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

Sistemas Operativos

Primer Cuatrimestre de 2005

Trabajo Práctico Minix

Fecha de Entrega: 5/7/05

Grupo 2	LU	Correo electrónico
Jonathan Chiocchio	849/02	jonycobain@yahoo.com.ar
Gabriel Honoré	503/03	ghonore@tutopia.com
Gabriel Tursi	699/02	gabrieltursi@hotmail.com

RESUMEN

En el presente trabajo ejercitamos la instalación del sistemas operativo Minix 2.0, así también cómo la utilización de los comandos básicos. También aplicamos en forma práctica algunos de los conocimientos aprendidos en la materia tales como: modificaciones en el *scheduler*, en la administración de memoria, manejo de semáforos, *System Calls*, Productor - Consumidor, entre otros.

PALABRAS CLAVE

Minix - Comandos - Semaforos - Kernel

Índice

Introducción

Enunciado

Sistemas Operativos - Primer Cuatrimestre 2005
Trabajo Práctico Minix

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.

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.

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

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.

Principal

Ejercicios

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

Aclaración Previa

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>`.

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, a diferencia de otros sistemas operativos, cuyo kernel es un programa monolítico, está dividido en módulos, y se compone por una colección de procesos que se comunican entre sí y con los procesos de usuario, empleando como premisa la transferencia de mensajes para la comunicación entre procesos. Esto lo hace más sencillo, mantenible y escalable.

Minix está estructurado en cuatro capas, realizando cada una una función bien definida. A continuación se muestra un gráfico que ilustra éstas capas:

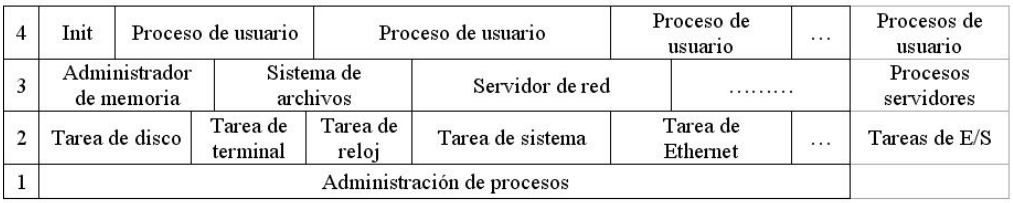


Figura 1: Layout de Minix

La capa de administración de procesos atrapa todas las interrupciones, se encarga de la planificación de procesos y ofrece a las capas superiores un conjunto de procesos independientes con los cuales se pueden comunicar a través de mensajes.

La capa que le sigue 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 contiene procesos que son de utilidad a todos los procesos de usuarios. Por el administrador de memoria pasan todas las llamadas al sistema en las cuales se requiere 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.

En la capa superior, la cuarta, encontramos todos los procesos de usuario, como ser *shells*, editores, compiladores y programas escritos por el usuario.

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.:
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

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

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

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.

El comando `mknod`

Puesto que Minix trata a los dispositivos como entradas en el sistema de archivos, se necesita una herramienta para crear un archivo nuevo que no tenga i-nodos apuntando a un bloque del disco rígido, sino que debe apuntar a un dispositivo de hardware. Para este fin sirve el comando `mknod`. En el caso que el usuario desee agregar un dispositivo nuevo, tendrá que utilizar éste comando para conseguirlo.

Ahondando en tecnicismos, *mknod* crea un FIFO (tubería con nombre), un fichero especial de bloques, o un fichero especial de caracteres, con el nombre especificado.

Así, un fichero especial 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 ficheros especiales se refieren a dispositivos físicos (disco, cinta, terminal, impresora) o a servicios del sistema operativo (`/dev/null`, `/dev/random`).

Cuando se construye un fichero especial de bloques o caracteres, hay que dar tras el tipo del fichero los números de dispositivo mayor y menor. Por omisión, los permisos de los ficheros creados son 0666 menos los bits puestos a 1 en la *umask*.

3. Comandos Básicos de MINIX/UNIX

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:

1. Loguearse como *root*
2. Invocar `passwd` sin argumentos
3. Escribir la clave: "root"
4. Confirmar la clave escrita: "root"

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

Comentarios

Nos encontarmos con un problema al intentar loguearnos después de haber cambiado el password, ya que el *timeout* para loguearse (el cual se supone que dura 60 segundos) no funciona correctamente, pues expira en menos de 3 segundos.

