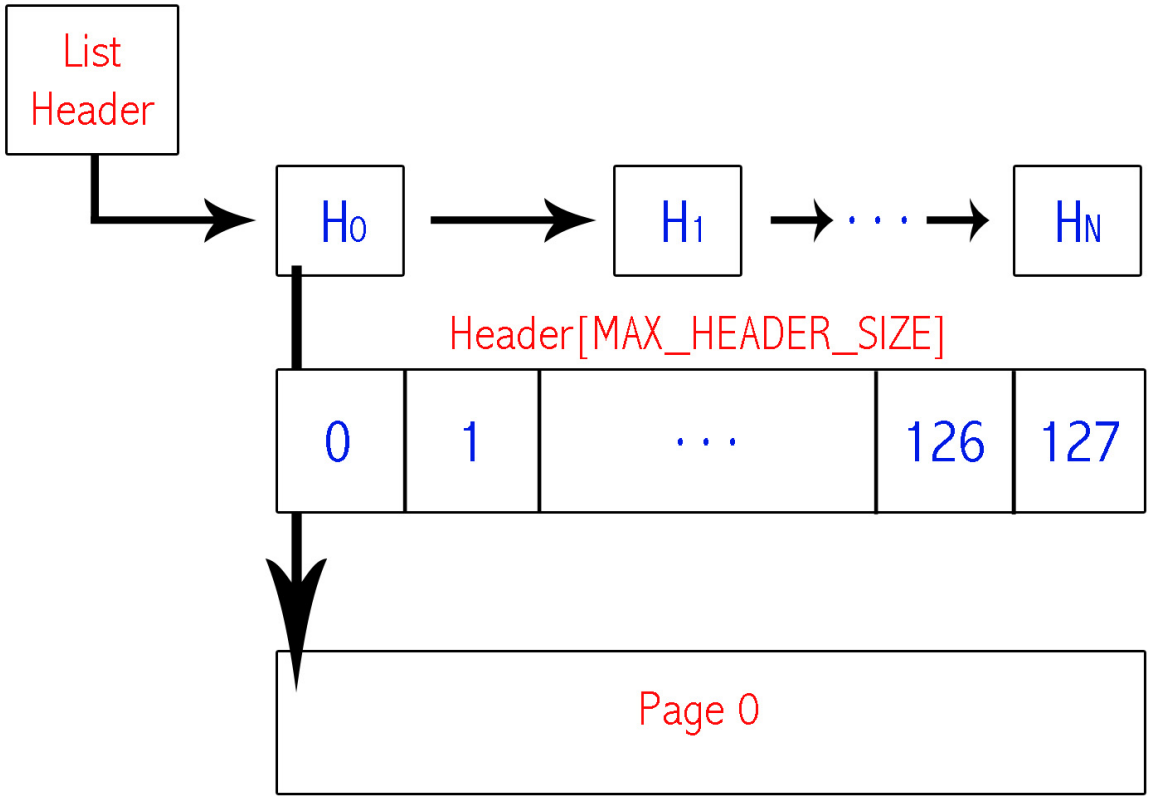
La estructura de un nodo es la siguiente:

```
typedef struct element{
    struct element *  next; // Puntero al nodo previo
    struct element *  prev; // Puntero al siguiente nodo
    mem_header headerEl; // Header que describe una página
     } element;

typedef element * Element;
```

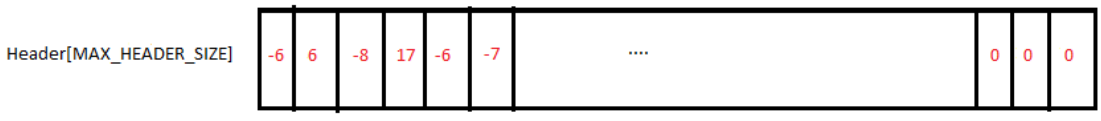
Y por último el header que describe a una pagina es una estructura de la forma:

```
typedef struct mem_header{

     char header[MAX_HEADER_SIZE];
     unsigned char blocks_cont; //  Cantidad minima contigua de bloques,
al principio 128
     int pid;
} mem_header;
```

Donde MAX_HEADER_SIZE 128.



En la imagen se puede observar claramente como una página está vinculada con la lista de páginas. La relación que hay entre una página y el header es más compleja.
Cada header está dividido en 128 bloques de 32 bytes, lo que da un direccionamiento de hasta 4096bytes lo que equivaldría a una página entera.
Dentro del header se almacenará un número positivo, negativo o cero. Un cero implicaría que es el último bloque libre del header.
Un número negativo que se refiere a la cantidad de bloques de 32 bytes que se reservaron y un número positivo indican una cantidad contigua de bloques que fueron liberados. Por lo tanto se podrá encontrar un header de la siguiente manera.



