Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Sistemas Operativos

Segundo Cuatrimestre de 2006

Trabajo Práctico Minix

Fecha de Entrega: 05/12/06

Grupo 8	LU	Correo electrónico
Martín Cammi	676/02	jack_1640@yahoo.com
Federico Mendez	703/02	mendezfede@yahoo.com.ar
Laura Perez Vultaggio	554/03	laura_vultaggio@yahoo.com.ar

RESUMEN

El presente trabajo se focalizará principalmente en la modificaci'on del sistema operativo Minix, creado por Andrew S. Tanenbaum. Los puntos alcanzados incluir'an desde la descripción de los comandos mas comunes y su utilización, hasta la modificación de la placnificación de procesos y de memoria del sistema así como también la implementación de semáforos para manejar la exclusión mutua entre procesos

PALABRAS CLAVE

Minix - sistema operativo - semáforos - procesos

Índice

1. Enunciado **3**

1.1. Objetivos del práctico 3

1.2. Herramientas necesarias 3

1.3. Formato de Entrega 4

1.4. Cambios al enunciado del práctico, fechas de entrega, etc. 4

1.5. Aclaraciónes Previa 6

2. Ejercicios **6**

2.1. El comando `make` 8

2.2. El comando `mknod` 9

2.3. Comandos Básicos de MINIX/UNIX 10

2.4. Uso de STDIN, STDOUT, STDERR, PIPES 20

2.5. Administrador de Usuarios 22

2.6. Ejecución de procesos en Background 25

2.7. Compilación del kernel de minix 27

2.8. Características de Minix 29

2.9. Implementación de un Driver 39

2.10. Modificación del Scheduler y la adm. de Memoria 41

3. Modificación de Código **42**

3.1. Modifique el "Scheduler" original del MINIX para el nivel de usuario . . 42

3.1.1. Planteo 42

3.1.2. Expectativas 43

3.1.3. Testeos y resultados 43

3.1.4. Conclusiones 44

3.2. Modifique la administración de memoria original del MINIX 44

3.2.1. Planteo 44

3.2.2. Expectativas 45
3.2.3. Testeos y Resultados 45
3.2.4. Conclusiones 47
3.3. Imlpementación de un System Call 47
3.4. Primitivas para manejo de semáforos y las primitivas P y V 50
3.4.1. Interfaz de Usuario 51
3.4.2. Implementación 52
3.4.3. Modificaciones para agregar las System Calls 54
3.4.4. Estructuras de datos en la implementación en el MM 56
3.4.5. Código de la Implementación en MM 56
3.4.6. System Calls 56
3.4.7. Task calls en microkernel 63
3.5. Pruebas 65
3.5.1. Prueba de exclusión mutua del recurso pantalla 66

Sistemas Operativos - Segundo Cuatrimestre 2006
Trabajo Práctico Minix

---

1. Enunciado

1.1. Objetivos del práctico

Al terminar este trabajo Ud. habrá aprendido a:

- Instalar MINIX 2.0 en su versión DOSMINIX o sobre un simulador.
- Utilizar convenientemente los principales comandos de MINIX.
- Compilar y ejecutar programas escritos en lenguaje C.
- Aplicar en forma práctica algunos de los conocimientos brindados en la materia.
- Realizar modificaciones al Sistema Operativo MINIX.

1.2. Herramientas necesarias

Para resolver los ejercicios propuestos necesitará:

- Una PC con Windows 95/98 con, al menos, 80 Mb de Disco o el entorno necesario para instalar el simulador elegido.

Requisitos de Entrega

Lugar y Fecha de entrega:

- La fecha y hora de entrega para este práctico es la que figura en el cronograma de la materia. (se alienta y acepta la entrega del trabajo, en su totalidad, en forma anticipada)
- Los trabajos deben ser entregados PERSONALMENTE a alguno de los docentes de la cátedra en los horarios de clase o de consulta. No se aceptarán trabajos depositados en el Departamento de Computación o en cualquier otro lugar.
- No se aceptarán trabajos incompletos.

1.3. Formato de Entrega

Se deberá entregar en un medio digital, preferentemente CD, una o varias imágenes del sistema operativo, con los cambios efectuados al mismo y que contenga además todas las demás resoluciones en formato fuente, formato ejecutable y los programas de prueba que se utilizaron para comprobar que los cambios o resoluciones funcionan correctamente. Además se deberán entregar todos los archivos necesarios para que esa imagen o imágenes ejecuten perfectamente.
Se deberá, además, entregar un DOCUMENTO IMPRESO. Ese documento debe reunir las siguientes características:

1. Formato de Presentación
   a) Impreso en hojas de tamaño A4 encarpetadas.
2. Secciones del documento (Todas obligatorias):
   a) Carátula de presentación: Debe incluir OBLIGATORIAMENTE:
      1) Asignatura
      2) Número y Descripción del trabajo práctico
      3) Año y Cuatrimestre de Cursado
      4) Identificación del Grupo
      5) Nombres, Apellidos y direcciones de correo electrónico de TODOS los Integrantes del grupo
3. Sección Principal: Aquí debe incluirse la resolución de cada uno de los problemas planteados. Para cada respuesta debe indicarse OBLIGATORIAMENTE, el número y título del problema al que corresponde tal como aparece en el enunciado y los comandos y/o programas utilizados para resolverlos. Se deberá indicar claramente en que directorio y bajo que nombre se encuentran los fuentes, los ejecutables y los programas de prueba

1.4. Cambios al enunciado del práctico, fechas de entrega, etc.

Cualquier cambio en los enunciados, fechas de entrega, etc. será informado utilizando dos métodos:

- La página Web de la Materia.
- La lista de correo sisop@listas.dc.uba.ar.

Ud. no puede alegar que no estaba al tanto de los cambios si esos cambios fueron anunciados utilizando alguno de los dos métodos.

SUGERENCIA: Consulte frecuentemente la página de la materia y asegúrese de que ha sido incorporado a la lista de correos.
Los grupos serán de hasta un máximo de tres (3) integrantes.

1.5. Aclaraciónes Previa

Ayudas:
`http://www.dc.uba.ar/people/materias/so/html/minix.html`

Setear Teclado Español

```
cp /usr/lib/keymaps/spanish.map /etc/keymap
echo loadkeys /etc/keymap >> /etc/rc
sync; sync; shutdown
```

Pruebe el ash, el man, el apropos y el mined

Hemos decidido cambiar el *shell* que posean por *default* los usuarios que utilicemos a lo largo de este trabajo, por razones de comodidad, tales como poder acceder al historial de comandos ejecutados, poder utilizar la tecla TAB con los resultados conocidos similares a Linux, etc .... A dicho fin, hemos utilizado el comando:
`chsh <usuario_en_cuestion> /usr/bin/ash`
Como es de suponer el comando `chsh` cambia el *shell* del `<usuario_en_cuestion>`.

2. Ejercicios

1. Instalación del Sistema Operativo

   a) Indique gráficamente el layout del sistema operativo MINIX indicando sus componentes y las funciones que brindan. En especial indique los *system_calls* que contienen.

   Rta.:

   Minix es un Sistema Operativo multiusuario y multitarea cuya Administración de procesos a diferencia de otros sistemas operativos donde el kernel es del tipo monolítico (no esta dividido en modulos), Minix lo está y posee una serie de procesos que se comunican entre sí junto con los procesos de usuario a traves de la transferencia de mensajes. El uso de este diseño permite tener una estructura modular y flexible y hace mas sencilla la tarea de por ejemplo reemplazar todo el sistema de archivos por uno totalmente distinto sin tener que recompilar todo el kernel.

   Minix está estructurado en cuatro capas cada una de las cuales realiza una función especifica. A continuación se muestra un gráfico que ilustra estas capas:

<table>
<tr><td>4</td><td>Init</td><td colspan="2">Proceso de usuario</td><td colspan="2">Proceso de usuario</td><td>Proceso de usuario</td><td>...</td><td>Procesos de usuario</td></tr>
<tr><td>3</td><td colspan="2">Administrador de memoria</td><td colspan="2">Sistema de archivos</td><td colspan="2">Servidor de red</td><td>.....</td><td>Procesos servidores</td></tr>
<tr><td>2</td><td colspan="2">Tarea de disco</td><td>Tarea de terminal</td><td>Tarea de reloj</td><td>Tarea de sistema</td><td>Tarea de Ethernet</td><td>...</td><td>Tareas de E/S</td></tr>
<tr><td>1</td><td colspan="7">Administración de procesos</td><td></td></tr>
</table>

Figura 1: Layout de Minix

La capa de mas bajo nivel es la de administración de procesos, es quien intercepta todas las interrupciones, se encarga además de la planificación de procesos y ofrece a las capas superiores un conjunto de servicios independientes con los cuales se comunica a través de mensajes.

La capa siguiente contiene los procesos de E/S(salvo la tarea del sistema). Se requiere una tarea por cada tipo de dispositivo, incluidos discos, impresoras, terminales, interfaces de red y relojes. Asimismo, si existen otros dispositivos de E/S, se necesitará agregar una nueva tarea por cada uno de ellos. Estas dos primeras capas se combinan conformando el *kernel*.

La tercer capa se encarga de proporcionar servicios de utilidad a todos los procesos de usuarios denominados *servidores*. Estos procesos servidores se ejecutan a un nivel de privilegio menor al del kernel por lo que no pueden, por ejemplo, acceder directamente a puertos de E/S. Por el administrador de memoria pasan todas las llamadas al sistema que requieren administración de memoria, mientras que por el sistema de archivos pasan aquellas llamadas que involucran operaciones sobre archivos. Los servidores se inician cuando se inicia el sistema, y se mantienen activos mientras el sistema lo esté.

La cuarta y última capa es la superior, en ella se encuentran todos los procesos de usuario, como ser *shells*, editores, compiladores y programas escritos por el usuario.

Como una característica de los procesos en Minix podemos decir que estos pueden generar subprocesos hijos que a su vez tambien pueden generar otros subprocesos desde ellos formando un arbol de procesos, incluso todos los procesos de usuarios del sistema forman parte de un solo arbol cuya raiz es el proceso *mit* al inicio del arranque.

b) Instalar MINIX en su versión para DOS (DOSMINIX), WINDOWS o LINUX (BOCHS) según se describe en
`http://www.dc.uba.ar/people/materias/so/html/minix.html`

2. Herramientas

   Indique que hace el comando `make` y `mknode`. Cómo se utilizan estos comandos en la instalación de MINIX y en la creación de un nuevo kernel. Para el caso del make muestre un archivo ejemplo y explique que realiza cada uno de los comandos internos del archivo ejemplo.

   Rta.:

2.1. El comando `make`

Al desarrollar software en un ambiente dónde debemos compilar el código fuente que escribimos, a menudo nos ocurre que debemos compilar una gran cantidad de archivos (archivos de Assembler, código C, *linkear* con librerías, etc . . .) antes de poder tener al programa corriendo en la computadora. Por suerte, Minix se encarga de esta tarea monótona y muchas veces improductiva, si utilizamos `make`.

El comando `make` permite realizar mantenimiento a programas (en general, es más útil cuando los programas son grandes).
En un archivo `makefile` ó `Makefile` espera tener especificadas las dependencias de compilación entre los archivos fuente del programa, es decir, qué pasos se deben realizar para compilar, ensamblar, enlazar, o cualquier otro tipo de tareas realizables sobre el programa en cuestión. Este programa permite detectar modificaciones en los archivos objeto y/o fuentes a fin de realiar únicamente los pasos que sean necesarios para tener el programa actualizado (es decir, si ya está compilado un archivo determinado y no se modificó, no se lo vuelve a compilar).

A continuación, mostraremos un ejemplo de Makefile, que corresponde a un trabajo práctico que hemos realizado en la materia "Organización del Computador II" [1]

```
all: oc2grep
clean:
    rm *.o oc2grep

oc2grep: parser.o filtro.o main.o
    g++ main.o parser.o filtro.o -o oc2grep

parser.o: parser.cpp
    g++ -c parser.cpp -o parser.o
```

[1] Se trata de un programa similar al *grep* de Linux, pero hecho en Assembler de Intel, utilizando expresiones regulares y autómatas finitos.

```
filtro.o: filtro.asm
    nasm -g -f elf -Dsistema=1 filtro.asm

main.o: main.asm
    nasm -g -f elf -Dsistema=1 main.asm
```

Explicación:

- Al invocar al comando `make` con el parametro `all`, será lo mismo que si lo invocasemos con el parametro `oc2grep`
- Al invocarlo con `clean`, ejecutará el comando `rm *.o oc2grep` que borra todas las compilaciones que habíamos hecho con anterioridad. Esto sirve para cuando queremos asegurarnos que se compilen todos los archivos fuentes de nuevo.
- En la línea que marca que se debe hacer cuando se lo invoca con `oc2grep`, le estamos diciendo al comando que ejecute el compilador GNU de C++ con tres archivos de código objeto y que lo ensamble en un ejecutable llamado `oc2grep`. Pero lo más importante de todo es que, en este caso particular, le estamos marcando las dependencias de compilación que posee nuestro programa:
  para poder hacer lo que ya hemos mencionado, el comando `make` deberá antes resolver las "etiquetas" `parser.o`, `filtro.o` y `main.o`; las cuales, cada una de ellas, posee comandos específicos a realizar *con anterioridad* a la compilación de la "etiqueta" `oc2grep`.

En este ejemplo, hemos visto cuan útil es éste comando, si no lo usásemos, deberíamos haber hecho estas compilaciones *a mano* o en el mejor de los casos, con un script, el cual (suponiendo que se tratara de un script simple) no nos provee la facilidad de evitar la compilación de un archivo que no ha cambiado.

2.2. El comando `mknod`

mknod - crea archivos especiales de bloques ó caracteres ó FIFO

SINOPSIS

mknod nombre b/c/p [mayor menor]

DESCRIPCIÓN

nombre: nombre definido por el usuario para nombrar al dispositivo.
El argumento que sigue a nombre especifica el tipo de fichero a construir:

- p para un FIFO
- b para un archivo especial de bloques (con buffer)
- c para un archivo especial de caracteres (sin bufferes)

Tras el tipo del archivo hay que indicar los números mayor y menor. Mayor indica el número que tiene asignado el driver encargado de un cierto tipo de periferico, y menor indica la instancia de ese dispositivo. (deben indicarse en base diez, o en octal con un 0 inicial). Por omisión, los permisos de los ficheros creados son 0666 ('a+rw') menos los bits puestos a 1 en la umask.

Un archivo especial se almacena como una tripleta (booleano, entero, entero) en el sistema de archivos. El valor lógico determina si es un archivo especial de caracteres ó un archivo especial de bloque. Los dos enteros son los números de dispositivo mayor y menor.

Así, un archivo de este tipo casi no ocupa sitio en el disco, y se emplea sólo para la comunicación con el sistema operativo, no para almacenamiento de datos. A menudo los archivos especiales se refieren a dispositivos físicos (disco, cinta, terminal, impresora) o a servicios del sistema operativo (dev/null, /dev/random).

