Flying-High OS

Magni Nicolás, Purita Nicolás, Zemin Luciano R.

May 19, 2010

Contents

1 Introducción

1.1 Objetivo

Se debe crear un sistema operativo multitarea, asignándole a cada proceso un tiempo de ejecución. El multitasker corre directamente sobre memoria y montado sobre una base de un sistema monotarea. Para cargar el Sistema Operativo se utiliza el bootloader GRUB de Unix.

1.2 Enunciado

Se desea implementar un Multitasker, cuyo objetivo principal es el de asignar tiempos de ejecuci ón a diferentes procesos en memoria. El sistema deberá ser implementado para plataformas Intel de 32 bits, utilizando el procesador en modo protegido. El multitasker debera ser preemptivo, es decir, cualquier tarea puede ser desalojada del microprocesador. El encargado de administrar el CPU es el scheduler el cual tomará como base de tiempo la interrupción de hardware INT8 correspondiente al timer tick, para realizar la asignación de tiempo (time slot).

1.3 Actividades

1. Se deberá elegir la forma en que se resguardará el contexto de cada tarea, y se deberá elejir entre las siguientes opciones
   - Utilizar los TSS que provee el microprocesador Intel 386
   - Realizar una implementación propia de código.
2. Se deberán implementar dos tipos de scheduling distintos. Principalmente uno de ellos deberá considerar la prioridad de los procesos para asignar los slots de tiempo.
3. El sistema deberá estar programado de manera que se diferencien los estados básicos de Corriendo, Esperando y Listo. Por otra parte, cada proceso deber á tener un valor de prioridad entre 0 y 4 que indique la importancia del proceso. Además se deberá demostrar el funcionamiento de los mismos con programas de prueba y se deberá poder corroborar el estado del proceso y el porcentaje de procesador que está ocupando con la ayuda del comando top. También deber á existir un comando kill que permita matar procesos en ejecución. Tener en cuenta que kill debe matar también a todos los hijos de ese proceso.
4. Se deberá poder tener al menos 4 terminales distintas y alternar entre ellas de manera similar a Linux.
5. Se deberán poder ejecutar diferentes tareas a través de comandos ingresados por teclado. La sintaxis de los comandos quedan a elección del desarrollador.
6. El sistema debe tener la posibilidad de correr los mismos procesos tanto en foreground como en background. Para este último se deberá utilizar el caracter & al igual que en UNIX.

7. El sistema debe tener un módulo de administración de memoria mediante paginación para los procesos, el mismo se encargará de lo siguiente:

   - Cada proceso tendrá su stack propio en una página, a la cual solamente él tendrá acceso. Cada proceso podrá leer y escribir libremente sobre esta página pero no páginas de otros procesos.
   - Los procesos no poseerán un heap propio, ya que están corriendo sobre la misma zona de datos del SO.
   - Ninguń proceso deberá leer o escribir directamente ninguna variable global del SO. En caso de que haya variables globales que estén pensadas para ser leidas por procesos usuario, deberán tener una función que las copie a una zona de heap propia al proceso, simulando un system call.

2 Material entregado

Se entrega un archivo comprimido que contiene el directorio raíz del proyecto. El mismo está formado mediante la estructura básica de un proyecto, a saber:

- inc, con todos los headers
- src, con el código fuente
- bin, directorio de salida del kernel luego de su compilación
- img, directorio en donde se encuentra la imǵen de diskette que se utilizará para correr el sistema operativo, cuyo kernel se actualiza con el compilado mediante un comando en consola
- doc, que incluye toda la documentación del proyecto, en formato doxygen, incluído este informe También se proveen los archivos makefile, para la compilación, bochsrc, para la configuración del bochs, y mtools, para la configuración de esta utilidad.

3 Kernel

3.1 Objetivo

El *Kernel* es el encargado de levantar todo el sistema operativo.

3.2 Esquema general

La función principal del *Kernel* es cargar en la IDT las rutinas de atención del teclado, del timer tick y las interrupciones de la *int 80*, iniciar la paginación, iniciar el multitasker e inicializar las TTYs. Ademas de todas éstas inicializaciones, tambien inicia el driver de video, el módulo de shared memory, los semáforos y por último habilita las entradas del Pic que correspondan al Timer Tick y al Teclado.

3.2.1 Llamadas a Sistema

Para inicializar cada módulo se realiza una llamada a sistema, ya que en ese instante en que se esta inicializando no puede perder el procesador, por lo tanto todas estas funciones deshabilitan antes las interrupciones. Toda función que hayamos considerado que no puede perder la atención realiza una llamada a sistema.

4 Paginación

4.1 Objetivo

El sistema operativo debe tener un módulo de paginación, por lo tanto cada proceso debe tener asociada cierta cantidad de páginas. Por consecuente se debe implementar el manejo de la excepción correspondiente a paginación ya que el módulo de administración de memoria debe verificar que un proceso no utilize páginas no asignadas a él. Todos los procesos comparten el heap, es decir que no existen variables globales dentro del sistema operativo, y en el caso que existiesen, deben estar en el heap del kernel, por lo tanto si el proceso desea obtener algun valor de alguna variable debe simular un system call y copiar la variables al heap propio del proceso.