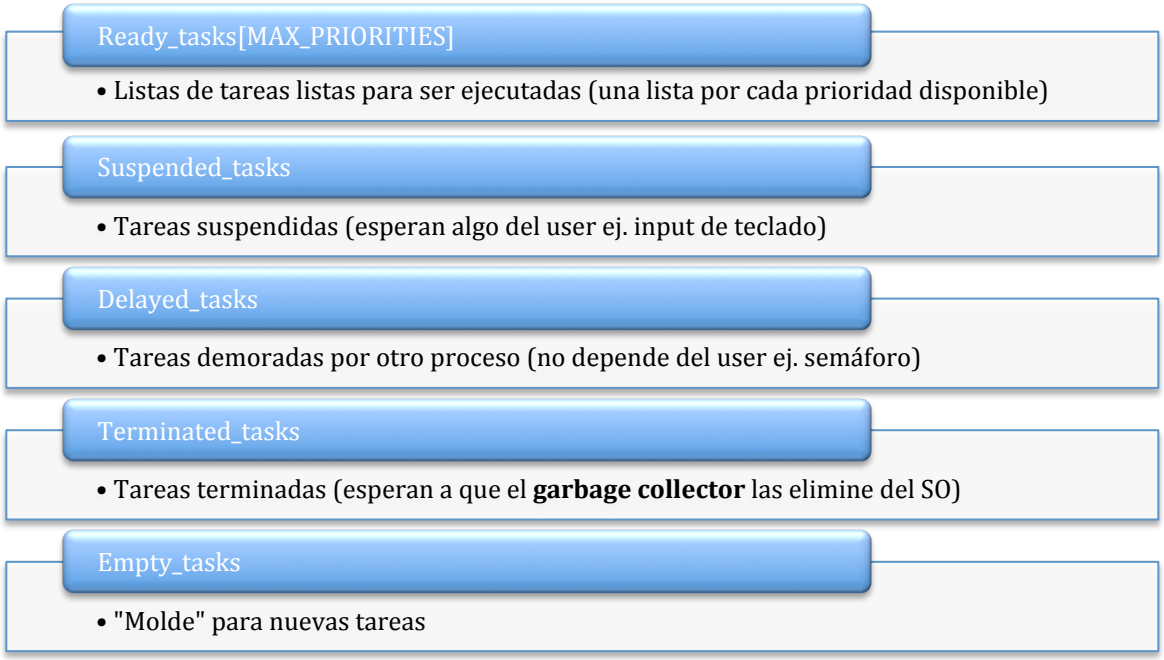
Si se realiza un malloc podrán suceder 3 casos:

- El primer caso consiste en que ninguno de los espacios previamente liberados posea memoria suficiente para alocar el nuevo puntero, en tal caso se cambiará el último 0 por tal número negativo.
- El segundo caso consiste en que existe un bloque previamente liberado de igual tamaño del que se quiere reservar, en ese caso, se cambia el signo de ese número quedando como negativo.
- El último caso, es que haya un bloque previamente liberado de tamaño mayor al que se quiere reservar y en tal caso se genera una nueva partición (con un numero positivo) del tamaño restante. Si se realiza un free en caso de que el bloque anterior y/o el posterior estén liberados, se genera un nuevo bloque de tamaño igual a la suma de los 3. En caso de que el bloque a liberar sea el último, se lo reemplaza por un 0 indicando un nuevo final de header. Si se realiza un realloc se tratará de reservar una nueva posición de memoria fijándose si entra en una zona nueva y en caso de no entrar se fija si el bloque anterior y posterior están libres y la suma es mayor o igual al tamaño pedido, en cualquiera de estos casos se copia la información a la zona nueva.

Procesos

Los procesos estan modelados mediante la estructura *Task_t* que se detallará más adelante.

El algoritmo principal de *scheduling* implementado es muy similar al implementado por Tanenbaum en **Minix** y utiliza el concepto de distintas colas y prioridades.



Según la cola en la que las tareas se encuentren el *scheduler* les cederá o no el procesador. Básicamente el algoritmo le cederá el procesador a la próxima tarea que esté lista para ejecutar y tenga la mayor prioridad para ese entonces. Existe un caso en el que una tarea de mayor prioridad puede no "ganar" el procesador y este se da cuando la tarea que está corriendo actualmente está en **modo atómico**. Si esto sucede, la misma podrá monopolizar el procesador hasta el fin de su ejecución.