Luego de investigar sobre el tema, encontramos que es posible configurar en *Bochs* el PIT (programmable interval timer), opción que nos permite especificarle al *Bochs* que mantenga el PIT sincronizado con el tiempo real, para prevenir que el emulador funcione demasiado rápido.

Por lo tanto, activamos la opción de tiempo real, agregando la siguiente línea en el archivo de configuración `boschsrc.bxrc`:

```
pit: realtime=1
```

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 *grupo*. Creamos el directorio con la siguiente invocación:

```
mkdir /usr/grupo
```

3.5. cp

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

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

```
cp /etc/passwd /usr/grupo
```

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/grupo/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/grupo/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/grupo/passwd` se debe invocar:

  ```
  chmod u+r+w+x /usr/grupo/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/grupo/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/grupo/passwd` se debe invocar:

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

  Aclaración:

  No se debe dejar espacios entre los operadores de cambio de permiso.
  También se puede ejecutar la misma instrucción simplificando los parámetros, ya que los operadores no pasados explícitamente, son tomados como negativos, quedando de la siguiente manera:

  ```
  chmod u=rwx,g=rx,o=x /usr/grupo/passwd
  ```

  Otra opción es tomar cada terna pensada en bits y pasar el valor que la selección representa, por ejemplo, si los tres tipos de permisos están habilitados, sería como tener los 3 bits encendidos (111), por lo que el valor será 7, asi quedaría la instrucción anterior usando esta notación:

  ```
  chmod 751 /usr/grupo/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 de lograr mayor claridad del presente documento, se ejecuta el comando `grep` con el modificador `-l` para que sólo muestre el nombre y la ruta de 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. ¿Qué 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 *gabriel*:

```
adduser gabriel other /usr/gabriel
```

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 sesión *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` y `/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 12528   2682   9846   75096  20018  55078  /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	20018	55078
/dev/fd0	35	1405

Cuadro 1: Espacio Libre y Ocupado del File System

3.17. `ps`

3.17.1. Cuántos 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
 32  co  0:00 -ash
 33  c1  0:00 getty
 39  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
   32  co  0:00 -ash
   21   ?  0:00 update
   33  c1  0:00 getty
   44  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.:
En primer lugar, debemos indicar al simulador que deseamos que `/dev/fd0` referencie a la diskettera. Para ello modificamos el archivo `bochsrc.bxrc`.

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

```
ln /dev/fd0 /dev/dosA
```

Finalmente, ejecutamos `dosdir a`. La salida obtenida es:

```
APUNTE~1.TEX
GRABAC~1
CRACKS
DATOSP~1.DOC
ITR.TXT
LOQUEF~1.TXT
PELICU~1.TXT
SCIENT~1.URL
SO_150~1.ZIP
INFORME.ZIP
ABUSCA~1.TXT
FEDE
TP_POO.ZIP
```

3.21. `dosread`

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

Rta.:

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

La salida de este último comando es:

```
Esto es un archivo de prueba
```

3.22. `doswrite`

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

Rta.:

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

Luego, para ver el contenido del archivo en el diskette:

```
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:
gabriel:##gabriel:10:3:gabriel:/usr/gabriel:/usr/bin/ash
martin:##martin:11:3:martin:/usr/martin:
```

4. Uso de STDIN, STDOUT, STDERR y 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.:
Utilizamos el comando `ls -R /usr/src > /usr/grupo/fuentes.txt`, dónde "`>`" es el caracter que permite redireccionar la salida estándar al archivo `/usr/grupo/fuentes.txt`.
El modificador `-R` lo utilizamos para mostrar el contenido de los subdirectorios **R**ecursivamente.

4.1.2. wc

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

Rta.:
Utilizamos el comando `wc /usr/grupo/fuentes.txt`

La salida obtenida es:

```
1987 1909 17579 /usr/grupo/fuentes.txt
```

Esto significa, el archivo tiene:

- 1987 líneas;
- 1909 palabras; y
- 17579 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.:
Ejecutamos el comando `sort /etc/passwd >> /usr/grupo/fuentes.txt`, dónde el caracter `>>` sirve para agregar la salida estándar al archivo `/usr/grupo/fuentes.txt`, sin detruir el contenido que previamente poseía.