4.2 Esquema general

Como se implementó un módulo de administración de memoria, se debió implemantar un *malloc*. Un criterio tomado es que el kernel llama una sola vez al memory map donde obtiene todo su espacio kernel. En cambio el malloc llama reiteradas veces al memory map donde se asignan las páginas asociadas al heap de ese proceso. Toda la información de las páginas asignadas se encuentran en la tabla de procesos.

4.3 Problemas y soluciones

Un gran problema que obtuvimos por medio de la paginación fue que no podiamos crear mas de 4 procesos en simultáneo, esto se produjo ya que luego de ver reiteradas veces el código y no darnos cuenta de cuál era la causa del Page Fault, decidimos seguirlo desde código Assembler donde pudimos encontrar la razón por la que se lanzaba la excepción y era porque el **CR2** tenía carada una dirección de una página invalida, que por lo tanto no estaba presente y lanzaba Page Fault. El principal problema fue en el armado de los frames de las páginas, donde el algoritmo no contemplaba un caso donde había que dar de baja un frame, levantar otro y asi consecutivamente. Luego de una gran reestructuración de nuestro sistema, tambień nos dimos cuenta de que se nos escapó asignar los nuevos valores definidos en *defs.h* para llevar a cabo la paginación, y por lo tanto retornaba direcciones no deseadas. Este problema se detallará al final del informe.

4.4 Limitaciones

Como no se utilizó segmentación de páginas, el malloc siempre retorna páginas contiguas y no se almacena ningun registro sobre segmentos otorgados.

5 File System

5.1 Modelado

Se implmentó una simulación de un File System, pero unicamente teniéndose los pseudoarchivos *STDIN* y *STDOUT*. Cada proceso tiene almacenado en su estructura estos pseudoarchivos. Cabe destacar que nuestro sistema operativo utiliza una estructura FILE para el sistema de archivos, la cual es muy similar a la de UNIX.

5.2 Esquema general

Al tener cada proceso sus archivos de entrada y salida, resultó fácil anexar los mismos a las ttys, que serán explicadas luego. Así, ante un cambio de contexto a otro proceso, o ante un cambio de foco de tty, los procesos no pierden su posibilidad de utilizar estos archivos, a menos que realmente no deban realizarlo, en cuyo caso el sistema operativo se encarga de manejarlo.

5.3 Problemas y soluciones

Inicialmente, los procesos no tenían un filesystem propio, sino que compartían un filesystem que poseía cada tty. Esto trajo muchísimos problemas en cuanto a manejo de la salida y entrada estándard de los procesos. Afortunadamente se resolvió implementarlo de la forma antes explicada, y esto trajo muchas soluciones.

5.4 Limitaciones

Si bien la limitación más obvia es que no se posee un filesystem propiamente dicho, sino que solo se tienen archivos de entrada y salida estándard para cada proceso, este pseudo filesystem es fácilmente extendible a, quizás, un filesystem completo, partiendo probablemente de un archivo de salida de error, y luego un sistema de archivos propiamente dicho.

6 TTY

6.1 Objetivo

Dado un numero maximo de tty's, estas son creadas en la inicialización del sistema. Estas almacenan información que sólo es modificada por la TTY, ya que esta es la encargada de traducir los distintos lenguajes que se manejan en el sistema.

6.2 Modelo

Al comienzo del desarrollo del proyecto, no estaba claro el concepto de TTY, y por esta razón se decidió investigar sobre el mismo. Se decidió seguir, a grandes rasgos, el modelo de UNIX, con sus respectivos ajustes. Al iniciar el sistema ,se crea una cantidad fija de tty's, allí se almacena toda la información necesaria para el buen funcionamiento, y luego es seteada la tty en foco, que será modificada ante cualquier entrada o salida de datos. A cada tty se le asocia un proceso corriendo en ella, como así también son almacenados buffers de entrada y salida de datos. Respecto a los demas campos que posee su estructura, estos son utilizados para un menejo correcto de los buffers. Los mismos solamente son accedidos por la tty. En el stdin se pueden encontrar todos los caracteres ingresados por el teclado, y serán almacenados en la tty en foco. Respecto al stdout de la tty, es un buffer circular de tamaño igual al de la pantalla, y se mantiene el scroll de la pantalla, mateniendo el estado actual de la misma. En este la información almaceanda es en lenguaje shell. Se hizo de esa forma, para que sea más sencillo parsear el stdout, en el caso de querer redireccionarlo. El comportamiento de la tty puede variar dependiendo del tipo de comunicación que el proceso puede interpretar. En el modo **Canónico**, se interpretan los caracteres de control y se llama a la función que maneja el caracter ingresado(handler), si no es un caracter de control se lo almacena en el buffer interno. Al momento de interpretar que el usuario a presionado un **enter**, todos los caracteres ingresados son colocados en el stdin del proceso que se encuentra en foco en la tty en foco. En el modo **Raw**, todo lo ingresado es colocado en el stdin del proceso, los únicos caracteres que son parseados son los **F1, F2, ...**, ya que es la única manera de poder cambiar entre tty's. Todo pasaje de información del buffer de la tty hacia los stdin de los procesos se realiza mediante las primitivas write y read, para lograr así mantener la circularidad de los buffers del file system.