Los archivos especiales de bloque son dispositivos similares a discos (donde se puede acceder a los datos dado un número de bloque, y p.ej. tiene sentido tener un caché de bloques). Todos los otros dispositivos son ficheros especiales de caracteres. (Hace tiempo la distinción era diferente: la E/S a un archivo especial de caracteres sería sin buffer, y a un archivo especial de bloques, con buffer.)

3. 2.3. Comandos Básicos de MINIX/UNIX

3.0. Póngale password root a root.

Rta.:

`passwd`: Cambia la contraseña del usuario que invocó el comando.

`passwd [usuario]`: Cambia la contraseña de un usuario especificado. Sólo el súper-usuario(root) puede cambiar la clave de cualquier usuario.

Pasos realizados:

a) Loguearse como *root*

b) Invocar `passwd` sin argumentos

c) Escribir la clave: "root"

d) Confirmar la clave escrita: "root"

Finalmente, la nueva clave establecida para el súper-usuario es "root"

3.1. `pwd`

Indique qué directorio pasa a ser su current directory si ejecuta:

`pwd`: Muestra el nombre del directorio actual.

3.1.1. `# cd /usr/src`

Rta.:

El directorio actual pasa a ser `/usr/src`

3.1.2. `# cd`

Rta.:

El directorio actual pasa a ser `/`

3.1.3. ¿Cómo explica el punto 3.1.2?

Rta.:

El comando `cd` invocado sin parámetros establece como directorio actual el indicado por la variable de entorno `$HOME`. En el caso del súper-usuario, dicha variable de entorno referencia al directorio `/`.

3.2. `cat`

Cual es el contenido del archivo `/usr/src/.profile` y para que sirve.

Rta.:

`cat`: Concatena archivos y escribe el resultado en la salida estándar (stdout).

Para ver el contenido del archivo `/usr/src/.profile` el comando se invoca de la siguiente manera:

```
cat /usr/src/.profile
```

La salida obtenida es:

```
# Login shell profile.

# Environment.
umask 022
PATH=.:/usr/local/bin:/bin:/usr/bin
PS1="-w# - "
export PATH,PS1

# Erase character, erase line, and interrupt keys.
stty erase '^H' kill '^U' intr '^?'

# Check terminal type.
case $TERM in
dialup|unknown|network)
        echo -n "Terminal type? ($TERM) "; read term
```

```
        TERM="${term:-$TERM}"
esac

# Shell configuration.
case "$0" in *ash) . $HOME/.ashrc;; esac
#
```

El archivo `/usr/src/.profile` contiene la personalización del entorno minix. Contiene la variable de entorno `PATH`, es decir, una variable cuyo valor indica los directorios dónde minix buscará cuando ejecutemos un comando sin explicitar la ruta de acceso a él.

Otra variable de entorno interesante es `PS1`, la cual define el "prompt" que utilizará el intérprete. Con la orden "export", podemos exportar dicha variable (y otras mas) al entorno. El entorno es el conjunto de variables a las cuales tienen acceso todas las órdenes que el usuario ejecute.

3.3. `find`

En que directorio se encuentra el archivo `proc.c`

Rta.:

Para obtener el directorio en el cual se encuentra el archivo `proc.c`, podemos invocar el comando `find` de la siguiente manera:

```
find / -name proc.c -print
```

De esta forma, estamos indicando que busque el archivo recursivamente desde el directorio `/`, que se desea buscar los archivos cuyo nombre sea `proc.c`, y que los resultados se deben mostrar por la salida estándar (si no se escribe `-print`, igualmente se muestra el resultado por salida estándar).

La salida obtenida es:

```
/usr/src/kernel/proc.c
```

3.4. `mkdir`

Genere un directorio `/usr/<nombregrupo>`

Rta.:

Para crear un directorio en el directorio `/usr` el comando se debe invocar de la siguiente manera:

```
mkdir /usr/<nombre>
```

donde `<nombre>` es el nombre del directorio a crear. Para el ejercicio, llamaremos a este directorio *grupoSisOp*. Creamos el directorio con la siguiente invocación:

```
mkdir /usr/grupoSisOp
```

3.5. `cp`

Copie el archivo `/etc/passwd al directorio /usr/<nombregrupo>`

Rta.:
Para copiar el archivo `/etc/passwd` al directorio `/usr/grupoSisOp`, escribimos la siguiente instrucción:

```
cp /etc/passwd /usr/grupoSisOp
```

3.6. `chgrp` Cambie el grupo del archivo `/usr/<grupo>/passwd para que sea other`

Rta.:
Para cambiar el grupo al que pertenece el archivo `/usr/grupo/passwd` y establecerlo en *other* se debe ejecutar lo siguiente:

```
chgrp other /usr/grupoSisOp/passwd
```

3.7. `chown` Cambie el propietario del archivo `/usr/<grupo>/passwd para que sea ast`

Rta.:
Para cambiar el propietario al que pertenece el archivo `/usr/grupo/passwd` y establecerlo en ást'se debe ejecutar lo siguiente:

```
chown ast /usr/grupoSisOp/passwd
```

3.8. `chmod` Cambie los permisos del archivo `/usr/<grupo>/passwd` para que

- el propietario tenga permisos de lectura, escritura y ejecución

  Rta.:
  Para que el propietario tenga permisos de escritura, lectura y ejecución sobre el archivo `/usr/grupoSisOp/passwd` se debe invocar:

  ```
  chmod u+r+w+x /usr/grupoSisOp/passwd
  ```

- el grupo tenga solo permisos de lectura y ejecución

  Rta.:
  Para que el grupo tenga solo permisos de lectura y ejecución sobre el archivo `/usr/grupo/passwd` se debe invocar:

  ```
  chmod g+r+w-x /usr/grupoSisOp/passwd
  ```

  Es decir, se debe sacar el permiso de escritura (o no hacer nada si no lo tenía).

- el resto tenga solo permisos de ejecución

  Rta.:
  Para que el resto (es decir, los que no son ni el propietario ni miembros del grupo) tenga solo permisos de ejecución sobre el archivo `/usr/grupoSisOp/passwd` se debe invocar:

  ```
  chmod o-r-w+x /usr/grupoSisOp/passwd
  ```

3.9. `grep`
Muestre las lineas que tiene el texto include en el archivo `/usr/src/kernel/main.c`

Rta.:

Para realizar lo solicitado se ejecuta el comando

```
grep include /usr/src/kernel/main.c
```

La salida obtenida es:

```
#include "kernel.h"
#include <signal.h>
#include <unistd.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"
```

Muestre las lineas que tiene el texto POSIX que se encuentren en todos los archivos `/usr/src/kernel/`

Rta.:

Para realizar lo solicitado, y con el único objetivo que la salida no sea poco visualizable en este documento, se ejecuta el comando `grep` con el modificador `-l` para que sólo muestre los archivos que contienen el texto POSIX. Si queremos cumplir al pie de la letra el enunciado, deberíamos ejecutar el mismo comando pero sin el mentado modificador.

```
grep -l POSIX /usr/src/kernel/*
```

La salida obtenida es:

```
/usr/src/kernel/kernel.h
/usr/src/kernel/rs232.c
/usr/src/kernel/system.c
/usr/src/kernel/tty.c
```

3.10. `su`

3.10.1. ¿Para qué sirve?

Rta.:

El comando `su` permite loguearse temporalmente como súper-usuario (*root*) u otro usuario. Crea un *shell* extra en memoria y pide el password del usuario con el cual me quiero loguear temporariamente. Es útil para hacer determinadas tareas como un usuario particular (distinto al actual, obviamente) y luego volver transparentemente al usuario con el que estabamos trabajando previamente.

3.10.2. Que sucede si ejecuta el comando `su` estando logueado como *root*?

Rta.:

Como ya dijimos, el comando `su` instala una nueva instancia del shell en memoria y la ejecuta sobre la anterior. Estando como *root*, si se ejecuta el comando sin

argumentos, no se pide password, pero igualmente se genera y ejecuta la nueva instancia.

3.10.3. Genere una cuenta de `<usuario>`

Rta.:
Vamos a crear el usuario *user1*:

```
adduser user1 other /usr/user1
```

Con esto, estamos creando dicho usuario, especificando que el grupo al que pertenece será `other` y que su directorio *home* será `/usr/gabriel`.

3.10.4. Entre a la cuenta `<usuario>` generada

Rta.:
Para realizar esto, no deslogueamos, utilizando el comando `exit` y cuando aparece el login en pantalla ingresamos `gabriel`.

3.10.5. Repita los comandos de 3.10.2

Rta.:
Observamos que ejecutando el comando `su`, se nos pide que ingresemos password, y al ingresar el password de *root*, se nos abre una instancia del shell, logueados como *root*.

3.11. `passwd`

3.11.1. Cambie la password del usuario *nobody*

Rta.:
ejecutamos el comando `passwd nobody` y se nos pide que ingresemos el nuevo password y su confirmación. El nuevo password que ingresamos es: "nobody".

3.11.2. presione las teclas ALT-F2 y verá otra sesion *MINIX*. Logearse como *nobody*

Rta.:
Para realizar esto, no deslogueamos, utilizando el comando `exit` y cuando aparece el login en pantalla ingresamos `nobody`. Cuando se nos pide la contraseña, ingresamos `nobody`. Un nuevo shell se nos abre, con directorio personal `/tmp`.

3.11.3. ejecutar el comando `su.`

Rta.:
Ejecutamos `su`.

3.11.4. ¿Que le solicita ?

Rta.:
Se solicita el password del *root*.

3.11.5. ¿Sucede lo mismo que en 3.10.2? ¿Por qué?

Rta.:
En el item 3.10.2 no se pedía el password para acceder a *root*, dado que el comando se invocó desde una sesión de *root*.

3.12. `rm`
Suprima el archivo `/usr/<grupo>/passwd`

Rta.:
Para borrar el archivo `/usr/grupo/passwd` se debe invocar `rm` de la siguiente manera:

```
rm /usr/grupo/passwd
```

3.13. `ln`
Enlazar el archivo `/etc/passwd` a los siguientes archivos `/tmp/contra1 /tmp/contra2`
Hacer un `ls -l` para ver cuantos enlaces tiene `/etc/passwd`

Rta.:
Para crear dos enlaces (*duros*, puesto que Minix no soporta los llamados links *simbólicos*) al archivo `/etc/passwd` llamados `/tmp/contra1` y `/tmp/contra2`, ejecutamos dos veces el comando `ln` de la siguiente manera:

```
ln /etc/passwd /tmp/contra1
ln /etc/passwd /tmp/contra2
```

Para poder chequear cuántos enlaces tiene el archivo `\etc\passwd`, ejecutamos el siguiente comando:

```
ls -l /etc/passwd
```

La salida que pobtenemos es la siguiente:

```
-rw-r--r--  3 root     operator      395 Apr 24 12:18 passwd
```

El número 3 que observamos indica que hay 3 nombres que referencian a un mismo archivo en memoria, consecuentemente con ésto, si despues borrasemos el archivo `/etc/passwd`, lo que estaríamos borrando sería una referencia al archivo y no el archivo en sí. La manera de borrar el archivo físicamente es eliminando todas las referencias (o *links* duros) que posea.

3.14. `mkfs`
Genere un Filesystem MINIX en un diskette

Rta.:
Para crear un filesystem MINIX en un diskette se ejecuta el siguiente comando:

```
mkfs /dev/fd0 1440
```

Dicho diskette *formateado* podemos apreciarlo en el archivo `minix_fs_floppy`, ubicado en el directorio de instalación de *Bochs*.

3.15. `mount`

Montelo en el directorio `/mnt`

Presente los filesystems que tiene montados

Rta.:

Para montar el filesystem recién creado en el directorio `/mnt` se escribe:

```
mount /dev/fd0 /mnt
```

La salida es:

```
/dev/fd0 is read-write mounted on /mnt
```

3.16. `df`

Que espacio libre y ocupado tienen todos los filesystems montados? (En KBytes)

Rta.:

Para ver información sobre los filesystems montados en el sistema se invoca el comando `df` sin parámetros.

La salida del comando es:

```
Device   Inodes Inodes Inodes  Blocks Blocks Blocks Mounted  V Pr
 total  used   free    total  used   free     on
 -----  -----  -----   -----  -----  -----  -------  - --
 /dev/hd1   496    202    294    1480    335   1145  /        2 rw
 /dev/hd2 11680   2675   9005   70056  19945  50111  /usr     2 rw
 /dev/fd0   480      1    479    1440     35   1405  /mnt     2 rw
```

Por lo tanto, el espacio libre y ocupado en los filesystems montados en el sistema son:

Dispositivo	Espacio ocupado (en KB)	Espacio libre (en KB)
/dev/hd1	335	1145
/dev/hd2	19945	70056
/dev/fd0	35	1405

3.17. `ps`

3.17.1. Cuantos procesos de usuario tiene ejecutando?

Rta.:

Para obtener una lista de los procesos del usuario se invoca al comando `ps` sin parámetros:

```
ps
```

La salida obtenida es:

```
PID TTY  TIME CMD
125  co  0:00 -sh
147  c1  0:00 -sh
179  co  0:00 ps
```

Por lo tanto, hay 3 procesos de usuario corriendo actualmente.