4.2.2. `wc`

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

Rta.:
Ejecutamos el mismo comando que en 4.1.2 y obtenemos la siguiente salida:
`1997 1926 17986 /usr/grupo/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.:
Usamos el siguiente comando: `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.:
Utilizamos `date -q < /usr/grupo/hora.txt` y obtenemos como *output*:

```
Please enter date: MMDDYYhhmmss. Then hit the RETURN key.
Fri Apr 24 23:55:00 MET DST 2020 @
```

4.3.3. Presente la fecha del sistema

Rta.:
Utilizamos el comando `date` sin parámetros y la salida es la siguiente:
`Fri Apr 24 23:56:07 MET DST 2020`

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.:
Redireccionamos la salida de errores (STDERR) estándar a un archivo:

`dosdir k 2> /usr/grupo/error.txt`,
dónde el modificador `2>` le indica al *shell* que debe redireccionar la salida estándar de error y no la salida éstandar normal. Luego de haber ejecutado dicho comando, si intentamos mostrar el contenido del archivo `/usr/<grupo>/error.tmp`, obtenemos el siguiente mensaje de error:

```
cat: cannot open error.tmp: No such file or directory
```

pues el archivo no existe.

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.:
La forma correcta de hacer lo pedido es ejecutando el siguiente comando:

```
\verb@ls -l /usr/bin/s* | grep sync | wc@
```

donde el caracter `*` indica que luego de la letra "s" puede venir 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`

Por lo tanto:

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

5. Utilitario para cuentas de usuario

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.:
Realizamos dicho script, el cual reside en el archivo `/usr/grupo/cuentas`.
Al ejecutarlo podemos pasar como parametro el número 1, el cual setea un flag de *debug*, que es muy útil al momento del desarrollo y decidimos dejarlo para que el corrector tenga la posibilidad de observar en qué secuencia se realizan las acciones. Aclaramos que en la inclusión del código en el presente informe, prescindiremos de todo el código concerniente a *debugging* con el fin de acotar la cantidad de código útil impreso.

La clave de todo el manejo de usuarios (y grupos de usuarios) en Minix radica básicamente en tres archivos:

1. `/etc/passwd`
2. `/etc/group`

3. `/etc/shadow`

La dificultad del script consistió en descubrir de qué manera se puede *filtrar* dichos archivos así tambien, cómo realizar determinadas validaciones para que el ingreso del usuario sea *mínimamente* consistente. Sin embargo, hay algunas validaciones que no realizamos puesto que se supone que el usuario no va a ingresar mal dicha información, a saber:

- No validamos que el usuario ingrese una ruta de directorio bien formada. (del tipo `/<directorio>/<subdirectorio>`) Sin embargo, si tenemos en cuenta el caso en que el directorio home que especifica el usuario no existe, lo creamos.
- No validamos si la ruta que especifica el usuario para el *shell* es bien formada o si tal *shell* existe en el sistema o no.

Como contrapartida a las licencias de validación antes explicadas, también debemos aclarar que efectuamos *muchas* validaciones de datos ingresados, algunas de ellas son:

- Validación del *username*, tanto existencia en el sistema como longitud y composición alfanumérica.
- Idem anterior para el grupo. Si el grupo no existe, se lo agrega al archivo `/etc/group`, sino se revisa dicho archivo y se extrae el *gid* correspondiente al grupo ya existente.

El script básicamente permite

- Agregar un usuario al sistema completo, con nombre de pila completo, *username*, *password*, *grupo*, *home* y *shell* particular.
- Modificar un usuario, para acceder a este submenú, lo primero que hace el programa es solicitar al usuario que ingrese el nombre del usuario que desea modificar, si dicho usuario no existe, el programa termina avisando sobre tal situación. Sino, en el menú "Modificar" podemos cambiar el nombre de pila del usuario, el *password*y *shell*.
- Borrar un usuario, validando la existencia del usuario ingresado.
- Borrar un grupo. Como ya es costumbre, aclarando su existencia en el sistema.