6.3 Esquema general

En esta sección se explicara el flujo de información dentro del sistema. Se comenzará en modo **Raw**, un proceso intenta leer de su stdin algún caracter ingresado, como el bufer se encuentra vacío, se duerme. Una vez que el usuario preciona una tecla, la misma es alamacenada en el buffer del teclado, el driver de teclado coloca el caracter ingresado en el stdin de la tty, esta toma el carcter, lo procesa, y si es un caracter de control se ejecuta la función asociada al mismo caracter, sino se lo coloca en el stdin de la tty que se encuentra en foco, y se procede a despertar el proceso dormido en la tty si es necesario.
En el modo **Canónico**, a diferencia de lo explicado anteriormente, los datos son actualizados al sdtin del proceso una vez que se halla ingresado un **new line**

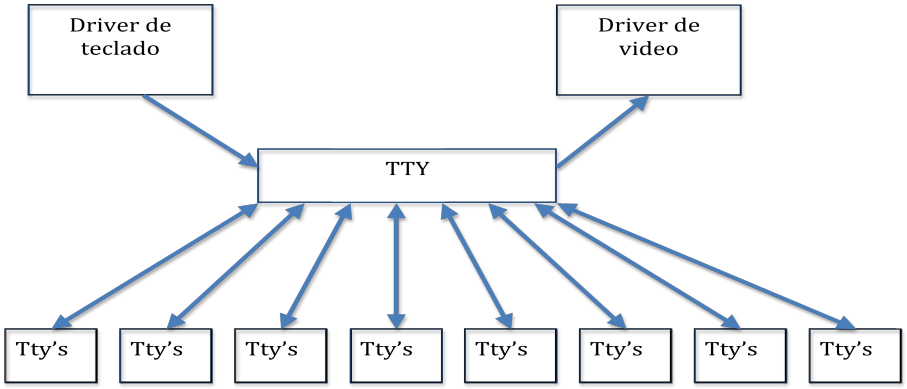

Figure 1: Flujo de datos entre los driver, y la tty.

Si un proceso quisiera escribir en pantalla, se realizan los siguientes pasos. En modo **Canónico**, el proceso es el encargado mediante la primitiva write de colocar los caracteres en el sdtout que le pertenece. En modo **Raw** se utiliza una api encargada de colocar en pantalla los caracteres, esta api ha sido diseñada inspirada en **ncurses**. Ambos sdtouts son actualizados en cada interrupción del timer tick, al igual que la pantalla.

En la figura 1 se muestra el flujo de datos, entre los drivers y la tty. En la figura 2 se muestra el flujo de datos entre los procesos y la tty.

6.4 Problemas y soluciones

Se nos presentaron muchos problemas con la TTY ya que a medida que se iba leyendo caracteres o preguntando algo, nos dábamos cuenta de que teníamos que mejorarlo, y, por lo tanto, las TTYs sufrieron infinidad de modificaciones desde el comienzo del desarrollo, desde su inexistencia, hasta convertirse en un sistema realmente avanzado en lo que nos respecta.
El primer problema que nos surgío fue el de los lenguajes que manejaba la TTY, relacionado con sus buffers. En un principio teníamos un único *STDIN*, por lo que tuvimos que cambiarlo y asignarles un buffer *STDIN* a cada TTY.
Otro problema con ésto fué la diferenciación entre el *STDIN* y *stdin*, donde para nosotros el *stdin* es la entrada estándar del proceso que estaba corriendo y no necesariamente es el que está en foco en la TTY actual, menos aún podría serlo si fuera una shell que se encuentra dormida. Por lo tanto se tuvieron que modificar los write y read para que escriban sobre un *stdin* y *stdout* indicado, implementándose para ello las funciones fread y fwrite. Cabe destacar que nos resultó bastante complicado darnos cuenta de que ante un pasaje de datos desde la tty hacia el proceso correspondiente que colgaba de la misma, en realidad se le estaba pasando la información al proceso que estuviera corriendo al momento