3.17.2. Indique cuantos son del sistema.

Rta.:

Para obtener una lista de todos los procesos (usuario y sistema) se invoca al comando `ps` indicando como argumento `-x`:

La salida obtenida es:

```
PID TTY  TIME CMD
 0   ?  0:00 TTY
 0   ?  0:00 SCSI
 0   ?  0:00 WINCH
 0   ?  0:00 SYN_AL
 0   ?  3:32 IDLE
 0   ?  0:00 PRINTER
 0   ?  0:00 FLOPPY
 0   ?  0:00 MEMORY
 0   ?  0:00 CLOCK
 0   ?  0:00 SYS
 0   ?  0:00 HARDWAR
 0   ?  0:00 MM
 0   ?  0:00 FS
 1   ?  0:00 INIT
125  co  0:00 -sh
27   ?  0:00 update
147  c1  0:00 -sh
180  co  0:00 ps -x
```

Por lo tanto, la cantidad de procesos del sistema es:
cantidad total de procesos =
Cant. total de procesos - Cant. de procesos del usuario = 18 - 3 = 15

3.18. `umount`

3.18.1. Desmonte el Filesystem del directorio `/mnt`

Rta.:

Ejecutamos el comando `umount /dev/fd0`
La salida que obtenemos es:
`/dev/fd0 unmounted from /mnt`

3.18.2. Monte el Filesystem del diskette como read-only en el directorio `/mnt`

Rta.:

Ejecutamos el comando `mount /dev/fd0 /mnt -r`

Obtenemos el siguente mensaje:
`/dev/fd0 is read-only mounted on /mnt`

3.18.3. Desmonte el Filesystem del directorio `/mnt`

Rta.:
Idem 3.18.1

3.19. `fsck`
Chequee la consistencia de Filesystem del diskette

Rta.:
Para chequear la consistencia del filesystem creado en el item 3.14 se invoca:
`fsck /dev/fd0`
La salida obtenida es:

```
Checking zone map
Checking inode map
Checking inode list

blocksize =  1024        zonesize  =  1024

     0    Regular files
     1    Directory
     0    Block special files
     0    Character special files
   479    Free inodes
     0    Named pipes
     0    Symbolic links
  1405    Free zones
```

3.20. `dosdir`
Tome un diskette formateado en *DOS* con archivos y ejecute `dosdir a`
Ejecute los comandos necesarios para que funcione correctamente el comando anterior

Rta.:

Se debe configurar el archivo `bochsrc.bxrc` para que el floppy referencie a la diskettera (floppya: 1_44=a:, status=inserted)

El comando `dosdir` accede al dispositivo a traves de `/dev/dosA`, con lo cual se sdebe crear un link con ese nombre al dispositivo `/dev/fd0` `ln /dev/fd0 /dev/dosA`

Finalmente, ejecutamos `dosdir a` o `dosdir /dev/fd0`. La salida obtenida es:

```
ORGA2.ZIP
STRING.ZIP
INFORMA.TEX
```

3.21. `dosread`

Copie un archivo de texto desde un diskette *DOS* al directorio `/tmp`

Rta.:

`dosread a informe.txt > /tmp/informe.txt cat /tmp/itr.txt`

La salida de este último comando es el contenido en texto plano del archivo .txt

3.22. `doswrite`

Copie el archivo `/etc/passwd` al diskette *DOS*

Rta.:

`doswrite a passwd < /etc/passwd dosread a passwd`

La salida de este último comando es:

```
root:##root:0:0::/:/usr/bin/ash
daemon:*:1:1::/etc:
bin:##root:2:0:Binaries:/usr/src:/usr/bin/ash
uucp:*:5:5:UNIX to UNIX copy:/usr/spool/uucp:/usr/bin/uucico
news:*:6:6:Usenet news:/usr/spool/news:
ftp:*:7:7:Anonymous FTP:/usr/ftp:
nobody:8ubLIV6hCe/.U:9999:99::/tmp:
ast:*:8:3:Andrew S. Tanenbaum:/usr/ast:
user1:##user1:10:3:user1:/usr/user1
```

4. 2.4. Uso de STDIN, STDOUT, STDERR, PIPES

4.1. STDOUT

4.1.1. conserve en el archivo `/usr/<grupo>/fuentes.txt` la salida del comando `ls` que muestra todos los archivos del directorio `/usr/src` y de los subdirectorios bajo `/usr/src` **Rta.:**

`#ls {R /usr/srcd > /usr/grupoSisOp/fuentes.txt` El comando ls permite listar los directorios. el símbolo `>` permite redireccionar la salida estándar al archivo /usr/grupo/fuentes.txt. El modificador `-R` se utiliza para mostrar el contenido de los subdirectorios de manera **R**ecursiva.

4.1.2. Presente cuantas líneas, palabras y caracteres tiene `/usr/<grupo>/tmp/fuentes.txt`

Rta.:

`# wc /usr/grupoSisOp/fuentes.txt` El comando wc (word count) lista secuencialmente la cantidad de lineas, palabras y caracteres del archivo indicado.

La salida que se obtiene es:
`19 19 125 /usr/grupo/fuentes.txt`

Esto significa que el archivo tiene:

- 19 líneas;
- 19 palabras; y
- 125 caracteres

4.2. STDOUT

4.2.1. Agregue el contenido, ordenado alfabeticamente, del archivo `/etc/passwd` al final del archivo `/usr/<grupo>/fuentes.txt`

Rta.:
`# sort /etc/passwd >> /usr/grupo/fuentes.txt` El símbolo `>>` sirve para agregar la salida estándar al archivo `/usr/grupo/fuentes.txt` sin detruir su contenido previo.

4.2.2. Presente cuantas lineas, palabras y caracteres tiene `usr/<grupo>/fuentes.txt`

Rta.:
`# wc /usr/grupoSisOp/fuentes.txt` Ejecutando el mismo comando que en el punto 4.1.2 se obtiene: `1995 1924 17864 fuentes.txt`

4.3. STDIN

4.3.1. Genere un archivo llamado `/usr/<grupo>/hora.txt` usando el comando echo con el siguiente contenido:
2355

Rta.:
`# echo 2355 > /usr/grupo/hora.txt`

4.3.2. cambie la hora del sistema usando el archivo `/usr/<grupo>/hora.txt` generado en 4.3.1

Rta.:
`# date -q < /usr/grupo/hora.txt` La salida es la siguiente:

```
Please enter date: MMDDYYhhmmss. Then hit the RETURN key.
Mon Sep 4 23:55:00 MET DST 2006
```

4.3.3. Presente la fecha del sistema

Rta.:
`# date` Se utiliza el comando date sin parámetros y la salida es la siguiente:
`Mon Sep 4 23:56:19 MET DST 2006`

4.4. STDERR
Guarde el resultado de ejecutar el comando `dosdir k` en el archivo `/usr/<grupo>/error.txt.`
Muestre el contenido de `/usr/<grupo>/error.tmp`

Rta.:

`#dosdir k 2> /usr/grupoSisOp/error.txt` donde el modificador `'2>'` indica al *shell* que debe redireccionarse la salida estándar (STDERR) de error en vez de a la salida estándar normal. `cat /usr/grupoSisOp/error.txt` Se utiliza el comando `cat` para poder vel el contenido del archivo direccionado. El resultado es: `cat: cannot open error: No such file  or directory`

4.5. PIPES Posiciónese en el directorio / (directorio raíz), una vez que haya hecho eso:

4.5.1. Liste en forma amplia los archivos del directorio `/usr/bin` que comiencen con la letra s. Del resultado obtenido, seleccione las líneas que contienen el texto *sync* e informe la cantidad de caracteres, palabras y líneas.
Nota 1: Está prohibido, en este ítem, usar archivos temporales de trabajo
Nota 2: si le da error, es por falta de memoria, cierre el proceso de la otra sesion, haga un `kill` sobre los procesos *update* y *getty*.

Rta.:
El comando es el siguiente:
`ls -l /usr/bin/s* | grep sync | wc`

donde el caracter `*` indica que luego de la letra "s" puede haber cualquier caracter. Ese resultado le es enviado al comando `grep sync` mediante el operador `|` (pipe) y el resultado de esa "búsqueda", será enviado mediante otro *pipe* al comando `wc`. La salida obtenida es:
`2 18 140`

Siendo:

- Cantidad de líneas: 2
- Cantidad de palabras: 18
- Cantidad de caracteres: 140

5. 2.5. Administrador de Usuarios

Diseñar y programar un utilitario para el administrador del sistema, que permita crear, borrar o modificar en forma automática cuentas de usuarios. De forma interactiva pedirá todos los parámetros necesarios del nuevo usuario. Automáticamente comprobará y validará el identificador del nuevo usuario. Creará en caso de

que no exista, su directorio asociado o directorio de trabajo, haciéndolo su propietario. Asignará al nuevo usuario un shell particular. Modificará los archivos group y shadow con los nuevos valores. En caso que el grupo no exista deberá agregarlo. Deberá agregar el usuario al grupo El borrado de un usuario llevará consigo la desaparición de todos sus archivos y directorios asociados y la modificación de los archivos group y shadow. En todos los casos deberá controlar la consistencia de la información.

Rta.:
Para la efectiva realización del script requerido se tomó la decisión de "modularizarlo", creando así un script que permitiera agregar usuarios, otro que los elimine y, finalmente, uno que una a estos dos últimos.

agregarUsuario

Para este script debimos, en principio, crear un grupo al que asignar a los usuarios. Decidimos entonces crear el grupo `users` y asignarle el siguiente número de grupo –9 en este caso–. Una vez hecho esto, realizamos varias pruebas con el comando `adduser`. Como se habló en el punto 3, `adduser` solicita como parámetros de entrada el nombre del usuario a crear, el grupo al que este usuario va a pertenecer y la dirección de su carpeta personal. Luego, automáticamente agrega al usuario a los archivos `/etc/passwd` y `/etc/shadow`, crea su carpeta personal y cede la propiedad de la misma al nuevo usuario.

Tras algunas pruebas con nombres de usuario de longitud variable y directorios personales en lugares estrafalarios, se descubrieron los siguientes restricciones a la hora de crear un usuario:

- El nombre de usuario no puede tener más de ocho caracteres
- Todos estos deben ser alfanuméricos
- El directorio personal siempre tendrá que estar dentro de `/usr` para que el comando funcione correctamente

Para mejorar la organización de los usuarios, se decidió crear el directorio `/usr/home` y colocar allí dentro los directorios de los usuarios.

Dado que se tenían en cuenta estas consideraciones, se comenzó a programar el script en sí.

Primero se le solicita al usuario que ingrese un nombre para el nuevo usuario. Luego se realiza un proceso de verificación de la existencia del nombre de usuario ingresado y la longitud del mismo. Para lo primero se realiza una llamada a `grep "^$userName"+: /etc/passwd`. Esto se debe a que los usuarios aparecen en el archivo `/etc/passwd` de la siguiente manera:
`nombre:##nombre:numero_de_usuario:numero_de_grupo:profile:directorio_personal`

Por lo que `grep` realizará una búsqueda de las palabras que comiencen con `nombre del usuario:`.

Para determinar la cantidad de caracteres ingresados se utiliza la secuencia de comandos `echo "$userName || wc -c"`. Esto hace que se cuente la cantidad de caracteres. Desafortunadamente siempre cuenta un caracter de más. Nuestra suposición es que debe contar también al caracter de finalización de string de *C*, el caracter \0. Este error es fácilmente solucionable, simplemente se le resta un 1 a la cantidad de caracteres que `wc` cuenta.

Una vez elegido un nombre de usuario que cumpla con los requisitos, el script llama a la función `adduser $userName users /usr/home/$userName`. Con esto se crea al usuario con el nombre ingresado, se lo agrega al grupo `users` y se agrega un directorio con su nombre a `/usr/home`, del que se le da propiedad. Dado que el comando imprime todo esto en pantalla, se decidió redirigir la salida estándar a `/dev/null`.

El script llama luego a `passwd $userName` para agregar la contraseña del usuario.

eliminarUsuario

Este script fue más complicado de implementar que el anterior pues no existe un comando que elimine usuarios en Minix. Para borrar una cuenta de usuario es necesario deshacerse de todo rastro del mismo. Esto es, eliminar sus archivos y carpetas personales y su aparición en los archivos `/etc/passwd` y `/etc/shadow`. Lo primero se logra sencillamente usando el comando `rm -fR` –esto es, eliminar carpetas y los archivos que contengan sin preguntar–. Para eliminar las apariciones de los usuarios de los archivos antes mencionados, se utilizó el comando `grep -v "^$userName"+: /usr/passwd` *o* `/usr/shadow`. Esta aplicación de `grep` devuelve todas las líneas del archivo en cuestión en las que NO aparezca la expresión a buscar –en este caso, el nombre del usuario seguido de un ':'–. En un primer momento se consideró redirigir la salida del `grep` al mismo archivo que se estuviera analizando, pero luego se tomó en cuenta que, dado el funcionamiento de la redirección del `STDOUT`, lo primero que se haría sería eliminar el contendio de `/etc/passwd` y/o `/etc/shadow`, creando así considerables problemas[2]. Por ende, se decidió redirigir la salida estándar a un nuevo archivo y luego renombrarlo por `/etc/passwd` o `/etc/shadow`, según corresponda.

Cabe aclarar que, antes de realizar el procedimiento, el script verifica que el nombre de usuario pasado como parámetro de hecho exista, corroborando su aparición en `/etc/passwd`. Tampoco permita que se intente eliminar a `root`.

userManager

[2]Básicamente, MINIX deja de funcionar. Lo intentamos en una imagen de disco backupeada.

El script que solicita el enunciado se compone, a grandes razgos, de un case de tres ítems principales –y un cuarto para salida del sistema– dentro de un loop.

- Agregar un nuevo usuario
- Modificar los datos de un usuario existente
- Eliminar a un usuario existente

El primero y el tercer ítem realizan las mismas comprobaciones de coherencia explicadas en las secciones anteriores, con la única diferencia de que se ejecutan en un ciclo.

El segundo ítem ofrece el listado de los usuarios existentes mediante una llamada a `grep ":9:/etc/passwd`. Dado que el número del grupo `users` fue arbitrariamente elegido por nosotros, simplemente realizamos una búsqueda de las líneas del archivo `/etc/passwd` en las que figure dicho número para que nos traiga un listado completo de los usuarios[3]. Luego, solicita un nombre de usuario, realiza la misma verificación que se realiza antes `eliminarUsuario` y, luego, brinda otras tres opciones para modificar:

- Contraseña
- Nombre completo del usuario
- Shell

Para modificar la contraseña simplemente se llama al comando `passwd $userName`.

Para cambiar el nombre completo del usuario (el que aparece en su profile), se llama al comando `chfn` –change full name–, que solicita el nombre del usuario y el nombre que se le quiere asignar.

Finalmente, en la tercera opción se ofrecen los dos Shells que vienen integrados en MINIX: **MINIX Shell** (`/usr/bin/sh`) y **Alchemist Shell** (`/usr/bin/ash`). Una vez elegido uno de ellos, se le aplica al usuario con el comando `chsh`.

Nota: Los Scripts auxiliares agregarUsuario y eliminarUsuario solo deben ser llamados desde el script principal userManager, ya de ejecutarlos por separado podrían producir errores como por ejemplo el borrado completo del archivo passwd

6. 2.6. Ejecución de procesos en Background

Crear el siguiente programa

[3]Este método fue luego adoptado a la hora de realizar la verificación de usuarios antes de eliminarlos para evitar tener que salir del sistema para obtener este mismo listado.

```
/usr/src/loop.c
#include <stdio.h>
int main(){
int i, c;
while(1){
c = 48 + i;
printf("%d",c);
i++;
i = i % idgrupo;
}
}
```

Compilarlo. El programa compilado debe llamarse *loop*, Indicando a la macro idgrupo el valor de su grupo.
a) Correrlo en foreground. ¿Que sucede ? Mate el proceso con el comando `kill`

Rta.:
Escribimos el codigo fuente con el `mined`, luego lo compilamos con el `cc` utilizando la siguiente sintáxis:
`cc loop.c -Didgrupo=20 -o loop`
A continuación, corremos el archivo y vemos que el ciclo del programa nunca termina, y en cada iteración muestra un valor por pantalla. Por lo tanto, al ejecutar el programa en foreground el usuario pierde el control de la consola. Para volver a obtener el control de la misma, presionamos *ALT+F2*. Con esto tenemos acceso a una nueva consola.

Después de loguear, pedimos la lista de procesos con el comando `ps`, identificamos el *PID* del proceso `loop` y ejecutamos `kill` pasando como parámetro dicho identificador. De esta forma, matamos el proceso infinito y volvemos a obtener el control de la primera consola.

b) Ahora ejecútelo en background
`/usr/src/loop > /dev/null &`
¿Que se muestra en la pantalla?
¿Que sucede si presiona la tecla F1? Que significan esos datos?
¿Que sucede si presiona la tecla F2? Que significan esos datos?