He aqui, el código fuente del script. Eliminamos (como ya hemos aclarado) las impresiones por pantalla de *debug*.

```
#!/usr/bin/ash
#
# Cuentas 1.0 - Administra cuentas de usuarios y grupos
#       Chiocchio - Honore - Tursi
#

DEBUG="$1"

# Necesitamos ser root.
case "`id`" in
  'uid=0('*)
;;

  *) echo " Este programa debe ser ejecutado por ROOT" >&2; exit 1
esac

opcion=0

while [ "$opcion" -ne 5 ]
do
  echo
  echo MENU PRINCIPAL
  echo --------------
  echo 1. AGREGAR UN USUARIO
  echo 2. MODIFICAR UN USUARIO
  echo 3. BORRAR UN USUARIO
  echo 4. BORRAR UN GRUPO
  echo 5. SALIR
  echo
  echo Ingrese la opcion:
  read opcion

  case "$opcion" in
  1)
############ START AGREGAR USUARIO ############

########## USUARIO ##########

    usuarioInvalido=1
    while [ "$usuarioInvalido" -eq 1 ]
```

```
    do

      echo Ingrese Usuario:
      read usuario

      # Validamos el usuario.
      len=‘expr "$usuario" : '[a-zA-Z][a-zA-Z0-9]*$'‘

      if [ "$len" -eq 0 -o "$len" -gt 8 ]
      then
      echo >&2 \
        "El nombre de usuario debe ser alfanumerico
        y no mayor a 8 caracteres"
        echo

      else
        # El nombre de usuario no debe existir previamente

        if grep "^$usuario:" /etc/passwd >/dev/null
        then
          echo "El usuario $usuario ya existe en el sistema" >&2
          echo >&2

        else
          # El usuario es valido y no existe en el sistema
          usuarioInvalido=0

        fi

      fi

    done

########## USER ID ##########

    # Encontramos el primer userID libre mayor a 10.

    uid=10
    while grep "^[^:]*:[^:]*:$uid:.*" /etc/passwd >/dev/null
    do
```

```
        uid=‘expr $uid + 1‘
    done

########## GRUPO ##########

    grupoInvalido=1

    while [ "$grupoInvalido" -eq 1 ]
    do

      echo Ingrese el Grupo:
      read grupo

      # Validamos el grupo.
      len=‘expr "$grupo" : ’[a-zA-Z][a-zA-Z0-9]*$’‘

      if [ "$len" -eq 0 -o "$len" -gt 8 ]
      then
        echo >&2 \
        "El nombre del grupo debe ser alfanumerico
        y no mayor a 8 caracteres"
        echo
      else
        grupoInvalido=0
        gid=‘sed -e "/^$grupo:/!d"
        -e ’s/^[^:]*:[^:]*:\\([^:]*\\):.*/\\1/’ /etc/group‘
      fi
    done

########## HOME ##########

    homeInvalido=1

    while [ "$homeInvalido" -eq 1 ]
    do

      echo "Ingrese el Home del usuario:"
      echo "(Ingrese la ruta absoluta completa o la ruta
      sera relativa al directorio actual"
      read home
```

```
      # Validamos el home.
      len=‘expr "$home" : '[a-zA-Z/_\.0-9][a-zA-Z/_\.0-9]*$'‘

      if [ "$len" -eq 0 ]
      then
        echo >&2 \
        "El nombre del home esta vacio o es invalido"
        echo >&2
      else
        homeInvalido=0

      fi
    done

########## SHELL ##########

    shellInvalido=1

    while [ "$shellInvalido" -eq 1 ]
    do

      echo "Ingrese el Shell del usuario:"
      echo "(Ingrese la ruta absoluta completa
      o la ruta sera relativa al directorio actual"
      read shell

      # Validamos el shell.
      len=‘expr "$shell" : '[a-zA-Z/_\.0-9][a-zA-Z/_\.0-9]*$'‘

      if [ "$len" -eq 0 ]
      then
        echo >&2 \
        "El nombre del shell esta vacio o es invalido"
        echo >&2
      else
        shellInvalido=0

      fi
    done
```

```
########## NOMBRE COMPLETO DEL USUARIO ##########
      echo Ingrese el nombre completo del usuario:
      read nombre

########## EJECUCION DE TAREAS ##########

        grupoExistia=1
        if [ `expr "$gid" : '[0-9]*$'` -eq 0 ]
        then
          grupoExistia=0

          # Encuentra el menor groupId disponible
          gid=1
          while grep "^[^:]*:[^:]*:$gid:.*" /etc/group >/dev/null
          do
            gid=`expr $gid + 1`
          done
        fi

    # Si el grupo existia no agrego nada al archivo /etc/group...
    if [ "$grupoExistia" -eq 0 ]
    then

      if [ -f /etc/group ]
      then
        echo $grupo:*:"$gid": >> /etc/group
        echo
      fi
    fi

    # Inhibimos las interrupciones.
    trap '' 1 2 3 15

    # lockeamos el archivo password,
    creando una nueva referencia, la cual usaremos luego.
    ln /etc/passwd /etc/passwd_temp || {
      echo "el archivo /etc/passwd esta ocupado, intente mas tarde"
      echo
      exit 1
```

```
  }