Task_t

La estructura "Task_t" guarda toda la información de un proceso:

```
struct task {
      char              name[PROCNAME_MAX];
      TaskState_t       state;
      Prio_t            priority;
      uint              pid;
      unsigned          atomic_level;
      char *            stack;
      void *            ss;
      void *            sp;
      Task_t *          parent;
      TaskQueue_t *     queue;
      Task_t *          prev;
      Task_t *          next;
      bool              foreground;
      bool              success;
      bool              in_time_q;
      Task_t *          time_prev;
      Task_t *          time_next;
      Time_t            ticks;
      int               tty_number;
      ttyScreen_t *     screen;
      keyboard_t *      keyboard;
      shellLine_t *     linebuffer;
      void *            msg;
      unsigned          size;
      void *            descriptor_table[STREAM_MAX];
      registers_t       registers;
      TaskQueue_t       send_queue;
};
```

Mas allá de la complejidad de la misma, las variables más interesantes para destacar son:

- **name:** Guarda el nombre de la tarea (por ejemplo para ser impreso por "TOP")
- **state:** Es el estado actual de la tarea, el mismo puede ser:

    - TaskSuspended
    - TaskReady
    - TaskCurrent
    - TaskDelaying
    - TaskWaiting
    - TaskTerminated
    - TaskNULL
    - TaskEmpty
- **ss** y **sp:** Direcciones de memoria del segmento de stack y el puntero al stack respectivamente
- **screen**, **keyboard** y **linebuffer**: Buffers para la pantalla, el teclado y la linea de input respectivamente. Este último se usa, por ejemplo, para el autocomplete. Nótese que al crear un poroceso estos buffers son heredados cual file-descriptors
- **priority:** Prioridad del proceso
- **tty_number:** Tty en la que el proceso está ejecutándose
- **foreground:** Flag que indica si el proceso corre en foreground o background

Stackframe

El stack de una tarea simplemente guarda, en este orden las siguientes cosas:

- Registros de uso general
- CS
- IP
- EFLAGS
- Función de retorno

Es interesante destacar que la "función de retorno" es utilizada como una función de limpieza. Al ser creada, cada tarea alojará en su stackframe una función cleaner que hará que cuando la tarea finalice, ésta pueda liberar su stack y borrarse del vector del procesos.

Nacimiento de un proceso

Los procesos nacen al llamar a la función **CreateProcess**. El prototipo de la misma es el siguiente:

```
Task_t * CreateProcess(char* name, PROCESS process, Task_t * parent, int tty,
                       int argc, char** argv, void * stack_start,
                       int priority, int isFront);
```

Básicamente recibe un nombre, un proceso (lo que hará), un padre, una tty en la que ejecutará, la dirección en la que comienza su stack, una prioridad y una variable booleana que declara si corre en foreground o en background.

Cosas a destacar:

Si el padre es NULL, entonces se reserva memoria para los buffers de teclado y pantalla de la tarea; caso contrario, estos buffers son heredados.

Una vez finalizada su creación, la tarea es agregada a la cola *TaskReady* en la que se encuentran las tareas listas para ejecutar.

Muerte de un proceso

Matar una tarea es bien sencillo: ésta debe ser sacada de la cola de tareas listas para ejecutar y movida a la cola de tareas finalizadas (**Terminated_tasks**). Una vez allí, cuando el procesos nulo gana el control del procesador, este iterará por cada una de las tareas terminadas y liberará su stack, como así también las borrará de la lista de procesos, para así darle lugar a la creación nuevas tareas en su lugar.

Cambio de contexto

El cambio de contexto sucede ante cada interrupción del timer tick (INT 20). Al producirse ésta, su manejador llama a la función *select_next()* la cual selecciona a la próxima tarea a la que se le entregará el procesador. Para hacer esto, se hace apuntar el *esp* al stack de la tarea que se desee cargar y se levantan del mismo todos los registros de uso general. Finalmente se hace un *iret* que hará apuntar el *ip* y el *cs* a los lugares correspondientes como así también restaurará los flags.
En el caso de que la tarea haya terminado, como se mencionó anteriormente, se levantará del stack la función cleaner que hará todo lo necesario para liberar el stack, eliminar la tarea de la lista de procesos y demás.

Es interesante destacar que en el caso del scheduling con distintas colas y prioridades, la función *select_next()* hace todas las validaciones pertinentes para elegir de manera correcta a la próxima tarea a ejecutar. Por ejemplo se fija si la tarea actual está corriendo de forma atómica, se fija si existen tareas de mayor prioridad que ella para cederles el procesador, chequea si es la única tarea, etc.

Si es que no existe ninguna tarea que esté lista para ejecutar al momento de la interrupción del timer tick, *select_next()* le cederá el procesador a la **NullTask**. Esta tarea se ejecutará con la prioridad mínima y estará consumiendo ciclos de procesador mientras no haya otra cosa que hacer. Como se explicó anteriormente la tarea nula no sólo consume ciclos y es una tarea "boba" sino que tiene una importante función como **garbage collector**.

Kernel

Se pensó en un principio en un microkernel basado en el Sistema MINIX de Tanenbaum. Sin embargo la complejidad resultante de dicha tarea hizo que resultara más simple hacer un kernel monolítico altamente modularizado, que un sistema de tareas y procesos individuales que administran la PC en forma cooperativa.