Rta.:
b. La salida estándar se redirecciona al dispositivo nulo (una especie de *agujero negro*, utilizable cuando deseamos descartar la salida). Por lo tanto, por pantalla no observamos los números que se escriben en el ciclo. Sin embargo, se muestra un número que representa el PID del proceso que queda ejecutándose en background (producto de haber invocado el programa con un & al final).

Presionando la tecla F1 se obtiene un vuelco de la tabla de procesos activos:

```
--pid --pc- ---sp- flag -user --sys-- -text- -data- -size- -recv- command
  -11  ccc9   8ac4  0     928       0     2K    54K   100K        TTY
  -10  ccc9   8e98  8       0       0     2K    54K   100K ANY    SCSI
   -9  ccc9   929c  8       8       0     2K    54K   100K ANY    WINCH
   -8  ccc9   94c0  8       0       0     2K    54K   100K CLOCK  SYN_AL
   -7   547   9530  0  691127     321     2K    54K   100K        IDLE
   -6  ccc9   96f4  8       0       0     2K    54K   100K ANY    PRINTER
   -5  ccc9   9cd0  8       0       0     2K    54K   100K ANY    FLOPPY
   -4  ccc9   9ed4  8      19       0     2K    54K   100K ANY    MEMORY
   -3  ccc9   a114  8       1       0     2K    54K   100K ANY    CLOCK
   -2  ccc9   a314  8      28       0     2K    54K   100K ANY    SYS
   -1     0   a330  0     470       0     2K    54K   100K        HARDWAR
    0  2c49   7cc8  8       5       0  1024K  1036K    44K ANY    MM
    0  6fa5  1b6c4  8      93       0  1068K  1096K   138K ANY    FS
    1  17ed   2a34  8       0       1  1206K  1206K    11K MM     INIT
   32  ea85  13030  8       1       8   142K   204K   139K FS     ash
   74   2ca  2117c  0     119       2   307K   307K   134K        loop
```

Presionando la tecla F2 se obtiene un vuelco de ???:

```
PROC NAME-  -----TEXT-----  -----DATA-----  ----STACK-----  -SIZE-
 -1 HARDWA     0    8   d0     0   d8   c0    c0  198    0    100K
  0 MM         0 1000   31     0 1031   7d    7d 10ae    0     44K
  1 FS         0 10ae   72     0 1120  1b7   1b7 12d7    0    138K
  2 INIT       0 12d7    0     0 12d7   2b    2b 1302    0     11K
  3 ash        0  2a2   f9     0  39b   40   130  4cb    2    139K
  4 update     0  198    6     0  19e    1     f  1ad    2      6K
  5 ash        0  2a2  f9     0  6e4   40   130  814    2    139K
  6 loop       0  4cd    0     0  4cd   20   210  6cc    7    134K
```

- pid: Identificador del proceso
- pc: Program Counter, o registro de próxima instrucción
- sp: Stack Pointer, puntero al stack del proceso

7. 2.7. Compilación del kernel de minix

- a) Antes de generar una nueva versión del minix, prodeceremos a realizar un resguardo del minix original para poder restaurar el sistema en caso de falla. Para ello el comando *man boot* en la linea de comandos da información de los procedimientos que realiza minix para iniciarse. En primer lugar carga el

Monitor de Booteo de Minix desde `\boot` luego el Monitor carga los archivos binarios de minix desde `/minix` ó si e'ste es un directorio, la imagen mas reciente que se encuentre e él. (En general la imagen lleva el nombre de la versión 2.0.0)

Para realizar un resguardo se crea la carpeta /minix2 mediante el comando
`mkdir /minix2`
a continuación se copia la imagen del minix:
`cp /minix/* /minix2`
y se elimina la imagen previa:
`rm /minix/2.0.0`

Se modifica a continuación el archivo `/usr/include/minix/config.h` para agregarle las lineas que diferenciarán el booteo original del que se generará (Esto se indica en el enunciado del Tp).

Para recompilar el kernel se debe acceder al directorio:
`cd /usr/src/tools`
y utilizando el Makefile ejecutar
`make hdboot`

Esto realizará las siguientes acciones:

- Compila el kernel.
- Compila el file system.
- Compila el memory manager.
- Construye una imagen de booteo a partir de 1), 2) y 3).
- Copia la imagen al directorio /minix.

b) Para construir un nuevo diskette boot se debe ejecutar:
`cd /usr/src/tools make fdboot`
La salida es la siguiente:

```
cd ../kernel && exec make -
make: 'kernel' is up to date
cd ../mm && exec make -
make: 'mm' is up to date
cd ../fs && exec make -
make: 'fs' is up to date
make: 'image' is up to date
exec su root mkboot fdboot
Finish the name of the floppy device to write
```

```
(by default 'fd0'): /dev/fd0
/dev/fd0 is read-write mounted on /mnt
/dev/fd0 unmounted from /mnt
/usr/mdec/bootblock and 2 addresses to boot patched
into /dev/fd0
Test kernel installed on /dev/fd0 with boot parameters from
/dev/hd1
```

El comando `make` se encarga de recompilar los archivos no volviendo a compilar los que no se han modificado, ahorrando tiempo. En un determinado momento de la ejecución se pide el nombre del dispositivo de diskette (`/dev/fd0`) y a continuación escribe en él (emulado a través de la configuración del Bosch al archivo `floppya`).

En el archivo de configuración de Bochs (`bochsrc.bxrc`), se cambia el valor de la variable `boot` para que realice el *boot* desde el diskette creado (modificando también la línea
`floppya: 1_44=".\floppya", status=inserted`).

Se reinicia la máquina virtual y se ve como el sistema arranca desde el diskette satisfactoriamente.

8. 2.8. Características de Minix

Escriba un informe explicando que tipo de mecanismo utiliza MINIX para: a) Administrar el recurso procesador (scheduler). b) Administrar el recurso memoria. c) Administrar las operaciones de Entrada/Sa lida d) Administrar la información en disco (File System)

//

Rta.:

Administrador de Procesos

Veamos el scheduler de Minix:

En el caso de minix (sistema operativo con multiprogramación) lo que lo mantiene funcionando es el sistema de interrupciones. Los procesos se bloquean cuando solicitan una entrada, permitiendo mientras tanto que otros procesos ejecuten, una vez que están disponibles las entradas (teclado, disco, etc.) interrumpe al proceso que se esta ejecutando. El reloj genera también interrupciones, esto es para que un proceso que no abandone el estado de ejecutando por voluntad propia (E/S) pueda abandonar el procesador a otro en su lugar para que tenga oportunidad de

ejecutarse.

Cada vez que un proceso es interrumpido ya sea por propia voluntad o por reloj, nace la necesidad de determinar cual de todos lo procesos en espera será el elegido, cabe aclarar que esto también sucede cuando un proceso abandona el recurso procesador una vez que termina. El planificador de Minix usa un sistema de multicolas con tres colas, que corresponden a las capas 2, 3 y 4 de la figura "FIGURA 1: Layaut de Minix"', para las capas 2 y 3 los procesos se ejecutan hasta que terminan pero para el nivel 4 (procesos de usuarios) se utiliza una planificación Round-Robin. Las tareas tienen la prioridad más alta, les siguen el administrador de memoria y el servicio de archivos, y los procesos de usuarios vienen al final.

Cuando un proceso se escoge para ejecutarlo, el planificador verifica si hay una tarea lista, en el caso de existir mas de una elegirá la que se encuentre a la cabeza de la cola, si no existen tareas listas se escoge un servidor (MM o FS) si esta cola también estuviese vacía se elegirá un proceso de usuario.

Antes comentamos que los procesos de usuarios no ejecutan indiscriminadamente hasta que terminen sino que tiene un Quantum de desalojo, este quantum es de 100ms, por eso cuando se encuentra un proceso que cruzo ese limite se llama al planificador de procesos para ver si hay otro proceso a la espera de la CPU, si existe tal proceso, se envía al final de cola (que le corresponde) el proceso que estaba ejecutando y se pasa a ejecutar el proceso q se encontró en el paso antes mencionado. Recordemos que los procesos de memoria, las tareas y el sistema de archivos nunca son desalojados por el quantum (reloj).

Administrador de Memoria

A diferencia de otros sistemas operativos más modernos, Minix utiliza una administración muy simple para la memoria, no utiliza paginación ni segmentación. Por el contrario Minix utiliza la técnica de "'partición variable"'manteniendo una lista de agujeros ordenados en por dirección de memoria. Cuando se necesita memoria probablemente por una llamada al sistema, se examina la lista de agujeros utilizando "'primer ajuste"én busca de un lugar del tamaño suficiente. Una vez que el proceso fue colocado en su lugar permanece allí hasta terminar, o sea nunca se intercambia a disco y nunca se traslada a otro lugar de la memoria, además el área asignada no crece ni se encoge. Antes de explicar el porque de esta elección no debemos olvidar que Minix fue desarrollado en sus primeras versiones en el año 1986 (minix 2.0, 1996) y siempre tuvo como premisa poder correr en procesadores que van desde 8088 hasta Pentium contemplando memorias RAM de 2 mega. Y

sabiendo que hasta hace unos años atrás los programas de uso hogareños no requerían de grandes espacios en memoria, teniendo en cuenta todo esto podremos entender mejor las causas de la elección de esta administración de memoria. Son tres los factores fundamentales:

1) La idea de que Minix es para computadoras personales, no para sistemas grandes de tiempo compartido. Este factor provine de pensar que en, promedio, el numero de procesos en ejecución será pequeño y por lo tanto habrá regularmente suficiente memoria para contener a todos los procesos incluso con espacio de sobra, por esto último es que no se usa reemplazo, cuestión que haría mucho más compleja la implementación.

2) La necesidad de que Minix corriera en cualquier IBM PC Esta razón se deduce de la idea de que muchos procesadores de la familia del 8088 por Ej., no soportan memoria virtual, por tener una arquitectura de administración muy primitiva y por más que procesadores superiores como el 80386/80486 o Pentium puedan manejar memoria virtual, si Minix la implementara perdería la compatibilidad con procesadores anteriores.

3) El deseo de facilitar la implementación del sistema en otras computadoras pequeñas.

Este ultimo punto esta ligado a la posibilidad de que si Minix cuenta con una implementación sencilla que se podría transportar, ya que si utilizara segmentación o paginación seria casi imposible trasladarlo a maquinas que no contaran con estas capacidades. Esto mismo es relacionado también con el punto 2 antes comentado.

Y con respecto a la implementación hay también otro detalle inusual. El manejo de memoria no es parte del Kernel, sino que corre por cuenta del proceso de administración de memoria, que se ejecuta en el espacio de usuario y se comunica con el Kernel por el mecanismo de mensajes estándar. Dentro del esquema de capas descripto en el análisis de procesos, el proceso de memoria se ubica en la capa 3 de servidores. El tener la administración de memoria separada del Kernel permite que sea relativamente fácil modificar la política de administración de memoria (algoritmos, etc.), sin tener que alterar las capas inferiores del sistema operativo.

Administración de operaciones de E/S

A la hora del manejo de una acción de entrada / salida, podríamos nombrar a dos entidades que trabajan para ello como las importantes, se trata de los controladores de dispositivos (DRIVERS) y por otro lado el manejador de interrupciones. Estos dos elementos en muchos caos trabajan en conjunto, en muchos casos, por Ej., los controladores inician algún dispositivo de E/S y luego se bloquean en espera de recibir un mensaje que por lo general es emitido por el manejador de interrupciones que corresponde al dispositivo en juego. Por otro lado existen controladores que no interactúan directamente con el manejador de interrupciones, podríamos decir que gran parte del trabajo se delega al hardware sin utilizar la recepción de mensajes ni interrupciones.
A continuación mostramos las capas que participan en la interaccion de una E/S y que tarea desempeña cada una:

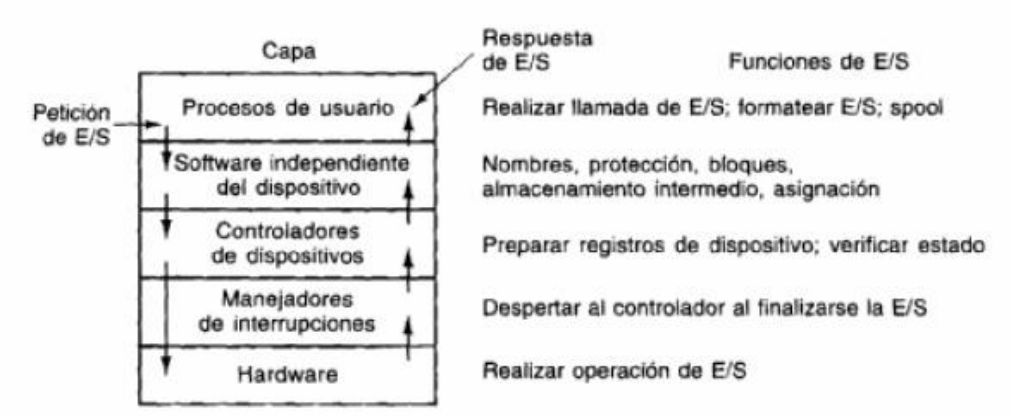

Figura 2: Comunicacion entre capas entrada/salida

En el caso de los dispositivos de disco es el controlador en hardware quien realiza todo el trabajo, lo que se hace es ordenar a un dispositivo a que realice la E/S y sea guarda luego a su finalización y en este caso se le exige muy poco al manejador de interrupciones.

Existen por otro lado casos en los que el manejador de instrucciones de bajo nivel tiene mas trabajo que hacer, imaginemos un dispositivo que por sus propias características reciba constantes pedidos de uso con sus consecuentes interrupciones, en este caso será notable el costo que comenzaran a tener el envió de mensajes, para solucionara este problema lo que se hace es que se pospongan el envió de los mensajes hasta la interrupción subsecuente cuando el dispositivo tenga realmente que hacer algo mas, este es el modo en el que MINIX maneja las interrupciones del reloj. La idea de esta forma de trabajo es que el manejador de interrupciones

va llevando cuenta en variables internas, de las interrupciones recibidas (de reloj por ejemplo) realizando un análisis previo y enviando un mensaje solo cuando hay alguna tarea real y concreta para hacer, por Ej. si se esta levando cuenta de la hora actual se puede estar analizando si ese momento coincide con el de una alarma o la ejecución de un nuevo proceso y de ser cierto recién allí se envía el m mensaje.
Otro ejemplo de este tipo de mecanismo es el teclado, que por sus propias características posee una cantidad de E/S baja en contraposición a la cantidad de interrupciones recibidas de su parte, básicamente lo que el hace manejador es despreciar los datos que pueden ignorarse porque no afectaran los datos concretos a tener en cuenta.

Los dispositivos del tipo RS-232 también se manejan de forma similar, donde por cada carácter que se envía no se procesa por si solo sino que se acumulan y se envían a procesar como respuesta a un solo mensaje.

Con respecto a los controladores de dispositivos, podemos decir que MINIX maneja uno particular para cada tipo de dispositivo estos son sencillamente procesos como cualquier otros que presentan una diferencia sustancial del resto de los procesos de usuarios y que se inician junto con la carga del SO en memoria, o sea se cargan junto con el kernel, por lo tanto todos los controladores compartirán un mismo espacio de direccionamiento lo que les permitirá tener una mejor comunicación entre ellos de ser necesario e inclusive si manejan un procedimiento común a ellos permite cargar este procedimiento una sola vez y solo hacer un enlace a cada uno con el.