  # Creamos el directorio, si ya existia el comando mkdir no hace nada
  mkdir "$home" 2> /dev/null

  # Creamos el nuevo home basandonos
  en el home de Andrew S. Tanenbaum ;)

  cpdir /usr/ast "$home" || {
    rm -rf /etc/passwd_temp "$home"
    exit 1
  }

  # Cambiamos los permisos de los archivos del nuevo usuario...

  chown -R $uid:$grupo "$home" || {
    rm -rf /etc/passwd_temp "$home"
    exit 1
  }

  # Agregamos una entrada en el archivo shadow si es que existe...

  if [ -f /etc/shadow ]
  then
    echo "$usuario::0:0:::" >>/etc/shadow || {
rm -rf /etc/passwd_temp "$home"
exit 1
  }
  pwd="##$usuario"
  else
  pwd=
  fi

  # Finalmente, agregamos una entrada en el archivo /etc/passwd...

  echo "$usuario:$pwd:$uid:$gid:$nombre:$home:$shell"
                    >>/etc/passwd || {
    rm -rf /etc/passwd_temp "$home"
    exit 1
  }
```

```
    # Eliminamos el "candado" en el archivo /etc/passwd.
    rm /etc/passwd_temp || exit

    passwd "$usuario" || {
    echo
    echo "No se pudo asignarle un password al usuario $usuario"
    exit 1
    }
    echo
    echo "El usuario $usuario ha sido agregado al sistema exitosamente"
    echo "Sumario:"
    echo "-----------"
    echo "Usuario: $usuario"
    echo "UserId: $uid"
    echo "Grupo: $usuario"
    echo "GroupId: $gid"
    echo "Nombre completo del usuario: $nombre"
    echo "Directorio Home: $home"
    echo "Ruta del Shell: $shell"
    echo

############## END AGREGAR USUARIO ##############
;;

  2)
    ########## MODIFICACION DE UN USUARIO #####

    echo Ingrese el nombre del usuario a modificar:
    read username

    grep "^$username:" /etc/passwd >/dev/null || {
      echo "El usuario $username no existe en el sistema. Abortando..."
      echo
      exit 1
    }

    opcionMod=0
    while [ "$opcionMod" -ne 4 ]
    do
```

```
clr
echo MENU MODIFICAR
echo --------------
echo 1. CAMBIAR NOMBRE DE USUARIO
echo 2. CAMBIAR PASSWORD DEL USUARIO
echo 3. CAMBIAR EL SHELL DEL USUARIO
echo 4. VOLVER AL MENU PRINCIPAL
echo
echo Ingrese la opcion:
read opcionMod

clr
case "$opcionMod" in
1)
  echo Ingrese el nuevo Nombre de pila del usuario "$username":
  echo
  read nombreFantasia
  chfn "$username" "$nombreFantasia" || {
  echo "No se pudo cambiar el nombre de $username."
  echo
  echo "El nuevo nombre de $username es $nombreFantasia"
  echo
}
;;
2)
  passwd $username || {
  echo "Se ha cambiado exitosamente el password de $username."
  echo
  }
;;
3)
  echo Ingrese la ruta completa
  del shell para el usuario "$username":
  echo
  read nombreShell
  chsh "$username" "$nombreShell"
  echo "El nuevo shell de $username es $nombreShell"
;;
4)
;;
```

```
      *) echo Opcion Invalida. Intente de nuevo.
         echo
      esac

    done

;;
  3)
     ########## BORRADO DE UN USUARIO ##########

     echo Ingrese el nombre del usuario a eliminar:
     read user2Delete

     grep "^$user2Delete:" /etc/passwd >/dev/null || {
       echo "El usuario $user2Delete no existe en el sistema. Abortando..."
       exit 1
     }

     home=‘sed -e "/^$user2Delete:/!d"
     -e ’s/^[^:]*:[^:]*:[^:]*:[^:]*:[^:]*:\\([^:]*\\):.*/\\1/’
     /etc/passwd‘

     uid=‘sed -e "/^$user2Delete:/!d"
     -e ’s/^[^:]*:[^:]*:\\([^:]*\\):.*/\\1/’
     /etc/passwd‘

     grep -v ":$uid:[0-9][0-9]*:" /etc/passwd > /tmp/passwd
     cp /tmp/passwd /etc/passwd
     rm /tmp/passwd


     if [ rm -rR "$home" 2> /dev/null ]
       then
         echo "El directorio $home
         ha sido eliminado del sistema de archivos"
       else
         echo "No se pudo eliminar el directorio $home"
     fi

     grep -v "^$user2Delete:" /etc/shadow > /tmp/shadow
```

```
     cp /tmp/shadow /etc/shadow
     rm /tmp/shadow

     echo "El usuario $user2Delete ha sido
     eliminado del sistema exitosamente"
;;
4)
     ########## BORRADO DE UN GRUPO ##########

     echo Ingrese el nombre del grupo a eliminar:
     read group2Delete

     grep "^$group2Delete:" /etc/group >/dev/null || {
       echo "El grupo $group2Delete
       no existe en el sistema. Abortando..."
       echo
       exit 1
     }

     grep -v "^$group2Delete:" /etc/group > /tmp/group
     cp /tmp/group /etc/group
     rm /tmp/group
     echo "El grupo $group2Delete ha sido eliminado de /etc/group"
;;
  5)
;;
  *) echo Opcion Invalida. Intente de nuevo.
     echo
  esac

# SE CIERRA EL CICLO PRINCIPAL DEL PROGRAMA
done
```