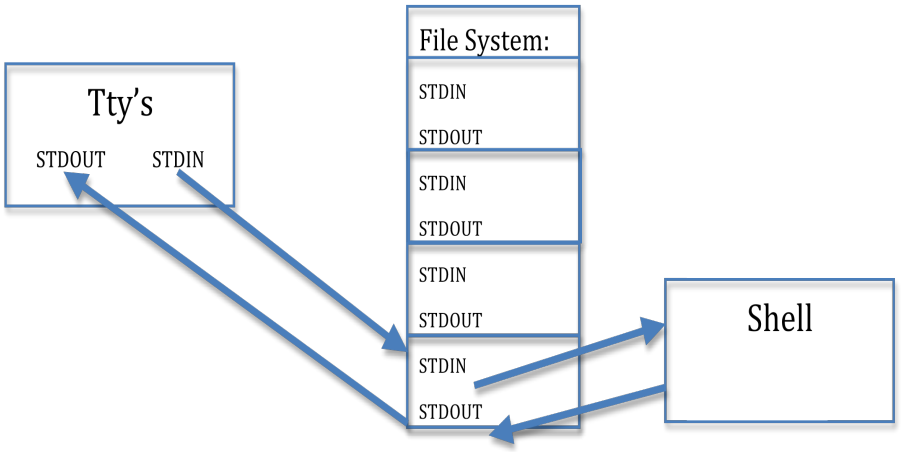

Figure 2: Flugo de datos entre los procesos, la tty y el stdout

que el timer tick interrumpía, y a su vez resolver escribir en el proceso que se encontraba en foco en la tty en foco, valga la redundancia. Este problema surgía ante un enter, ya que la linea que la tty acababa de procesar debía ser entregada al proceso que correspondiera, el cual sería el proceso en foco en la tty en foco, como se dijo antes. El problema era que en la funcion que se encargaba de eso, se utilizaba un write en STDIN, y para nosotros, stdin es el stdin del proceso que está corriendo, que no necesariamente es el que esta en foco en la tty actual. Menos aún podría serlo si es una shell que está dormida. Por ese motivo, se tuvo que implementar una función que devolviera el stdin del proceso en foco en la tty en foco para asi poder darle la informacion, mediante las mencionadas fread y fwrite. Un problema de no gran envergadura fue el manejo de los índices, ya que en un principio se utilizaron dos variables de desplazamiento únicamente, y se incrementaba de a un paso o en el caso que fuese un caracter de control se desplazaba lo indicado por ese caracter, pero por cuestiones de simplicidad se decidió utilizar una varible que indica la cantidad de caracteres que escribió en el *STDOUT* de la TTY, una variable que dice la fila en la cual se encuentra y una variable para la columna. Del mismo modo para las variables de lectura, se continuó el mismo criterio.

6.5 Limitaciones

Se tuvieron que tomar ciertas consideraciones para poder desarrollar el sistema, una de ellas es una limitada catidad de caracteres que se pueden almacenar en modo Raw, debido a que si no se presiona **new line**, el buffer de la tty no es refresacado, por esa razon se perderían los caracteres ingresados una vez que el buffer esté completo.

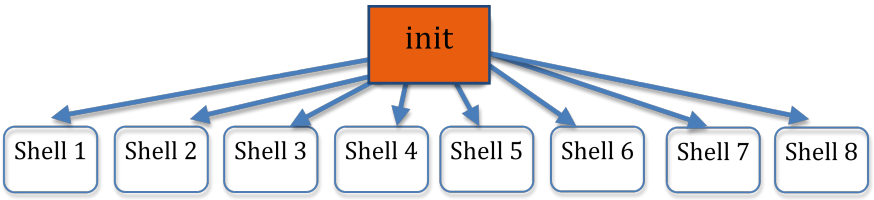

Figure 3: Estado inicial del sistema, los procesos que están en azul están bloqueados, y los que están en naranja esperan a que terminen sus hijos. El sentido de las flechas representa a quién se está esperando

7 Procesos

7.1 Objetivo

Se lanzarán inicialmente nueve procesos, y serán creados en el siguiente orden, init, shell1, shell2, ... ,shell8. En la figura 3, se presenta un diagrama de arbol, demostrando la relación entre ellos. Al comenzar a correr init, este crea las 8 shells y se dueme en espera del retorno de sus hijos, tomando el procesador únicamente cuando no existe otro proceso en condiciones de tomarlo, como por ejemplo, si solo se encuentran creadas las 8 shells que se encuentran bloqueadas esperando datos de entrada.

7.2 Modelado

A diferecncia de linux, los procesos corren el mismo archivo binario, por esa razon se debe tener mucho cuidado con las variables globales. Cada proceso al ser creado obtiene 3 frames de paginas, que luego serán utilizado para su stack y heap, tomando serias precauciones en cuanto al crecimiento de ambos, de manera que no se pisen entre sí.

7.3 Esquema general

A continuación en la figura 3, se muestra un diagrama del estado inicial del sistema una vez completada la inicialización. En la figura 4, se puede ver qué pasaría si el usuario quisiera ejecutar el proceso top.

7.4 Problemas y soluciones

En éste área nos topamos con 2 problemas importantes El primero fué cómo resolver la entrega del microprocesador de manera tal que el proceso init sólo trabajase cuando ningún otro proceso pudiera hacerlo. Para ello se resolvió que el scheduler retorne en dichos casos al proceso init, y que el mismo se encuentre, luego de realizar sus tareas pertinentes, en un loop "infinito" que espera a la terminación de sus hijos, y sólo cuando no posee nignuno, sale del loop para reiniciar el sistema, dado que en éste punto, el control sobre el sistema ya no es recuperable.