Por consiguiente, se puede ver el uso del Memory Manager y el Scheduler en forma conjunta para servir a los procesos que se sustentan en ellos.

Scheduler

En líneas generales el algortimo de scheduling propuesto es un sistema de multiples colas, para la tabla de procesos mostrados previamente. Existen en total 11 colas entre las cuales los procesos se pasean. Dichas colas son:

- 1 cola de Procesos Suspendidos.
- 5 colas de Procesos Listos para ejecutarse.
- 1 cola de Procesos Demorados.
- 1 cola de Procesos Esperando.
- 1 cola de Procesos Terminado.
- 1 cola de Procesos Vacíos.

En principio, todos los procesos se encuentran en la cola de Procesos Vacíos. Estas colas funcionan en forma de FIFOs, siendo el primero que entra, el primero en salir. A su vez, las 5 colas de Procesos Listos sirven para distintas prioridades. De tal forma que existen 5 prioridades de ejecución de comandos, y en dichas prioridades, por la forma de actuar de la cola, el comportamiento es simil a un algoritmo de Round Robin.
Cabe destacar que el proceso que se encuentra ejecutándose en ese momento NO se encuentra en ninguna cola. Es decir, es un proceso distinguido. Esto puede facilitar a futuro la creación de IPCs complejos de manera simple.
A su vez, existe una lista de procesos en primer plano corriendo. Esta lista es del mismo tamaño que la cantidad total de TTY en el sistema. Funciona de manera de ventana. Donde se puede acceder a dicha lista por medio de una variable que mira cada parte de la lista de procesos en primer plano, según el deseo del usuario. De esta manera, los procesos pueden intercambiar facilmente el acaparar los recursos provistos a cada TTY, sin importarle el usuario, y el usuario se abstrae totalmente del funcionamiento interno del sistema de disposición de recursos.
A su vez, el proceso nulo también es un proceso distinguido. Esto permite que no sea necesario ocupar nunca un lugar en el planeo del scheduler.

Conclusiones

Trabajo base

Dado que la consigna del trabajo establecía retomar un desarrollo previo de otra materia (Arquitectura de Computadoras), parte de la complejidad del presente trabajo fue debido a que los integrantes del grupo habíamos realizado distintos trabajos prácticos en distintos cuatrimestres. Se decidió hacer un *merge* entre las soluciones de los diferentes trabajos dando como resultado el producto final.

Esto nos sirvió para entender la importancia de la buena documentación del código como así también el buen estilo y modularización del mismo.

Assembler

Si bien la realización del trabajo no fue sencilla, y volver a programar en bajo nivel luego de programar en Java fue algo extraño, al programar en assembler uno tiene control total sobre el micro procesador y puede realizar tareas que con otros lenguajes resulta imposible. Además al realizar el trabajo se pudo aprender mucho del funcionamiento interno de la PC y de su interacción con los dispositivos más importantes de I/O, como así también fijar el concepto de sistemas multi-tarea y las nociones de scheduling.

Conceptos aprendidos

Una vez finalizado el trabajo entendimos qué era realmente un sistema operativo básico multitarea y la complejidad de su desarroll. Asimismo comprendimos las ventajas de un sistema con memoria paginada, y la seguridad que brinda esta característica. Aunque lograr un eficiente y seguro manejo de memoria es muy complejo.

Nos resultó algo dificultosa la tarea de *debugging* en el proceso de desarrollo debido a que no encontramos herramientas que la faciliten. Una de las problemáticas de la programación a tan bajo nivel es que, por ejemplo, ante un herror la máquina virtual (bochs) automáticamente se cuelga y/o se reinicia. Este error puede ser tan "sencillo" como estar escribiendo en una página ausente o estar desreferenciando un puntero mal asignado. La estrategia que encontramos para subsanar esta problemática fue la implementación de una función *kprintf()* la cual funciona de manera similar a printf() pero escribe directamente en video (0xB8000); es decir que no se fija ni tiene en cuenta las TTYs, los buffers, los procesos y demás sino que directamente imprime en pantalla. De esta forma podíamos ir imprimiendo los valores de las variables y/o los registros e ir evaluando por qué es que las cosas no se ejecutaban de la forma esperada.
A su vez, otra estrategia utilizada fue la de crear una función *halt* la cual explícitamente ejecutaba la instrucción *hlt* del procesador y hacia por ende que este detenga su ejecución. La técnica consitía en ir

"bajando" la línea del halt hasta encontrar la línea de código que producía que la máquina virtual se reiniciara. Una vez encontrada esta línea se analizaba el problema puntual y se seguía desplazando la línea de halt hacia adelante hasta encontrar un posible nuevo foco de conflicto.