6. Ejecución de procesos en Background

Crear el siguiente programa

`/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.:
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	32	0	2K	54K	100K		TTY
-10	ccc9	8e98	8	0	0	2K	54K	100K	ANY	SCSI
-9	ccc9	929c	8	15	0	2K	54K	100K	ANY	WINCH
-8	ccc9	94c0	8	0	0	2K	54K	100K	CLOCK	SYN_AL
-7	547	9530	0	165607	807	2K	54K	100K		IDLE
-6	ccc9	96f4	8	0	0	2K	54K	100K	ANY	PRINTER
-5	ccc9	9cd0	8	123	0	2K	54K	100K	ANY	FLOPPY
-4	ccc9	9ed4	8	14	0	2K	54K	100K	ANY	MEMORY
-3	ccc9	a114	8	32	0	2K	54K	100K	ANY	CLOCK
-2	ccc9	a314	8	54	0	2K	54K	100K	ANY	SYS
-1	0	a330	0	741	0	2K	54K	100K		HARDWAR
0	2c49	7cc8	8	9	0	1024K	1036K	44K	ANY	MM
0	6fa5	1b6c4	8	455	0	1068K	1096K	138K	ANY	FS
1	17ed	2a34	8	1	3	1206K	1206K	11K	MM	INIT
106	ea85	13030	8	0	7	142K	204K	139K	FS	ash
107	1691	40b8	8	1	1	102K	125K	23K	FS	getty
111	227	21184	0	130	7	280K	280K	134K		loop

Donde:

- **pid** es el identificador de proceso
- **sp** es el puntero al stack del proceso
- **flag** es el registro flag del procesador
- **user** es el tiempo (de usuario) que lleva el proceso corriendo en el sistema
- **text** es el puntero al principio del segmento de código del proceso
- **data** es el puntero al principio del segmento de datos del proceso. Este valor coincide con el anterior pues en Minix los procesos comparten un bloque de memoria que se asigna y libera como una unidad ("espacios I&D combinados").
- **size** es el tamaño de pila asignado al proceso

- **command** muestra el nombre del comando con el cual se invocó al proceso

Presionando la tecla F2 se exhiben los detalles de uso de memoria de los procesos (mapa de memoria).Ésta tabla, muestra los punteros a los segmentos de código, datos y stack de los procesos iniciados en ésta consola:

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	236	f9	0	32f	40	130	45f	2	139K
4	getty	0	198	18	0	1f3	10	41	234	2	23K
5	loop	0	461	0	0	461	20	210	671	7	134K

Más en detalle,

- **PROC** es un numero asignado al proceso
- **NAME** es el nombre que poseen en el bcp
- **TEXT, DATA, STACK** muestran las entradas del arreglo `mp_seg` definido en `/usr/src/mm/mproc.h`, la tabla de procesos del administrador de memoria. Cada entrada es una estructura que contiene la dirección virtual, la dirección física y la longitud del segmento, todas medidas en *clics* (típicamente 256 bytes).
- **SIZE** es el tamaño que ocupa el proceso en memoria.