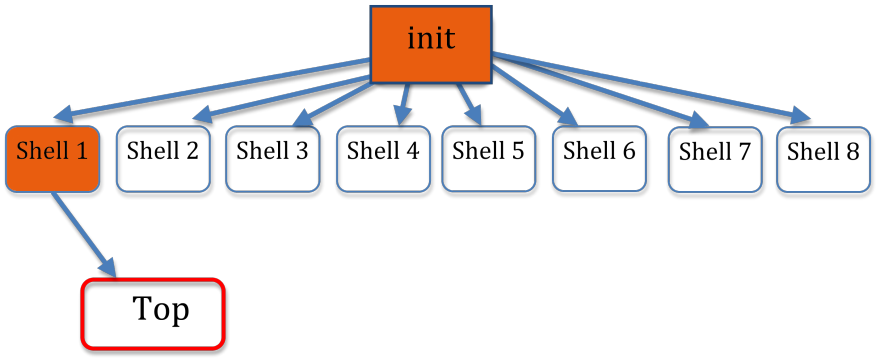

Figure 4: Estado inicial del sistema, los procesos que estan en azul están bloqueados, y los que están en naraja esperan a que terminen sus hijos, y rojo el proceso que se está ejecutando. El sentido de las flechas representa a quién se está esperando

7.5 Creacioń de procesos

La creación de los procesos supuso uno de los, si es que no el mayor desafío del trabajo práctico. Crear un proceso supone infinidades de consideraciones y acciones para su correcto funcionamiento, desde el seteo de su PID, pasando por la creación de su stack frame, hasta la asignación de su tty, la cual particularmente se hereda de su padre. Para todo esto, se definió una estructura enorme que contiene todo lo que un proceso necesita. Esta estructura puede ser consultada en la documentación Doxygen del sistema operativo, proporcionada junto con toda la documentación. Para que el lector posea un panorama general, esta estructura contiene información acerca del pid del proceso, el pid de su padre, el filesystem del proceso, su nombre, sus frames de heap y stack, su prioridad, la cuál es utilizada por el scheduler, la información de los hijos del proceso, su estado de atomicidad, su estado respecto al sistema, que puede ser, entre otras cosas, bloqueado, corriendo o listo, la tty que tiene asignada, y su nivel respecto de la tty que posee asignada, que puede ser background o foreground. El armado del stack frame de los procesos supuso la mayor complicación del procedimiento de creación de los mismos. Dado que en nuestra forma de cambio de contexto, se "engañ a" al microprocesador para que "retorne" de la interrupción del timer tick al siguiente proceso, el stack frame debe contener los datos necesarios para que este retorno de interrupción funcione correctamente, y por encima de dichos datos, la información pertinente del proceso en sí. El mayor problema fué la confección y de los punteros esp y ebp, que pasarían a ser los del proceso siguiente, ya que deberían estar construidos de manera tal que al retornar del multitasker, el stack se recuperase correctamente de los movimientos que había realizado antes de cambiar el contexto, y así luego del retorno, en el handler de la interrupción, al realizar el pop de los registros backupeados y realizar iret, el microprocesador tomase como dirección de retorno, segmento de código e eflags, los correspondientes al comienzo del proceso elegido. Otro problema creando el stack frame fue que inicialmente utilizábamos en compilación la opción -O para

optimización, el compilador en la función del multitasker hacia una compilación in-line de una función que utiliza la misma, llamada freeterminatedproceses, la cual requería 3 registros para trabajar, y además un espacio para variables locales mayor. Debdio a eso, el compilador para la función multitasker reservaba un tamaõ para las mismas mucho mayor que el estándard de 0x000018, y por ello, cuando uno arma el stock frame, en el esp tiene que reservar dicho tamaõ para que cuando en el multitasker se cambie el stock, al salir la función, la misma acomode el stack pointer adecuadamente para que obtenga la dirección de retorno correcta, y así poder luego en el handler de la int-08 con el iret "retornar" al nuevo proceso seleccionado por el multitasker. Para solucionar esto, se realizó un seguimiento línea a línea de la salida de assembler de la compilación de la función del multitasker, y así se encontró esta compilación in-line, y luego se la desactivó para verificar que solo se reservara 0x000018, y poder así definir el tamaõ para variables locales de esa forma, y poder armar correctamente el stack frame.

7.6 Herencia de TTY's

La herencia de ttys, si bien no resultó complicada, merece cierta mención. Inicialmente, los procesos no poseen una tty asignada. Pero en caso de que su padre sí la posea, estos heredan la misma. Para ello, inicialmente cuando init, que no posee una tty asignada, al momento de crear los procesos shell, les asigna a cada uno su tty mediante un system call, y a partir de allí, cualquier proceso creado por las shells hereda su tty, y así sucesivamente.

7.7 Muerte de un proceso

Al terminar un proceso, este ejecuta una función genérica "exit". En la misma, básicamente, el proceso pasa a tener un estado TERMINATED, y fuerza la interrupción de timer tick que invoca al multitasker. El mismo, luego de cambiar los procesos, siempre ejecuta una función que se encarga de limpiar los procesos que entre corrida y corrida del mismo, pudieran haber terminado, y es así como los limpia. Dicha función exit tuvo que ser incrustada en el stack frame que se generaba al crear cada proceso.

7.8 Procesos ejecutados en background

Los procesos ejecutados en background, funcionan de la misma manera que los de foreground, excepto por el hecho de que su entrada estándard no es funcional, pudiendo sólo imprimir en pantalla mediante su stdout, tal y como sucede en UNIX.