Este diseño es altamente modular y es una de las principales diferencias con Unxis, veamos un esquema sobre como se intercomunican los procesos con las diferentes partes que entran en juego a la hora de una E/S utilizando diferentes estructuras.

Minix tiene un enfoque mucho mas modular (Figura a) como dijimos a diferencia de Unís (figura b), la ventaja para Minix es la de poder extenderse fácilmente a sistemas distribuidos en los que muchos procesos se ejecutan el diferentes computadoras.

Con respecto a la parte de software de E/S, podemos comentar que en Minix el proceso del sistema de archivos contiene todo el código de E/S independiente del dispositivo. El sistema de E/S esta íntimamente ligado con el sistema de archivos que se fusionaron para formar un solo proceso.

Por otro lado también existe software a nivel usuario, existen bibliotecas que lo

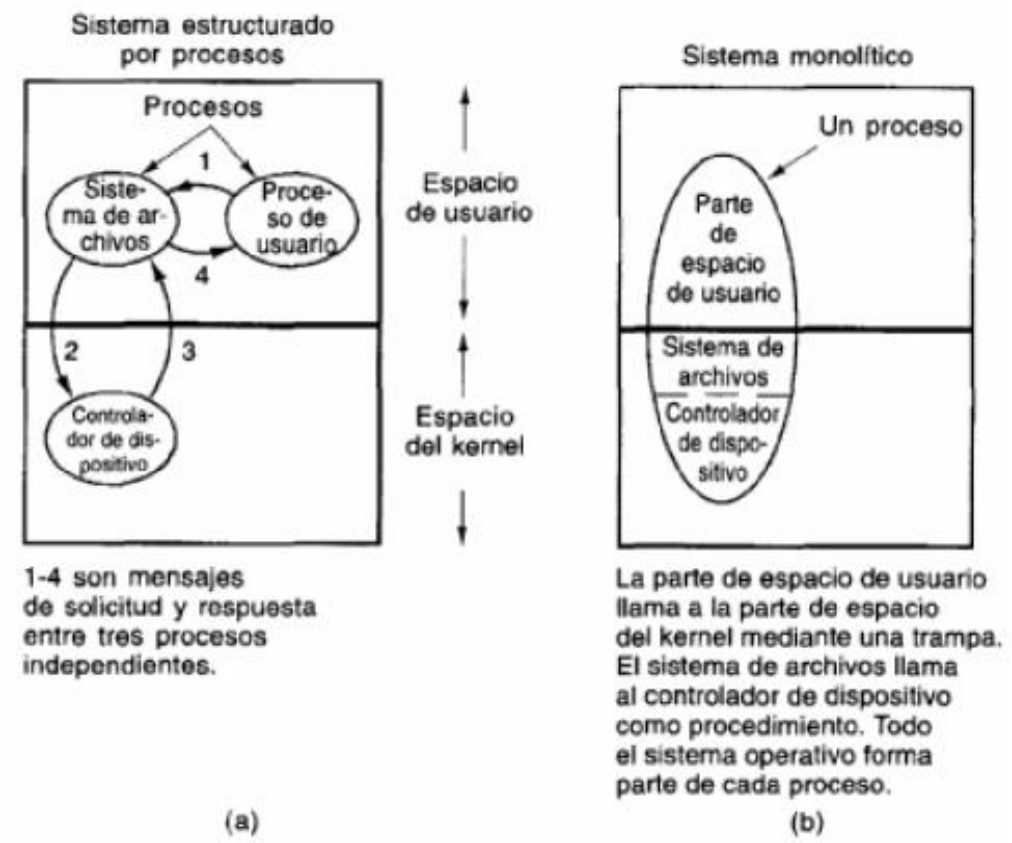

Figura 3: Estructura de entrada/salida

manejan, Minix tiene un sistema de spool por ejemplo.

- Administración de la Información
- Sistema de Archivos (File System)

Para describir como Minix maneja el sistema de archivos comenzaremos por detallar las tareas principales que todo sistema operativo debe tener en cuenta para lograr un correcto de la administración de la información, estas tareas serian:

- Asignar y Liberar espacio para los archivos
- Tener el camino o ruta a los bloques de disco y espacio libre
- Manejar correctamente la protección de los archivos contra agentes no autorizados.

Comenzaremos por decir que el sistema de archivos de Minix no es ni más ni menos que un gran programa (escrito con el lenguaje C) que se ejecuta en el espacio del usuario. A su vez para que los procesos puedan leer o escribir se comunican con el sistema de archivos mediante mensajes que indican la tarea a realizar estos son atendidos por el y luego el sistema devuelve una respuesta. Esta forma de trabajar nos muestra como el sistema archivos juega el papel de un servidor de archivos en red que en este caso y solo de forma casual se ejecuta en la misma maquina que lo invoco. Por otro lado manejar el file system de esta manera implica dos aspectos importantes

1- Se podría modificar el sistema de archivos con total independencia del resto de SO. 2- Trasladar el sistema de archivos a otra maquina no seria problema, mientras esta tenga un compilador de C se podría compilar allí y utilizarlo como servidor de archivos remoto autónomo. Lo único q se alteraría y deberíamos modificar seria el área del mecanismo de envió de mensajes.

Veamos ahora la organización de Sistema de Archivos:

El File System en Minix es una entidad lógica y autónoma que contiene i-nodos, directorios y bloques de datos, vale decir que esta estructura puede almacenarse en cualquier dispositivo por bloques (discos flexibles por Ej.). Todo sistema de archivos posee un primer bloque de arranque donde se encuentra código ejecutable. Cuando se enciende la computadora, el hardware lee el bloque de arranque del dispositivo de arranque (en general el disco duro) y lo coloca en memoria y comienza a ser ejecutado desde allí. Es esta el código que inicia la carga del mismo SO, luego de realizar su tarea este bloque no es vuelto a utilizar durante

la computadora permanezca encendida. En la siguiente figura mostraremos la estructura organizacional de un disco sencillo.

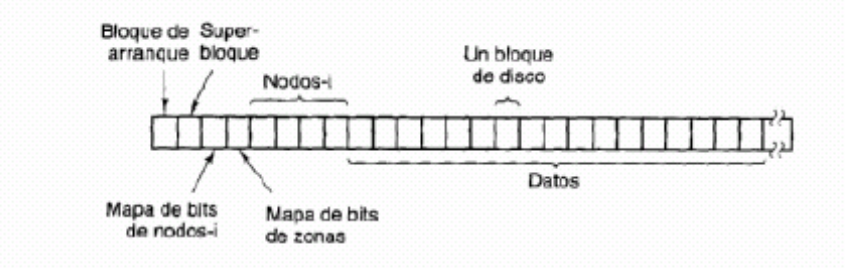

Figura 5-28. Organización del disco más sencillo: un disquete de 360K con 128 nodos-i y bloques de 1K (es decir, dos sectores consecutivos de 512 bytes se tratan como un solo bloque).

Figura 4: Organización del disco

Cuando Minix es cargado en una computadora, lo que se carga en una tabla a memoria inmediatamente es lo que se llama superbloque, este contiene la información que describe la organización del sistema de archivos y su función principal es la de indicarle al file system el tamaño de sus diversos componentes, también contiene información sobre los permisos sobre los archivos o bien si se esta utilizando alguna convención no estándar para el orden de bytes.

Es importante aclarar que antes de que un dispositivo disco sea usado como sistema de archivos en Minix, se le debe dar un formato en particular, para este fin se utiliza el comando "mkfs", este programa dirá a la unidad de disco la cantidad de bloques y tamaño de cada uno. Con este comando también se carga lo que se denomina numero "mágico.[en] el superbloque, que se utiliza para detectar si se esta utilizando un sistema de archivos validos para Minix, este permite también el no cargar basura a la hora de la carga del bloque de arranque Antes mencionamos que una parte del sistema de archivos es el i-nodo, este permite cargar la tabla inode, que es la que contiene información que no esta en el disco, por lo que es traída a memoria y permanece allí hasta que se cierra el archivo. La principal función del i-node de un archivo es indicar donde están los bloques de datos, también contiene información de modo, que indica el tipo de archivo de que se trata y da los bits de protección y de SETUID y SETGID, además se e encuentra el campo link del i-nodo registra el numero de entradas de directorio que apuntan al i-nodo y le permite al sistema de archivos saber cuando se debe liberar el espacio que

ocupa el archivo.

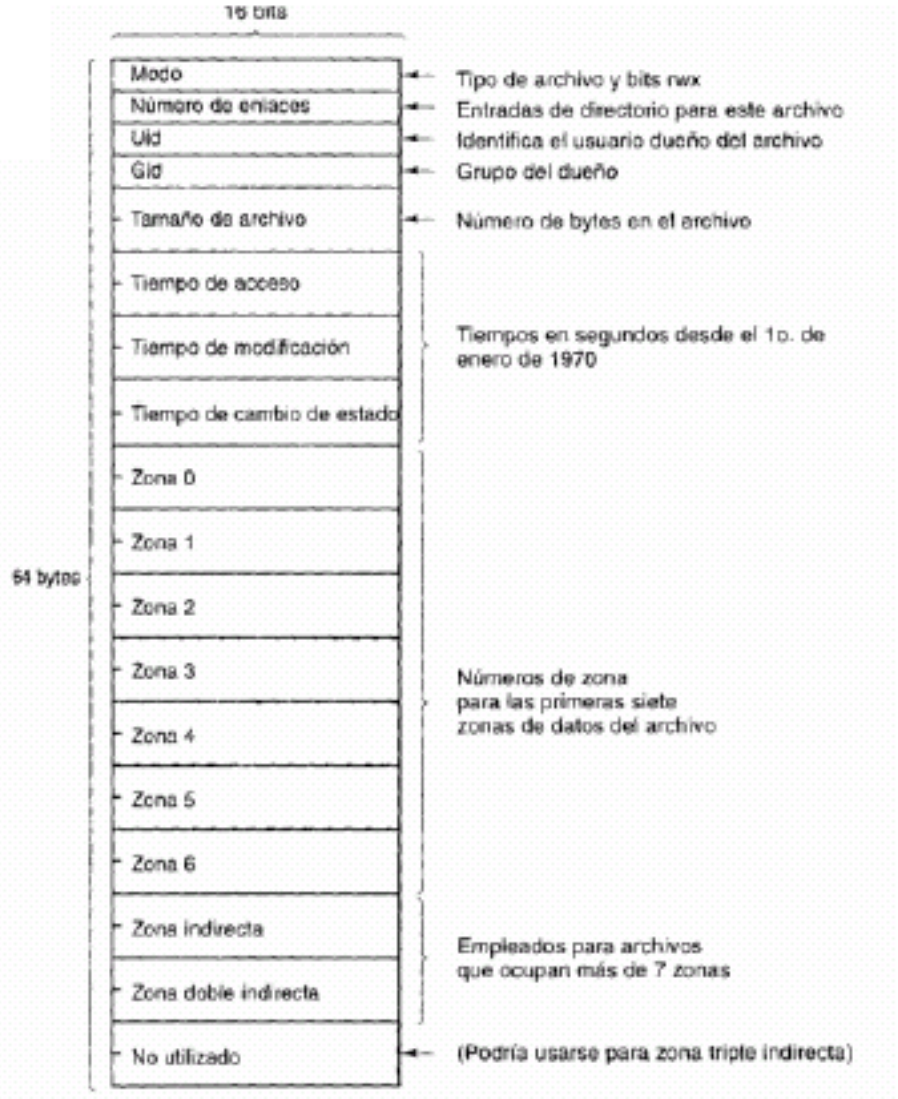

Figura 5: Tabla de i-nodo

Minix también cuenta con una cache de bloques, la cual logra un mejor rendimiento del sistemas de archivos Esta cache esta armada simplemente como un arreglo de buffers, cada uno conteniendo apuntadores, contadores y banderas, y con un cuerpo para un bloque de disco, y para administra estos buffers se utilizan las técnicas de (MRU) el mas recientemente utilizado hasta el menos recientemente utilizado (LRU). Además se usa una tabal de Hash para saber rápidamente si un bloque de disco esta en cache o no. Aquí se puede observar un diagrama que describe el funcionamiento de la cache.

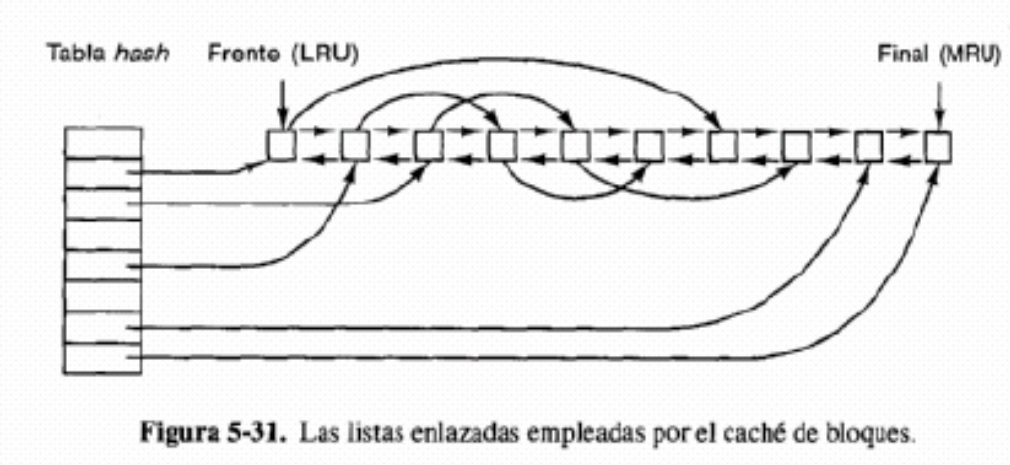

Figura 5-31. Las listas enlazadas empleadas por el caché de bloques.

Figura 6: Cache

Otro aspecto importante para lograr el manejo de archivos es la administración de directorios y nombres de ruta, hay un subsistema dedicado a este ítem en Minix. Cuando se ejecuta el comando open y se le da el nombre del archivo al mismo lo que se usara realmente es el i-nodo de ese archivo, por lo tanto es el sistema de archivos quien busca en el árbol de directorios y localiza el i-nodo. Un directorio para Minix consta de entradas de 16 bytes, de las cuales los dos primeros bytes son precisamente el i-nodo y el resto so explícitamente el nombre del archivo.