7. Generación de kernel minix

a) Generar un nuevo minix manteniendo la versión original.

b) Compilar los fuentes del kernel y construir un nuevo diskette boot. Bootee alternativamente del original y los originados por Ud. (ver sugerencias al fondo de los enunciados)

Rta.:
a) y **b)**

Si ejecutamos el comando `man boot`, obtenemos información sobre los pasos que realiza el minix al bootear. Allí vemos que al arrancar la máquina virtual, carga el primer sector del dispositivo de booteo a memoria y lo ejecuta. Este sector de booteo

carga el Monitor de Minix, el cual carga el kernel desde el directorio /minix, tomando el archivo más nuevo en el caso de haber varios.
Antes de realizar cualquier acción dejamos una copia del minix actual en la carpeta /minix2, para esto realizamos lo siguiente:

```
cp /minix/* /minix2
```

Por lo tanto, los pasos a seguir para generar un nuevo Minix, conservando la imagen anterior en el directorio `/minix2`, son:

- `cd` (nos situamos en el directorio raíz)
- `mkdir /minix2` (creamos un directorio llamado `minix2` para salvar la imagen anterior)
- `cp /minix/2.0.0 /minix2` (salvamos la imagen anterior)
- `rm /minix/2.0.0` (eliminamos la actual del directorio `minix`)
- `cd /usr/src/tools` (ingresamos en dicho directorio)
- `make hdboot` (utilizamos el Makefile, que allí se encuentra . . .)

Este último comando realiza las siguientes tareas:

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

Para construir un nuevo diskette boot debemos 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
```

Dónde vemos que el comando `make` se encarga de darse cuenta que no tiene que volver a compilar ciertos archivos y saltea dichas tareas, ahorrando mucho tiempo. Nos pide cual es el dispositivo correspondiente a la diskettera (`/dev/fd0`) y finalmente, escribe en el dispositivo diskette (que en el presente trabajo hemos emulado en el archivo `minix_fs_floppy`).

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

Reiniciamos la máquina virtual y vemos cómo el sistema arranca desde el diskette satisfactoriamente.

8. Modificación de códigos

a) Modifique el "scheduler" original del MINIX para el nivel de usuarios.
Rta.:
El planificador de procesos de minix utiliza un sistema de colas multinivel con tres niveles, que corresponden a las capas 2, 3 y 4 vistas en el ejercicio 1 parte a. Dentro de las capas 2 y 3, correspondientes a los niveles de tareas y servidores de procesos, estos se ejecutan hasta que se bloquean, mientras que los procesos de usuario se planifican mediante el método *round robin*, ya que cuando son desalojados por exceso de *quantum* son puestos inmediatamente al final de la cola . Estos procesos tienen la prioridad más baja, siguiéndole los procesos servidores y más arriba las tareas. En cada *tic* del reloj se verifica si el proceso que está siendo ejecutado es un proceso de usuario y si el mismo se ha estado ejecutando por mas de 100 ms, se invoca al planificador de tareas, quien busca en la cola si existe otro proceso de usuario esperando a ser ejecutado, caso en que el proceso actualmente en ejecución pasa al final de la cola del nivel más bajo, y pasa a ejecutarse el primer proceso de dicha cola. Para el resto de las colas no existe quantum, ya que los procesos que pertenecen a las mismas nunca son desalojados.

Teniendo en cuenta todo ésto y que en el ejercicio se pide que solamente se modifique el planificador para los procesos a nivel de usuarios, nos limitaremos a modificar el quantum que determina el desalojo de un proceso de usuario, ya que de otra manera deberíamos involucrarnos con la planificación a nivel de procesos pertenecientes a otras capas.

El código de las capas 1 y 2 se encuentra en el directorio `/usr/src/kernel`, por lo que allí encontraremos el código del planificador de procesos, más específicamente en el archivo `proc.c`. En este archivo, observamos la función `sched()`, que es la encargada de desalojar a los procesos de usuario. Dicha función no es llamada directamente, sino que es ejecutada solo por medio de `lock_sched()`, que ofrece un candado para la misma, usando la variable booleana `SWITCHING`. Es una tarea del reloj invocar a dicha función, esto se puede ver en la función `do_clocktick()`, del archivo `clock.c`:

```
if(--sched_ticks == 0){
if (bill_ptr == prev_ptr) lock_sched();
/*process has run too long*/
sched_ticks = SCHED_RATE; /* reset quantum */
prev_ptr = bill_ptr;
}
```

De donde se deduce claramente que `SCHED_RATE` fija el quantum, y está definido como:

`#define SCHED_RATE (MILLISEC*HZ/1000) (línea 55 de clock.c)`

Para poder llevar a cabo lo que comentamos que ibamos a realizar con el quantum, decidimos efectuar la siguiente medición:

ayudandonos con el comando de Minix