7.9 Procesos especiales

7.9.1 Top

El proceso Top es el encargado de mostrar al usuario toda la información pertinente relacionada con los procesos que se están ejecutando, desde su pid y su nombre, hasta su estado, pasando por el consumo de CPU que suponen, y su consumo de memoria, entre otras cosas. A medida que se procedía con la implementación del mismo, nuevas ideas respecto de los datos que podían

ser mostrados, iban vieniendo a nuestras mentes, dado que poco a poco nos dábamos cuenta de las posibilidades que poseíamos para mostrar información, dado que la estructura de cada proceso posee mucha información importante.

7.9.2 Shell

En esta sección se hablará acerca de como fué adaptada la shell para poder funcionar en un sistema multi tarea.
Como la shell que se tomó como base fue diseñada para correr en un sistema multi tarea, se tuvieron que realizar ciertas adaptaciones. Una de ellas fué, que las variables globales deberían pertenecer sólamente a esa shell, por esa razón se realizó un vector donde fueron almacenadas estructuras con todos los datos pertinentes para el correcto funcionamiento de la shell. Cada shell almacenaba en su heap dichas estructuras, y ésto asegura un nivel más de seguridad, ya que una shell no podrá acceder a datos de otra, por que no estarían presentes las paginas de la misma.
En ella se implementaron procesos de prueba que se pueden ejecutar para testear la funcionalidad del sistema.

7.10 Problemas y soluciones

La clave en el desarrollo del proceso Top se encontró en el cómputo del consumo del CPU. Para ello, cada proceso posee en su entrada de la tabla de procesos, un campo que supone un conteo de timer ticks durante los cuales recibió el procesador. Los mismos son utilizados por el proceso top, mediante system calls, para computar un promedio de utilización en porcentaje del CPU. Dichos valores son reseteados entre vuelta y vuelta del proceso top, a modo de poder ir actualizándolos.

7.11 Limitaciones

Por cuestiones del algoritmo que computa la información del consumo del CPU, el proceso Top sólo puede estar corriendo en una sóla tty.

8 Multitasker

8.1 Objetivo

Intercambiar tareas mediante software, y una correcta construcciíon del stack frame.

8.2 Modelo

El multitasker salvará el contexto de la tarea actual, apagará la presencia de sus páginas, cargará el contexto del siguiente proceso, previo encendido de la presencia de sus páginas.

8.3 Esquema general

Si bien el trabajo del multitasker no es demasiado complicado, sí lo es la creación del stack frame de modo que este cambio de contexto se realice correctamente, pero eso es un punto que ya ha sido tratado anteriormente.

8.4 Problemas y soluciones

Un gran problema que se tuvo con el algoritmo del scheduler fue que, al momento de cambiar los contextos, si el scheduler devolvía al proceso init, debido a que no se tenía otro proceso para correr, si el proceso al cual se le está por quitar el procesador se encuentra en condiciones de seguir ejecutándose, se le otorga el procesador al mismo nuevamente, sin realizar el cambio de contexto. Esto pudo realizarse mediante una simple comprobación luego del retorno del scheduler, pero notar esa comprobación tomó varias horas de análisis del algoritmo del mulstitasker.

9 Scheduler

9.1 RPG

9.1.1 Modelo

Implementar un algoritmo basado en juegos RPG. El algoritmo consite en asignarle un puntaje a un proceso en cada llamado del scheduler. Cuando un proceso llega al puntaje designado como máximo se lo agrega a la lista de procesos listos para ejecutar. Luego se verifica qué proceso es el que tiene más antiguedad. Una vez devuelto el proceso a procesar, se reinicia el contador de puntos rpg y la antiguedad.

9.1.2 Problemas y soluciones

Principalmente tuvimos un problema manejando la antiguedad de los procesos, luego decidimos tomar una función de evaluación que nos permite ir pasando por todos los procesos, que no alcanzen inmediatamente el puntaje máximo de rpg, para luego ser atendido.

9.1.3 Limitaciones

La función de evaluación en el caso de que se aumente la cantidad de procesos o disminuya habría que cambiarla, porque es en función de prioridades y cantidad máxima de procesos. Tambien habría que modificar el puntaje máximo de rpg a alcanzar.

9.2 Round-Robin

9.2.1 Objetivo

Implementar un algoritmo del tipo Round Robin. Básicamente se comporta como una lista circular donde se van devolviendo todos los procesos que esten en estado **READY** en el orden que se encuentran en la tabla de procesos.

9.2.2 Modelo

El modelo tomado es el siguiente. Se toma la lista de procesos y se la recorre en forma lineal verificando si algun proceso se encuentra listo para ser atendido, en el caso de que haya alguno se lo devuelvo y en la próxima llamada se arranca a recorrer desde esa posición y se realiza el mismo recorrido ya explicado anteriormente.

9.2.3 Problemas y soluciones

En un principio el algoritmo se quedaba en un loop infinito ya que buscasa desde init y como siempre estaba listo lo devolvia y a veces no retornaba o era interrumpido. Por lo tanto una solución fue llamar al scheduler en caso de que haya algun proceso para correr distinto de init. En un principio teníamos problemas con el algoritmo porque compilabamos con la opción de optimización y se agregaban funciones en el código que no queriamos, luego de darnos cuenta de ese error, comenzó a funcionar el algoritmo. En la recta final del desarrollo del

sistema operativo, una vez implementado el método mediante el cual el proceso init sólo recibe el procesador cuando ningún otro proceso puede recibirlo, el algoritmo de round robin comenzó a fallar. Simplemente no retornaba el proceso init en los casos que debía. Para ello, se revisó detalladamente, ahora con mayores conocimientos acerca de lo que realmente debía hacer, y se encontraron algunos errores en el mismo, que pudieron facilmente ser subsanados.