Por otro lado tenemos la necesidad de utilizar descriptores de archivos, lo que implementa Minix en este caso es devolver un descriptor de archivo al proceso que lo invoco para que lo pueda usar en los subsecuentes READ y WRITE. Para administrar dichos descriptores, el sistema de archivos mantiene dentro de su espacio de direcciones parte de la tabla de procesos, en particular hay tres campos que tiene importante interés, los primeros dos son apuntadores a los nodos-i del directorio raíz y del directorio de trabajo, las búsquedas de rutas siempre comenzaran en uno o en otro. Y el tercer campo es un arreglo indicado por numero de descriptor de archivo. y se utiliza para localizar el archivo correcto cuando se presenta un descriptor de archivo. Con esta táctica surge un problema a la hora de compartir archivos exitosamente, la manera que Minix implementa es la de introducir una tabla extra compartida que contiene todas las posiciones en los archivos, la idea es compartir realmente la posición de un archivo para que no ocurran incongruencias al manejar apuntadores a la posición de un mismo archivo para distintos procesos. Minix también utiliza un nueva tabla para manejar aspectos de seguridad de archivos, se trata de la tabla de candados de archivos,

llamada file_lock en esta tabla están todos los candados, y en cada entrada de la tabla tiene espacio para guardar el tipo de candado que tiene dicho archivo, por Ej. si es contra lectura o escritura, y es el identificador del proceso quien posee el candado mas un apuntador al i-nodo del archivo asegurado y por ultimo las distancias a las que están el primer y ultimo Byte de la línea asegurada.

9. 2.9. Implementación de un Driver

Describa en un informe explicando como se genera un driver de dispositivo tipo bloque y como se realiza la vinculación con el sistema de archivos en MINIX. Además describa la estructura del procedimiento de una tarea de E/S (I/O task). Sugerencias: Como ayuda busque las respuestas a las siguientes preguntas: Qué es el "major number" de un device. (com.h) Dónde se declara la entrada a un driver. (table.c) Cómo se llega desde un proceso usuario al driver. (table.c)

Rta.:

Los bloques son archivos especiales que pueden ser de varios tipos, a continuación recordaremos algunos de los comentarios expresados en el ejercicio 2.
Los bloques se crean con el comando el mknod, este comando requiere además de el pasaje de tres parametros, entre ellos el Mayor number y el Menor number, estos hacen referencia a l número que tiene asignado el driver encargado de un cierto tipo de periferico, y la instancia de ese dispositivo respectivamente. Todos los perifericos del mismo tipo poseen el mismo Mayor number ya que estan siendo manejados por un mismo driver.
Ejemplo:

```
mknod MiDriver b 15 1
```

Donde *MiDriver* indica el nombre del archivo especial, b indica que se trata de un archivo de bloque, 15 es el Mayor number, 1 es el Menor number que ya se definieron anteriormente.

A continuación explicaremos los pasos a seguir para la instación de un nuevo DRIVER. En primer lugar deben realizarse las modificaciones que permitan que el driver pueda interactuar con el Sistema Operativo para lograr la manipulación del periferico al que pertenece, para lograrlo se deberá alterar el Kernel y el FileSystem, más precisamente se modificará el *table.c* y el *Makefile* del Kernel y los archivos, *com.h*, *config.h*, *const.h*, *proto.h* y *table.c* pertenecientes al FileSystem.

Veamos de que forma se modifican dichos archivos:

1) Dentro de `usr/include/minix` se encuentra el archivo *config.h* , en donde se definirá la variable de código para el driver.
Ejemplo:

```
#define ENABLE_MYDRV_CODE 1@
#define ENABLE_MYDRV (ENABLE_MYDRV_CODE)@
```

Notar que el valor que tome la variable `ENABLE_MYDRV_CODE`, denotará si este dispositivo esta habilitado o no (en nuestro ejemplo vale 1, por lo tanto está habilitado).

2) En el archivo *com.h* ubicado en `usr/include/minix`, se debe definir el nombre del drive a continuacion del ultimo driver ya definido (esto es para lograr una correlación entre el numero de ID del nuevo driver y los ya existentes, ya que los IDs se definen recursivamente en base al anterior).
Ejemplo:

```
#define MYDRV (drvAnterior - ENABLE_MYDRV)
```

3) En el mismo path que los archivos anteriores se encuentra el archivo *const.h*, dentro del mismo se encuentran las definiciones del Mayor y Menor number, en la variable NR_TASK se almacena la cantidad de modulos que estan habilitados para su uso, por lo tanto se debera adicionar a la misma la nueva entrada del `MYDRV(ENABLE_MYDRV)`.
Ejemplo:

```
  NR_TASK ( <DISPOSITIVOS ANTERIORES> + ENABLE_MYDRV)
```

4) En la ruta `usr/src/kernel`, se encuentra el archivo *proto.h*, dentro del mismo se definen todas las funciones del Kernel, aqui se definira la funcion *mydrv_task*, hay que tener en cuenta que se deberá colocar esta definición en el mismo orden con respecto a las definiciones de los drivers ya existentes (tal como se hizo en el archivo com.h)
Ejemplo:

```
 #if ENABLE_MYDRV
   _PROTOTYPE (void mydrv_task, (void));
 #endif
```

5) Otra tarea será la definición del espacio dentro del stack del sistema, para que allí se pueda ubicar el manejo del driver, para lograrlo ubicaremos al archivo *table.c* que se encuentra en `usr/src/kernel`.
Ejemplo:

```
#define MYDRV_STACK (2 * SMALL_STACK * ENABLE_MYDRV)

(<DENTRO DE LA STRUCTURA TASKTAB[]>)
#ifdef (ENABLE_MYDRV_CODE)
{mydrv_task, MYDRV_STACK, "MYDRV0"},
#endif
```

Donde {mydrv_task, MYDRV_STACK, "MYDRV0"}, representa que la función mydrv_task utilizará como stack a MYDRV_STACK y se accederá al driver a través del nombre MYDRV0.

6)El último archivo a modificar es el *table.c*, que se ubica dentro de la ruta `usr/src/fs`, aquí se encuntra una tabla que mapea con los números del "system call" en las rutinas que lo realizan. El orden de las entradas de cada archivo en esta tabla determina el mapeo entre el Mayor number de los dispositivos y sus tareas. Por lo tanto se deberá ubicar la entrada de nuevo driver respetando el orden que se utilizó en el archivo *com.h.*
Ejemplo:

```
DT(ENABLE_MYDRV, dev_opcl, call_task, dev_opcl, MYDRV)
```

Los parámetros dev_opcl, call_task, dev_opcl representan, el open, el read y el close, del dispositivo.

7)Para ya tener listo el nuevo driver se deberá también definir la función que lo ejecute en algun archivo, llámese por ejemplo mydrv.c y almacenarlo en el directorio Kernel. Y por último se deberá recompilar el kernel para que se automodifique en base a los cambios antes realizados.

2.10. Modificación del Scheduler y la adm. de Memoria

a) Modifique el "scheduler.ºriginal del MINIX para el nivel de usuarios. b) Modifique la administración de memoria original del MINIX En ambos casos deberá describir en el informe cuales fueron las decisiones tomadas, cuáles fueron las expectativas y cuáles fueron los resultados obtenidos e informar el juego de

programas utilizados con los cuales se llegó a alguna conclusión. Tests o pruebas mencionados en Forma de) entrega **Rta.:**

3. Modificación de Código

3.1. Modifique el "Scheduler" original del MINIX para el nivel de usuario

3.1.1. Planteo

El planificador de tareas de MINIX utiliza un sistema de multicolas de tres niveles, donde cada una corresponde a los niveles 2, 3 y 4 vistos en el punto 1, parte a. Es en el nivel 4 donde se encuentran los procesos de usuario, cuya planificación se realiza mediante el método *round robin*. En cada *tic* del reloj se chequea que haya un proceso de usuario en ejecución. Si así fuera, se invoca al planificador para verificar si dicho proceso ha demorado en su ejecución más de 100 milisegundos, y la existencia de otro proceso de usuario en espera. Si ambas condiciones se cumplen, se desaloja al proceso actualmente en ejecución y se lo envía al final de la cola, para pasar a ejecutarse el primer proceso de dicha cola. Los procesos de nivel 2 y 3 se ejecutan hasta bloquearse por E/S o bien completar su ejecución.

Dado que se solicita modificar el *scheduler* a nivel de usuario, consideramos que la forma más transparente de hacerlo es modificando el *quantum* que se le asigna a cada proceso para el uso de procesador.

Los niveles 1 y 2 del sistema conforman el `kernel` del sistema, por lo que su código fuente estará en `/usr/src/kernel/`. Dentro de esta carpeta, el código del planificador de procesos se encuentra en `proc.c`. De este archivo nos llama la atención la función `sched()` por ser la encargada del *scheduler* del sistema. En dicha rutina se observan las líneas

```
if (--sched_ticks == 0)
[...]
sched_ticks = SCHED_RATE; /* reset quantum */
```

A partir de ellas deducimos que es `SCHED_RATE` quien maneja el quantum de planificación y que se trata de un número natural. Su definición[4] es

```
#define SCHED_RATE (MILLISEC*HZ/1000)
```

[4]que se encuentra en `clock.c` dentro del directorio antes mencionado

Donde `MILLISEC` está definida en el mismo `clock.c` con un valor de 100 y HZ se encuentra en `/usr/include/minix/const.h` y tiene un valor de 60. Realizando la cuenta arriba descrita, obtenemos que el valor de `SCHED_RATE` es de 6.

3.1.2. Expectativas

Nuestra suposición primordial es que al aumentar el valor del *quantum*, se ganará un cierto grado de eficiencia para los procesos que realicen mucho consumo del procesador, tal vez en detrimento de aquellos que se ven constantemente bloqueados por E/S. Por el contrario, reduciendo el *quantum*, se realizarían muchos más desalojos, con el consiguiente overhead que ello implica, aumentando así el tiempo total necesario para la ejecución de los procesos.

3.1.3. Testeos y resultados

Para realizar los testeos se utilizaron dos simples rutinas: `muchoProcesador.c` y `muchoProcesador.c`. La primera realiza dos ciclos anidados, cada uno de mil iteraciones que no realizan ninguna otra operación; el segundo ejecuta un ciclo de 10.000 iteraciones en el que se muestra por la salida estándar la iteración correspondiente. Para medir el cambio de performance se utilizó la función `time` del sistema, que devuelve el tiempo real de ejecución, el del proceso de usuario y el de sistema.

Para aumentar el valor del *quantum* del planificador simplemente multiplicamos la definición de `SCHED_RATE` por un valor natural. Consideramos que 1000 sería un factor de aumento aceptable.

Quisimos luego hacer el testeo inverso y dividir el valor de `SCHED_RATE` por 1000, pero dado que el valor de esta macro debía mantenerse dentro de los números naturales, para reducir su valor al mínimo sólo podíamos dividirlo por 6[5].

A la hora de realizar los testeos se decidió ejecutar varias corridas de las rutinas, midiéndolas con `time`, y obtener a partir de dichas mediciones un promedio de los tiempos de ejecución.

Con el *quantum* original se obtuvieron los siguientes resultados

Alto uso del procesador		Alto grado de E/S	
`real`	24.66	`real`	33.83
`usr`	4.04	`usr`	5.43
`sys`	0.16	`sys`	0.25

[5]Recordemos que 6 es el valor predefinido del *quantum*

Con mil veces el valor de *quantum* se obtuvieron los siguientes resultados

Alto uso del procesador		Alto grado de E/S	
real	4.50	real	34.00
usr	4.03	usr	5.45
sys	0.19	sys	0.25

Con el valor de *quantum* en 1 se obtuvieron los siguientes resultados

Alto uso del procesador		Alto grado de E/S	
real	25.16	real	34.16
usr	4.06	usr	5.47
sys	0.16	sys	0.26

3.1.4. Conclusiones

Como se puede apreciar en los resultados, nuestras suposiciones no fueron del todo acertadas.

Si bien el aumento exagerado del valor de *quantum* redujo drásticamente el tiempo de ejecución total del programa que sólo itera en ciclos anidados, no hizo mella significativa en aquel que realiza operaciones de E/S constantemente. Esto último puede deberse a que, como explicáramos en el planteo del problema, los procesos de nivel 2 y 3 no son desalojados, y las operaciones de E/S se encuentran entre ellos.

Por otro lado, la disminución del valor de *quantum* no parece haber afectado significativamente el desempeño de ninguno de los dos programas.

En el caso del que realiza E/S, debe basarse en el mismo principio explicado unos párrafos antes.

En cuanto al programa que realiza únicamente ciclos, suponemos que el problema radica en que el valor de *quantum* original es realmente bajo. Al aumentarlo pudimos hacerlo por un factor de 1000, pero sólo fue posible reducirlo en $\frac{1}{6}$ para alcanzar el mínimo valor posible.

3.2. Modifique la administración de memoria original del MINIX

3.2.1. Planteo

MINIX utiliza una administración segmentada, dividiendo el proceso en secciones de texto, datos y pila[6], así como el lugar necesario para el crecimiento de esta última. Sin embargo, MINIX 2.0 no reconoce memoria virtual, por lo que el

[6] Análogos a las secciones `.text`, `.data` y `.bss` de la codificación en Assembly

programa debe entrar entero en memoria. Para la asignación de ésta se utiliza el criterio *first fit*. Esto es, recorre la tabla de huecos de memoria en busca uno lo suficientemente grande como para que la memoria que solicita un proceso quepa; en cuanto encuentra uno, asigna la memporia necesaria y corre el inicio del hueco. Si el hueco se completó, se elimina la entrada de la tabla.

En este caso realizaremos una modificación en la asignación de memoria, cambiándola de *first fit* a *best fit* (hueco óptimo). Esto es, recorrerá la tabla de huecos entera y se asignará el hueco que provoque el menor "desperdicio".

Cabe destacarse que en MINIX la memoria no se cuenta en bytes, sino en *clicks*. Estos son clusters de 256 bytes cada uno.

3.2.2. Expectativas

Con esta modificación se espera conseguir una menor fragmentación de memoria, necesaria en este esquema pues MINIX no reconoce memoria virtual.

3.2.3. Testeos y Resultados

Para realizar los testeos necesitamos, en principio, poder conocer los huecos de memoria disponible. Para ello implementamos una rutina en el archivo donde se encuentra definida la lista encadenada de huecos para que la muestre en pantalla. El problema surgió al querer llamar a esta función desde un entorno de usuario. Conocemos dos posibilidades para ello. Una de ellas es implementar un System Call. Sin embargo, por una cuestión de comodidad, preferimos sacrificar eficiencia por eficacia e introdujimos dos llamadas a esta función a la hora de asignar la memoria (una antes y otra después de hacerlo).

Para facilitar la comprensión de los ejemplos, decidimos realizar una pequeña modificación más: en lugar de tomar la primera parte del mejor hueco disponible, tomamos la última. De esta manera, se evita el tener que hacer cuentas para ver a qué hueco fue asignada la memoria pues el inicio del mismo quedará inmutado (salvo que el hueco desaparezca).

Nos encontramos con un estado inicial de tres huecos: el primero de 1291 clicks; el segundo de 108; y el tercero de 56572 clicks.

La idea de la prueba a realizar es la siguiente:

- Realizar una reserva de memoria A.
- Realizar una segunda reserva de memoria B del mismo tamaño de A.
- Liberar el bloque A.
- Realizar una reserva de memoria C del tamaño de B

- Verificar que C toma el bloque liberado por A, pues es la mejor elección.