9.2.4 Limitaciones

El algoritmo en sí no maneja prioridades, sino el orden el cual fueron creados los procesos. Se podría hacer que la creación de procesos los inserte en forma ordenada en la tabla de procesos, y por lo tanto el algortimo *Round Robin* de alguna forma estaría teniendo en cuenta cierto nivel de prioridades.

9.3 IPCs Implementaciones

9.3.1 Shared memory

Se implementó un IPC basado en Shared memory, donde la cantidad de segmentos de shared memory está definida por un define. Se decidió utilizar la estructura que utiliza System Five para la shared memory y Posix para los semáforos. Cada shared memory ya tiene asociado un semáforo y la cantidad de frames que tiene desiganados para utilizar. La implementacion de shared memory y de los semáforos se realizó con un cuidado extremo, metodología que se adoptó a partir de conocimientos obtenidos en el primer trabajo práctico de la materia, que nos enseñaron que errores ínfimos en la implementación y/o utilización de una shared memory, desembocarían en resultados desastrozos e indebuggeables. Si bien la shared memory se instancia junto con un semáforo que se le provee al usuario para que controle el uso de la misma, el módulo de semáforos puede ser utlizado en cualquier parte y no únicamente para el uso de la shared memory. Los mismos fueron implementados de manera tal de que mediante un simple parámetro en su creación, los mismos funcionen como semáforos que bloquean el uso de un recurso hasta que el mismo se libere, o como un sistema de espera hasta ser señalizado por otro proceso.

9.3.2 Objetivo

Implementar un juego en donde se pueda verificar la comunicación entre procesos mediante un IPC, en nuestro caso **Shared Memory**.

9.3.3 Modelo

En un principio se decidió implementar como juego un Chat entre varias shells, pero finalmente se eligió hacer la batalla naval con ciertas restricciones. La creación de barcos se genera en forma aleatoria, es decir que se implementó un random muy básico donde usa un polinomio de grado 1 y como semilla utiliza los ticks realizados por el timer tick hasta el momento desde la carga del sistema operativo. El juego es de dos personas únicamente, como la tradicional batalla naval.

9.3.4 Esquema general

La forma en que se nos ocurrió para comunicar los dos procesos es que cuando se crea ese proceso, el juego proporciona la opción de ser host o unirse a una partida ya creada. Por lo tanto cuando se crea el primer proceso uno puede elegir entre ser host o unirse a una partida ya creada, y la forma de unirse es que cuando se crea el proceso host, se le muestra en pantalla el ID de la shared memory creada, por lo tanto el jugador 2 debe ingresar ese id para unirse a esa partida. Una vez ya pasados los primeros pasos comienza el juego. La información que viaja por la shared memory son las posiciones del tablero en donde se ubicaron las bombas. En un principio se había dicho de poner el tablero de cada jugador en la shared memory y que el jugador escribiera directamente sobre esa posición, pero nos dimos cuenta que en verdad la información que necesitan los dos son las posiciones del tablero para cada uno actualizar donde fué puesta la bomba y verificar si le dió a algun barco o si fué agua.
Cada vez que un usuario ingresa una posición en donde desea ubicar la bomba, se bloquea el tablero y una vez enviado el mensaje se desbloquea. De esta forma evitamos que los dos jugadores pongan en el mismo instante dos posiciones distintas y se mezclen sus ubicaciones.

9.3.5 Problemas y limitaciones

Un problema fue tomar la decisión de ubicar los barcos, si el usuario los podía ubicar o si lo generábamos nosotros, pero se optó por la segunda y los barcos pueden estar unicamente verticales u horizontales, no diagonales.
Otro problema que nos surgió fue que los semáforos se quedaban esperando pero se seguia imprimiendo el tablero esperando la conexión de algún usuario al juego, por lo tanto decidimos modificar los semáforos para que se queden esperando y no pueda realizar otra cosa, entonces cuando alguien ingresa una posición, recién entonces se refresca el tablero.

10 Otros problemas de relevancia

Los cli y sti, que deshabilitan interrupciones por hardware fueron también un problema. Dado que no son anidables, al comienzo fue complicado ver donde deberían utilizarse los cli y sti correspondientes para lograr una atomicidad apropiada par los manejos de interrupciones. Luego de un extenso análisis, se corroboró que al comienzo de un manejador de interrupción, deshabitar las interrupciones mediante cli no implicaba que al retornar las mismas siguieran deshabitadas, dado que se recuperan los eflags del proceso que fue interrumpido, y en dichos eflags las interrupciones seguían habilitadas. Por ello, se concluyó que sti no es necesario utilizarlo dado que consecutivas llamadas a interrupciones, sin retornar de la primera, no requieren de ningún sti dado que deben ser todas atómicas, y que se restituirían las mismas en cuanto la primera retornase. Es por ello que Sti solo se utiliza en el modulo de arranque del kernel para poder cargar correctamente el vector de interrupciones. Los cambios de contextos por sí solos logran restaurar la habilitación de interrupciones al recuperar sus eflags. Antes de todo esto, se intentó incluir cli y sti en la función increaseKernelDepth, dado que en la misma (encargada de setear en presente las paginas del heap del kernel cuando se realiza una llamada al sistema, y manejar bien las consecutivas indentaciones hacia el kernel), se pone en presente dichas paginas sólo en la primer llamada a increase, y sólo se deshabitan en la ultima llamada a decrease, siempre y cuando la cantidad de increases sea igual a la de decreases. Si bien posicionar cli en increase no resultaba un problema, sí lo resultaba ser el sti, dado que la imposibilidad de anidacion de sli y cti arruinaban la lógica.

Tuvimos otro problema al invocar la llamada a increase y decrease en los manejadores de interrupciones, dado que los mismos utilizan algunos registros, y antes de llamarlos se debía realizar un pusheo de los mismos, y recuperarlos a su retorno. Esto tomó bastante tiempo dado que es algo muy sutil que no debe ser tomado a la ligera. Por suerte se logró encontrar este error y salvar los registros pertinentes.

El último gran problema fue que en un momento no podíamos crear mas de 4 procesos, ya que al crear el 5to, tiraba page fault la creación de su stack frame. Tomo mucho tiempo encontrar la razon, tuvimos que seguir el código en assembler con el debugger de bochs para ver que direccion estaba intentando acceder. Eso se pudo hacer gracias a que en el CR2 queda la dirección que se quiso acceder que tiró page fault. Por ello notamos que estaba intentando acceder a cualquier lugar. Se debuggeo toda la paginación y se encontro que el armado de los frames de páginas que nosotros utilizamos, tenía un error en el algoritmo, y dicha revisión llevó a encontrar errores en la funcion que pone y saca la presencia de los frames, y además en el archivo defs.h donde figuran la mayoria de los defines del sistema operativo, que, para nuestra sorpresa, tenía valores viejísiimos que no concordaban con la estructura actual del sistema. Una vez correjidos dichos valores y funciones, la paginación fue finalmente correcta y pudo crearse tantos procesos como se quiso. Cabe destacar que este problema de paginación desencadenaba en que el malloc implementado no funcionase correctamente, pero, afortunadamente, al reparar la paginación, el malloc también comenzó a funcionar correctamente.

10.1 Conclusión

Fué de mayor importancia, seguir algunas convenciones de UNIX, ya que se facilitó el diseño del proyecto en general. Por otro lado, cabe destacar que es bastante complejo adaptar un sistema mono tarea a multi tarea tal y como lo tuvimos que hacer, debido a que esto generó problemas serios en el desarrolo del trabajo, más que nada en las últimas etapas. Se quiere resaltar que si no se toman desiciones correctas en la etapa de diseño del proyecto, se puede llegar a un punto en donde no se tiene otra solucion que tener que realizar una restructuración por demás importante. Por esta razón se decidió como se dijo anteriormente aplicar a grandes rasgos las estrucutara que utiliza UNIX.
Se optó por un kernel monolítico, por una cuestión de eficiencia, y codificación simple. El cambio de contexto resultó muy complejo, tal y como yá se explicó en el informe, ya que este implica una precisión a la que no estábamos acostumbrados. Se hace hincapié en este tema más que nada en la cosntrucción del stackframe.
A diferencia de proyectos anteriores, se realizó una programción en parejas, ya que se necesitaba de una atención constante, y mediante la misma, se notó que la programación a través de esta metodología, no solo que no representa una pérdia de horas hombre por estar dos personas en una misma máquina, escribiendo sólo una de ellas, sino que estamos bastante seguros de que rindió mucho más que si hubieramos programado por separado. Se pudo optar por esta metodología porque el código no era extenso, sino más bien complejo.
Al momento de desarrollar el juego se decidió buscar un juego y adaptarlo a las necesidades, ya que el trabajo no estaba orientado a esa tarea.
Se pudieron afianzar conocimientos sobre la estructura y la funcionalidad de un sistema operativo, creacion de procesos y cambios de contexto. Quedó en evidencia la razón de muchas deciciones que toman los desarrolladores de sistemas operativos, cuando estos están en etapa de desarrolo. Nos resultó más que evidente que todo el proceso de diseñar y desarrollar un sistema operativo multi tarea correcto, eficaz y eficiente, no es una tarea para nada fácil, y que requiere no sólo de gran cantidad de programadores, si es que se desea terminarlo en menos de 20 aõs, sino que también es necesario poseer conocimientos muy avanzados. Es por todo esto que, un poco en tono serio, y un poco en tono cómico, creemos seriamente que Richard Stallman y Linus Torvalds son dioses.

10.2 Bibliografía y fuentes

- Unix System Programming - Keith Haviland, Dina Gray, Ben Salama - Addison Wesley
- Operating Systems: Design and Implementation - Andrew S. Tanenbaum - Prentice Hall
- OSDEV Website - http://wiki.osdev.org
- StackOverflow WebSite - http://www.stackoverflow.com