Para ello decidimos tomar un tamaño lo suficientemente grande para que quepa en el hueco mayor, de manera que, al duplicarlo, haya lugar suficiente aún. Resolvimos entonces tomar 200,000 bytes[7], que equivalen a 2077 clicks[8]. Realizamos una primera llamada al procedimiento que reserva memoria y obtuvimos el siguiente resultado:

Hueco	Comienzo	Clicks
0	408	1291
1	2033	108
2	8963	54495

Realizamos entonces la segunda reserva de memoria, de exactamente el mismo tamaño y el resultado fue el siguiente:

Hueco	Comienzo	Clicks
0	408	1291
1	2033	108
2	8963	52418

Como se puede apreciar, en ambos casos se tomó la última porción del hueco mayor.

Procedemos ahora a liberar la memoria reservada por el primer proceso, quedando como resultado:

Hueco	Comienzo	Clicks
0	408	1291
1	2033	108
2	8963	52418
3	63458	2077

Notar la aparición de un nuevo hueco. Esta es, justamente, la memoria recién liberada. Si hacemos las cuentas, nos encontramos con que $8963 + 52418 = 61381$. A partir de ese lugar se encuentra reservado el segundo bloque de memoria. Esto se puede verificar fácilmente al sumarle el tamaño del mismo: $61381 + 2077 = 63458$.

Finalmente, reservamos nuevamente 2077 clicks de memoria, quedando entonces la tabla de huecos como sigue:

[7] Un arreglo de 100,000 posiciones de enteros, en esa época representados con 2 bytes

[8] Recordar que los clicks –al igual que cualquier cluster que se precie– son indivisibles

Hueco	Comienzo	Clicks
0	408	1291
1	2033	108
2	8963	52418
3	6358	0

Como se puede apreciar, en lugar de elegir el primer hueco donde cupiera la memoria a reservar –2–, el Administrador de Memoria eligió aquel que producía el menor desperdicio de memoria –en este caso, llenando el hueco 3 entero–.

A modo de consideración, los datos están tomados antes de que el Administrador de Memoria ejecute la eliminación del hueco inexistente de la tabla.

3.2.4. Conclusiones

Los resultados parecen indicar que nuestra suposición ha sido acertada: elegir el mejor hueco en lugar del primero disponible genera un grado considerablemente menor de fragmentación en la memoria.

11. 3.3. Imlpementación de un System Call

Implementar un nuevo *system call* llamado *newcall* que devuelva el *pid* del proceso que lo invocó y lo muestre por pantalla. (Ver sugerencias al fondo de los enunciados) **Rta.:**
Para implementar esta nueva llamada al sistema `(syscall)` nos basaremos en la explicacion dada en clase, la llamaremos newcall.

- Comenzaremos por crear el archivo `newcall.s` en la siguiente ubicacion `/usr/src/lib/syscall`, con el siguiente codigo de Assembler.

```
.sect .text
.extern __newcall
.define _newcall
.align 2
_newcall:

jmp __newcall
```

Este archivo nos permitira realizar el macheo entre la llamada al newcall y la `__newcall` que es la que se obtiene al compilar código bajo la norma POSIX, que sera la que se creara como lubreria para que los usuarios puedan

utilizarla sin conocer la esructura de la misma.

- Luego, modificamos el archivo Makefile en la misma carpeta para que se incluya posteriormente la compilaci´on del archivo newcall.s. Primero fue necesario inclu´ir la l´inea

  ```
  $(LIBRARY)(newcall.o) \
  ```

  Junto con los demas objetos a compilar por el Makefile. Y luego:

  ```
  $(LIBRARY)(newcall.o): newcall.s
  $(CC1) newcall.s
  ```

- Creamos tambien el archivo `/usr/src/lib/posix/_newcall.c` con el siguiente c´odigo:

  ```
  #include <lib.h>
  #define newcall _newcall
  #include <unistd.h>

  PUBLIC pid_t newcall()
  {
  message m;
  return(_syscall(MM, NEWCALL, &m));
  }
  ```

- De la misma forma que en el directorio anterior, modificamos el archivo Makefile en dicha carpeta para que se incluya la compilaci´on. Inclu´imos las siguientes l´ineas:

  ```
  $(LIBRARY)(_newcall.o) \ . . .
  $(LIBRARY)(_newcall.o): _newcall.c
  $(CC1) _newcall.c
  ```

- Incorporamos un nuevo prototipo al archivo `/usr/include/unistd.h` agregando la l´inea:

  ```
  _PROTOTYPE( int newcall, (void));
  ```

  Tener en cuenta que esta declaración se coloca dentro de `ifndef UNISTD_H`

- En el archivo `include/callnr.h` le asignamos el numero a nuestro system call mediante la l´inea:

  ```
  #define NEWCALL 77
  ```

Adem´as, incrementamos la constante `NCALLS` en uno para que la numeraci´on de system calls quede consistente.

- Modificamos el archivo `/usr/src/mm/table.c` agregando la l´inea

```
do_newcall, /* 77 = newcall */
```

Y tambien en el archivo `/usr/src/fs/table.c` colocamos en la misma entrada "77"la representación de que a esa entrada no le corresponde ninguna llamada del FS.

```
no_sys, /* 77 = newcall */
```

- También tendremos que insertar el PROTOTYPE en la tabla de funciones de atenci´on de System Calls del MM/FS ubicada en `/usr/src/mm/proto.h`

```
  /* newcall.c */
_PROTOTYPE( int do_newcall, (void) );
```

- Creamos el archivo `newcall.c` con el código de la nueva System Call: lo guardamos en el directorio `/usr/src/mm`.

```
  #include "mm.h"
  #include<minix/callnr.h>
  #include <signal.h>
  #inclue "mproc.h"

  PUBLIC int do_newcall()
  {
register struct mproc *rmp= mp;
register int r;
r = mproc[who].mp_pid;
return r;
  }
```

- Por ultimo, editamos el archivo `/usr/src/mm/Makefile` agrgarndo a la lista de objetos el objecto newcall.o para que lo incluya en la compilación:

```
OBJS = Xi.o Xi+1.o .... newcall.o
```

Donde Xi.o es el iesimo objeto del sistema.

- Ahora luego de haber modificado todos los archivos del sistema y habiendo agregado nuestra objeto a la libreria, deberemos recompilar el Kernel, para este fin usamos el comando `make hdboot` desde `/usr/src/tools`. Una vez hecho esto booteamos con la nueva imagen, creamos el archivo ej11.c en `/usr` con el siguiente código:

```
  #inclide "./src/lib/posix/_newcall.c"
  #include <lib.h>
  #include <unistd.h>
  #include <stdlib.h>

  int main()
  {
int r;
r = newcall();
printf("%d",r);
return 0;
  }
```

  Compilamos el archivo: `cc ej11.c -o newcall` Luego de esto solo resta llamar a la funcion con `./newcall` para que nos muestre por pantalla el pid del proceso actual.

12. 3.4. Primitivas para manejo de semáforos y las primitivas P y V

Implementar las primitivas para manejo de semáforos y las primitivas P y V (deben ser implementadas mediante system-call) (ver sugerencias al fondo de los enunciados) Pruébelo con el siguiente caso:
Modelo productor/consumidor. No olvide los test de prueba. (Tests o pruebas mencionados en Forma de entrega).

Rta.:

El manejo de semáforos lo hemos implementado mediante como system calls en el modulo MM. El bloqueo de los procesos solicitantes se logra mediante dos task calls implementadas en el microkernel.

3.4.1. Interfaz de Usuario

Las siguientes funciones son las que se utilizarán a nivel de usuario,

- `int sem_creat(int id, int val)`

  Crea el semáforo agregando al proceso invocante como usuario del semáforo con el nombre dado. En caso de que no exista un semáforo de tal nombre, el mismo se inicializa con el valor `val`.

  Valores Devueltos

  - `SEM_TOO_MANY_PROCS:` Demasiados procesos asignados a este semáforo.
  - `SEM_NO_AVAILABLE:` Se llegó al máximo posible de semáforos a definir.

- `int semp(int nombre)`

  Implementa el operador P(x) para el nombre de semaforo dado. Si el semáforo esta siendo utilizado (valor < 0), el proceso invocante es encolado y bloqueado. El valor es decrementado en 1.

  Valores Devueltos

  - `SEM_DADO:` semáforo concedido al proceso.
  - `SEM_INVALID:` semáforo inexistente o no se llamo a `sem_creat()`
  - `SEM_COLA_LLENA:` Demasiados procesos esperando el semáforo.

- `int semv(int id)`

  Implementa el operador V(x) para el semáforo con el nombre dado. El valor del semáforo es incrementado en 1.

  Valores Devueltos

  - `SEM_FREED:` Semáforo liberado exitosamente.
  - `SEM_INVALID:` semáforo inexistente o no se llamo a `sem_creat()`

- `int sem_get_val(int id, int* val)`

  Devuelve el valor asociado a un semáforo (contador de Disjktra).

  Valores Devueltos

  - `SEM_INVALID:` semáforo inexistente o no se llamo a `sem_creat()`
  - `SEM_OK:` Valor del semáforo obtenido exitosamente.

- `int sem_num_procs(int id)`

  Devuelve el número de procesos encolados en el semáforo con el nombre dado.

Valores Devueltos

- valor ¿0, Indicando la cantidad de procesos encolados.
- `SEM_INVALID` :semáforo inexistente.

3.4.2. Implementación

Lo primero que hemos hecho es para cada de las funciones de usuario, crearles la interfaz en el módulo de Memoria dle Minix. Esto lo hemos realizado siguiendo los pasos mencionados para el punto 11 de emphSystem Calls.

hemos creado emphSystem Calls para cada una de las funciones,

```
sem_creat
semp
semr
semv
sem_get_val
sem_set_val
sem_num_procs
```

En */usr/src/lib/posix/_sem.c* hemos definido las funciones que llamará el usuario

```
#include <lib.h>
#define semp _semp
#define semv _semv
#define semr _semr
#define sem_num_procs _sem_num_procs
#define sem_creat     _sem_creat
#define sem_get_val _sem_get_val
#define sem_set_val _sem_set_val

#include <unistd.h>

PUBLIC int semp(int sem)
{
  message m;

  m.m1_i1 = sem;
  return(_syscall(MM, SEMP, &m));
}
```

```
PUBLIC int semv(int sem)
{
  message m;

  m.m1_i1 = sem;
  return(_syscall(MM, SEMV, &m));
}


PUBLIC int semr(int sem)
{
  message m;

  m.m1_i1 = sem;
  return(_syscall(MM, SEMR, &m));
}

PUBLIC int sem_num_procs(int sem)
{
  message m;
  m.m1_i1 = sem;
  return(_syscall(MM, SEM_NUM_PROCS, &m));
}


PUBLIC int sem_creat(int name, int val)
{
  /*Devuelvo el id del semáforo si la llamada fue exitosa o de lo contrario -1
  indicando una condición de error.*/
  message m;
  int ret;

  m.m1_i1 = name;
  m.m1_i2 = val;

  ret =_syscall(MM, SEM_CREAT, &m);

  if (ret != SEM_ASSIGNED)
return(-1);
```

```
  else
return((int)m.m2_i1);
}


PUBLIC int sem_get_val(int sem, int* val)
{
  /*Devuelvo el valor si la llamada fue exitosa o de lo contrario -1
  indicando una condición de error.*/
  message m;
  int ret;

  m.m1_i1 = sem;

  ret =_syscall(MM, SEM_GET_VAL, &m);
  *val =  m.m2_i2;

  return(ret);
 }

PUBLIC int sem_set_val(int sem, int val)
{

  message m;
  int ret;

  m.m1_i1 = sem;
  m.m1_i2 = val;


  ret =_syscall(MM, SEM_SET_VAL, &m);

  return(ret);

}
```

3.4.3. Modificaciones para agregar las System Calls

Agregamos los prototipos de las nuevas funciones en */usr/src/mm/proto.h*

```
/*****************************/usr/src/mm/proto.h************************/
.
.
.
#ifdef UBA_FCEN
/*sem.c*/
_PROTOTYPE( int do_sem ,(void ) );
_PROTOTYPE( int do_semv ,(void ) );
_PROTOTYPE( int do_semp ,(void ) );
_PROTOTYPE( int do_semr ,(void ) );
_PROTOTYPE( int do_sem_creat ,(void ) );
_PROTOTYPE( int do_sem_num_procs ,(void ) );
_PROTOTYPE( int do_sem_get_val ,(void ) );
_PROTOTYPE( int do_sem_set_val ,(void ) );
#endif
.
.
.
/*****************************/usr/src/mm/proto.h************************/
```

Luego, las nuevas entradas en el vector de llamadas del MM (y su contrapartida en el fs).

```
/**************************/usr/src/mm/table.c*****************************/
.
.
.
#ifdef UBA_FCEN
do_semp ,/* 78 = semp */
do_semv ,/* 79 = semv */
do_semr  ,/* 80 = semr */
do_sem_num_procs  ,/* 81 = sem_num_procs*/
do_sem_creat  ,/* 82 = sem_creat*/
do_sem_get_val  ,/* 83 = sem_getval*/
do_sem_set_val  ,/* 84 = sem_setval*/
#endif
.
.
.
/**************************/usr/src/mm/table.c*****************************/
```

3.4.4. Estructuras de datos en la implementación en el MM

La estructura elegida consta de dos matrices y dos vectores, todos de longitud fija. Una matriz representará los procesos encolados para cada semáforo (0....n-1) y la otra que procesos tienen un semáforo asociado. Cada fila de la matriz se modelará como una cola y guardará un identificador de proceso (número de proceso recibido en la system call).

Otros dos vectores indicarán los nombres de los semáforos definidos y los valores de los contadores asociados a dichos semáforos.

Las declaraciones de las estructuras se realizaron en `glo.h`.

```
/*Cola de semáforos*/
EXTERN int sem_cola [MAX_SEMS][MAX_PROCS + SEM_PROC_BEGIN]
/*Procesos con semáforo asignado*/
EXTERN int sem_usa [MAX_SEMS][MAX_PROCS + SEM_PROC_BEGIN]
/*Nombres de los semáforos*/
EXTERN int sem_nom[MAX_SEMS]
/*Valores de los semáforos*/
EXTERN int sem_vals[MAX_SEMS]
```

Para recorrer la matriz y simular una cola existen dos contadores por fila llamados *head* y *tail*. El primero guarda la posición con el número del proceso que actualmente tiene al semáforo. El segundo es la posición a asignarle al próximo proceso a ser encolado. Eventualmente los valores de head y tail pueden exceder la última posición de la fila, pasando nuevamente a la primera, cerrándose un anillo. La cola estará llena cuando tail alcance el valor de head.

3.4.5. Código de la Implementación en MM

Se agregaron para la implementación los fuentes en `/usr/src/mm/` que contienen las definiciones de cada una de las funciones que mantienen la estructura.

`sem_creat.c, semp.c, semr.c, semv.c, sem_get_val.c, sem_set_val.c, sem_num_procs`

3.4.6. System Calls

Sos siguientes son los códigos de las funciones más importantes

sem creat

- Busca en `sem_nom` si existe un semáforo con el nombre pasado como parámetro.

- Si existe busca en la correspondiente fila de `sem_usa` una entrada libre y le asigna el pid.
- Si no existe, busca una fila libre en `sem_usa` y crea un semáforo nuevo con el pid asignado, además de setear el valor inicial en `sem_vals[]`.

El siguiente es el código de la función:

```
int do_sem_creat(int *sem)
{

  int i,j;
  int req_sem;
  int newval=mm_in.m1_i2;


  #ifdef SEM_DEBUG
  printf ("sem: Proceso %u quiere utilizar semaforo %u\n",
     mproc[who].mp_pid, mm_in.m1_i1);
  #endif

  req_sem = sem_get_id(mm_in.m1_i1); /*Obtengo el id para el nombre pasado*/

  /*Busco el índice si el semáforo ya existía o el ultimo disponible*/

if (req_sem != SEM_INVALID)
{
i = req_sem;
*sem = i;

/*Buscar entrada libre en sem_usa*/
for (j = 0; j < MAX_PROCS;j++){
if (sem_usa[i][j] == -1 || sem_usa[i][j]== mm_in.m_source){
sem_usa[i][j] =  mm_in.m_source;
#ifdef SEM_DEBUG
printf ("sem: Asignado el semáforo con id %u.\n", i);
#endif
return (SEM_ASSIGNED);
}
}
```

```
#ifdef SEM_DEBUG
printf ("sem: Demasiados procesos para ese nombre.\n");
#endif
return(SEM_MANY_PROCS);

}

  /*El nombre no existe, lo creo*/
  for (i = 0; i < MAX_SEMS; i++)
if (sem_nom[i] == SEM_FREE_SLOT)
{
*sem=i;
sem_nom[i]=mm_in.m1_i1;
/*Valor inicial del semáforo*/
sem_vals[i]=newval;
sem_usa[i][0] =  mm_in.m_source;
  #ifdef SEM_DEBUG
   printf ("sem: Asignado el nuevo semáforo con id %u.\n", i);
     #endif
return(SEM_ASSIGNED);
        }

#ifdef SEM_DEBUG
   printf ("sem: No hay mas slots libres para asignar semáforos!!!\n", i);
        #endif
return(SEM_NO_DISP);
}
```

semp

- Verifica que el nombre de semáforo pasado exista y que el proceso tenga una entrada en `sem_usa` para el mismo.
- Intenta encolar el número de proceso en la fila correspondiente de `sem_cola`. Puede darse que `sem_cola` este llena (condición error). En caso contrario, se verifica que el valor del semáforo sea mayor o igual a 0 para saber si debe pedir al kernel que proceso invocante sea bloqueado.

```
PRIVATE int do_semp()
{
  int ret;
```

```
  int req_sem;
  int head;
  int tail;
  int j;
  message m;

  /*pid = mproc[who].mp_pid;*/
  req_sem = sem_get_id(mm_in.m1_i1);  /*Obtengo el numero de semáforo requerido*/

  if (req_sem == SEM_INVALID)  /*semáforo invalido?*/
         return(SEM_INVALID);

  #ifdef SEM_DEBUG
  printf ("sem: Proceso %u requiere P(x) sobre semaforo %u\n", mproc[who].mp_pid, req_
  #endif


  for (j = 0; j < MAX_PROCS; j++)
  {
if (sem_usa[req_sem][j] == mm_in.m_source)
  break;
  }

  if (j == MAX_PROCS){
        #ifdef SEM_DBG
   printf ("sem: Proceso %u quiere actuar sobre semaforo %u,
    pero no llamo a sem_creat.\n", mproc[who].mp_pid, req_sem);
   #endif

return(SEM_INVALID);
  }

  head = sem_cola[req_sem][SEM_HEAD_POS];
  tail = sem_cola[req_sem][SEM_TAIL_POS];

  /*condición de cola vacía*/
  if (sem_vals[req_sem] >= 0){
   head = 0; /*Primer posición en array*/
tail = 1;               /*Siguiente posición array*/
ret = SEM_DADO;
```

```
        sem_cola[req_sem][SEM_HEAD_POS] = head;
        sem_cola[req_sem][SEM_TAIL_POS] = tail;
sem_cola[req_sem][head + SEM_PROC_BEGIN] =  mm_in.m_source;
sem_vals[req_sem]--;
#ifdef SEM_DBG
printf ("sem: Cola vacía. Asignando head %u. Tail en %u\n", head,tail);
#endif
}
   else if (tail == head) /*Cola llena*/
{
ret = SEM_COLA_LLENA;

#ifdef SEM_DBG
printf ("Cola llena!!\n");
#endif
}
else /*Lugar libre en la cola.  Encolar.*/
{
sem_cola[req_sem][tail+SEM_PROC_BEGIN] =  mm_in.m_source;
tail = (tail + 1) % SEM_COLA_LARGO;
sem_cola[req_sem][SEM_TAIL_POS] = tail;
m.m1_i1 =  mm_in.m_source;
sem_vals[req_sem] --;

_taskcall(SYSTASK, SYS_BLOCK, &m);
ret = SEM_DADO;

#ifdef SEM_DEBUG
printf ("Proceso encolado. Tail en %u\n", tail);
#endif
}

   return(ret);

}
```

semv

- Verifica que el nombre de semáforo pasado exista y que el proceso tenga una entrada en `sem_usa` para ese id.
- Elimina al proceso de la cola, incrementando en uno el puntero head.
- Si en la nueva posición hay un proceso (head distinto de tail) envía un mensaje al mismo desbloqueándolo. Si tal proceso es inválido, continúa con el siguiente (ver "Task calls en microkernel"mas abajo).

```
PRIVATE int do_semv()
{
  int ret;
  int req_sem;
  int head;
  int tail;
  int next_pid;
  int j;
  int err_cond = TRUE;
  message m;

  req_sem = sem_get_id(mm_in.m1_i1);  /*Obtengo el numero de semáforo requerido*/


  if (req_sem == SEM_INVALID)  /*semáforo invalido?*/
         return(SEM_INVALID);


  #ifdef SEM_DBG
  printf ("sem: Proceso %u quiere liberar semaforo %u\n", mproc[who].mp_pid, req_sem);
  #endif


  for (j = 0; j < MAX_PROCS; j++)
  {
if (sem_usa[req_sem][j] ==  mm_in.m_source)
  break;
  }

  if (j == MAX_PROCS){
        #ifdef SEM_DEBUG
   printf ("sem: Proceso %u quiere actuar sobre semaforo %u,
      pero no llamo a sem_creat.\n", mproc[who].mp_pid, req_sem);
```

```
   #endif

return(SEM_INVALID);
  }
  while (err_cond){
head = sem_cola[req_sem][SEM_HEAD_POS];
   tail = sem_cola[req_sem][SEM_TAIL_POS];


   /*Desencolar proceso*/
    head = (head + 1) % SEM_COLA_LARGO;
    if (head == tail) /*No hay mas procesos pendientes. No mando mensaje*/
    {
head = SEM_COLA_VACIA;
tail = 0;
next_pid = 0;
}else
next_pid = sem_cola[req_sem][head + SEM_PROC_BEGIN];

  sem_cola[req_sem][SEM_HEAD_POS] = head;
sem_cola[req_sem][SEM_TAIL_POS] = tail;
sem_vals[req_sem]++;

/*Hay alguien para avisarle que tiene el sem?
   Si es así, envio a ese proceso un mensaje para que pueda
   comenzar a usar el semáforo.  Esto lo hago antes de despertar
   al proceso que libero el semáforo.*/


   #ifdef SEM_DEBUG
      printf ("sem: Próximo proceso %u\n", next_pid);
      #endif

if (next_pid > 0){
m.m1_i1 = next_pid;

if (_taskcall(SYSTASK, SYS_UNBLOCK, &m) == -1)
printf ("sem: Próximo proceso en cola muerto!!!.
Continuando con el siguiente.\n");
/*reply(sem_get_who(next_pid), SEM_GRANTED, 0,0);*/
```

```
else

err_cond = FALSE;
}else err_cond=FALSE;
ret = SEM_FREED;
   }
   return(ret);

}
```

3.4.7. Task calls en microkernel

La ventaja de que la implementación de bloqueo de semáforos sea llevada a cabo dentro del microkernel, soluciona dos posibles situaciones que no se podrían controlar de otra manera:

- Si un numero de proceso se encuentra encolado para un semáforo, pero el proceso asociado no existe mas (ej. se le hizo un kill), al tocarle el turno de usar el semáforo se estaríamos enviando una señal a un proceso inexistente o que no esta esperando realmente al semáforo. Ambos casos traen aparejado un deadlock. En el primer caso, el resultado del *reply* podría ser además impredecible.
- Algo similar ocurre al reiniciar un semáforo. Al querer liberar todos los procesos que lo están esperando, no tenemos manera de determinar su estado, pudendo enviar señales incorrectas.

Por estas dos razones, es conveniente implementar el bloqueo y desbloqueo a nivel microkernel. El desbloqueo verifica primero si el proceso en cuestión existe y si realmente espera a un semáforo, mediante la verificación de un nuevo flag agregado para tal fin. Si falla cualquiera de las dos condiciones, se devuelve un código de error, permitiendo al MM actuar en consecuencia. Para el primer problema, como decisión de diseño, aplicamos la siguiente política: Si el próximo proceso a desbloquear es inválido, desbloquear el siguiente.

Esto tiene la ventaja de que se puede seguir utilizando el semáforo, aunque podría tener consecuencias impredecibles a nivel usuario. Otra política podría haber sido avisar al usuario mediante un panic del problema y dejar a todos los procesos bloqueados para siempre.

Para el segundo problema no hay que hacer nada. La llamada desbloqueante simplemente ignorará a los procesos inválidos.

```
/********************************* /usr/src/kernel/system.c ***************************
```

```
.
.
    case SYS_BLOCK:   r = do_block(&m) ; break;
    case SYS_UNBLOCK: r = do_unblock(&m);break;
    default:    r = E_BAD_FCN;
.
.
/********************************* /usr/src/kernel/system.c ************************
```

En el mismo archivo agregamos los prototipos y el código de las llamadas:

```
/********************************* /usr/src/kernel/system.c ************************
.
.
FORWARD _PROTOTYPE( int do_block, (message *m_ptr));
FORWARD _PROTOTYPE( int do_unblock, (message *m_ptr));
.
.


PUBLIC int  do_sem_block(m_ptr)
register message *m_ptr;
{
register struct proc *rp;
printf("SEMBLCKREQ");
rp=proc_addr(m_ptr->PROC1);
rp->p_flags |= B_X_SEM;
lock_unready(rp);
return(OK);
}

PUBLIC int do_sem_unblock(m_ptr)
register message *m_ptr;
{
int old_flags;
register struct proc *rp;
if (!isokprocn(m_ptr->PROC1)){
/*Proceso encolado muerto*/
return(-1);
}
rp=proc_addr(m_ptr->PROC1);
```

```
old_flags = rp->p_flags;
rp->p_flags &= ~B_X_SEM;
if (old_flags !=0 && rp->p_flags ==0){
lock_ready(rp);
return(OK);
}
else
/*El proceso no estaba bloqueado. No dar falsa alarma*/
return(-1);
}
```

```
/********************************* /usr/src/kernel/system.c ************************
```

Como se puede apreciar, en el caso en que se desbloquea un proceso, primero verificamos la validez del número mediante la función `isokprocn()` ya provista. Luego verificamos el flag `B_X_SEM` para ver si realmente estaba bloqueado. Estos dos tests nos salvan de dar el control a procesos inválidos. El flag `B_X_SEM` es nuevo y esta definido en */usr/src/minix/proc.h* como sigue:

```
/********************************* /usr/src/kernel/proc.h ***************************
.
.

#define P_STOP 0100 /*set when process is being traced*/
#ifdef UBA_FCEN
#define B_X_SEM 0200 /*bloqueado por semáforo*/
#endif
.
.
/********************************* /usr/src/kernel/proc.h ***************************
```

El valor del flag fue elegido igual que todos los demás, un número con un solo bit en 1 en una posición única, de manera que se pueda activar haciendo un OR y borrar haciendo un AND invertido.

3.5. Pruebas

A continuación presentamos una prueba llevada a cabo para nuestra implementación de semáforos. Todos los archivos relacionados se encuentran en */usr/tests/*.

3.5.1. Prueba de exclusión mutua del recurso pantalla

Aquí lanzamos dos procesos, A y B. Cada uno debe imprimir 10 líneas iguales, ÁAAAA'y 'BBBBB'. La impresión de cada línea esta separada por un `sleep` de un segundo. Las líneas de ambos procesos no deben mezclarse. Para garantizar esto, utilizamos un semáforo de exclusión. El proceso que esta imprimiendo actualmente tiene el semáforo. El otro espera a que termine sus 10 líneas. El código es el siguiente:

```
/******************************** /usr/tests/semtest.c ****************************
/*********************************semtest*********************************.
Este simple ejemplo sincroniza el uso del recurso pantalla entre dos
procesos que tienen que imprimir 10 líneas en forma ininterrumpida.
Para desactiva el semáforo, simplemente borrar la definición USE_SEM
*****************************************************************************/
#include <stdio.h>

int main(void)
{
int npid;
int i;

/*Reiniciar el semáforo*/
semr(5555);
npid = fork();

if (npid > 0){
#ifdef USE_SEM
sem_creat(5555, 0);
#endif
while(1)
{

#ifdef USE_SEM
semp(5555);
#endif
for (i = 1; i < 10; i++)
{
printf("AAAAAA\n");
sleep(1);
}
```

```
#ifdef USE_SEM
semv(5555);
#endif
}
}
else
{
#ifdef USE_SEM
sem_creat(5555, 0);
#endif
while(1)
{

#ifdef USE_SEM
semp(5555);
#endif
for (i = 1; i < 10; i++)
{
printf("BBBBB\n");
sleep(1);
}
#ifdef USE_SEM
semv(5555);
#endif
}
}

}

/********************************** /usr/tests/semtest.c ******************************
```

Ejecutando el programa SIN semáforos:

```
# ./sem_test
AAAAA
BBBB
BBBB
AAAAA
BBBB
AAAAA
```

```
BBBB
AAAAA
BBBB
AAAAA
BBBB
```

Y por como vemos, los procesos se solapan en una forma aproximadamente equitativa, que es lo esperado.
Ejecutando el programa CON semáforos se obtiene el resultado deseado:

```
#./ex1sem
sem: Proceso 98 reiniciando semaforo 0
sem: Proceso 98 quiere utilizar semaforo 5555
sem: Asignado el nuevo semaforo con id 0.
sem: Proceso 98 requiere P(x) sobre semaforo 0
sem: Cola vacia. Asignando head 0. Tail en 1
AAAAAA
sem: Proceso 99 quiere utilizar semaforo 5555
sem: Asignado el semaforo con id 0.
sem: Proceso 99 requiere P(x) sobre semaforo 0
SEMBLCRREQProceso encolado. Tail en 2
AAAAAA
AAAAAA
AAAAAA
AAAAAA
sem: Proceso 98 quiere liberar semaforo 0
sem: Proximo proceso 7
sem: Proceso 98 requiere P(x) sobre semaforo 0
EMBLCRREQProceso encolado. Tail en 3
BBBBBB
BBBBBB
BBBBBB
BBBBBB
```