```
time <comando>
```

que sirve para medir el tiempo de ejecución de un comando, mediremos la ejecución de un programa de prueba que realiza cinco *fork´s* dónde cada uno de los seis procesos realiza un ciclo vacío con un contador de 100.000.000, de modo que, de acuerdo al quantum que les otorgue el sistema operativo para ejecutar, todos en su conjunto terminarán con mayor o menor rapidez. Suponemos que agrandando el quantum, los procesos poseen el recurso procesador por más tiempo consecutivo con lo cual hay menor cantidad de intercambio de contexto entre ellos, y sabiendo que su E/S es nula, el tiempo real y de usuario de los procesos disminuirá.

Adicionalmente, probaremos empíricamente que lo contrario a lo anteriormente explicado ocurrirá si achicamos el quantum: los procesos serán susceptibles a mayor cantidad de desalojos (y consecuentemente, cambios de contexto) por lo cual necesariamente, tardarán más tiempo para ejecutar en su conjunto.

El código fuente de nuestra prueba [2] es:

```
El código es el siguiente:

void ciclo()
{
int contador = 100000000;
while (contador-->0);
}

int main(int argc, char* argv[])
{
if (fork() == 0)
ciclo();
else if (fork() == 0)
ciclo();
else if (fork() == 0)
ciclo();
else if (fork() == 0)
ciclo();
else if (fork() == 0)
ciclo();

else
{
wait(0);
wait(0);
wait(0);
wait(0);
wait(0);
}

return 0;
}
```

Para poder demostrar lo que supusimos, realizamos básicamente cuatro pruebas:

- con el comando `time` tomamos el tiempo del test utilizando el quantum original de Minix, para tener un valor de referencia con el cual contrastar los resultados. Aquí, el valor absoluto de `SCHED_RATE` es 6. Los tiempos son los siguientes:

[2] para saber la ubicación, ver la sección A

```
real 53.00
user 53.66
sys 0.00
```

- luego multiplicamos la constante `SCHED_RATE` por 1000, incrementando el quantum que dispone un proceso para ejecutar antes de ser desalojado, en una proporción de 1000 a 1. Así, recompilamos la imagen del kernel, *rebooteamos*, y corrimos el test (usando un quantum efectivo `SCHED_RATE = 6000`)obteniendo los siguientes tiempos: [3]

  ```
  real  35.00
  user  35.53
  sys    0.03
  ```

- intentamos realizar el ejercicio inverso al anterior, es decir, dividir por 1000 el *quantum* original de Minix pero nos dimos cuenta que si hacíamos eso la constante `SCHED_RATE` iba a poseer un valor menor a 1 y no entero (en realidad sería igual a 0,006). Es por éste motivo que decidimos que el valor mas grande por el que podíamos dividir al *quantum* original, conservándolo entero y positivo, era por 6. Este experimento consistió entonces en cambiar la línea del archivo `/usr/src/kernel/clock.c` por la siguiente:

  ```
  #define SCHED_RATE (MILLISEC*HZ/1000) / 6 /* (SCHED_RATE = 1) */
  ```

  y obtuvimos los siguientes resultados:

  ```
  real 58.00
  user 58.03
  sys 0.01
  ```

- Por último, decidimos ver que resultado obteníamos si efectuabamos el experimento contrapuesto al anterior, es decir, multiplicar el *quantum* por 6, de modo que los resultados obtenidos en las pruebas sean más fiables. Los resultados son los siguientes:

  ```
  real 51.00
  user 50.85
  sys 0.03
  ```

Podemos ver los valores recién explicados con mayor claridad en la siguiente tabla:

[3] Todos los tiempos que expondremos son producto de 3 corridas del mismo test, en igual contexto de ejecución -máquina recién booteada- y promediados para obtener un resultado estadísticamente confiable.

Valor de `SCHED_RATE`	Tiempo Real	Tiempo de Usuario	Tiempo de Sistema
$(MILLISEC * HZ/1000) * 1 = 6$	53.00	53.66	0.00
$(MILLISEC * HZ/1000) * 1000 = 6000$	35.00	35.53	0.03
$(MILLISEC * HZ/1000)/6 = 1$	58.00	58.03	0.01
$(MILLISEC * HZ/1000) * 6 = 36$	51.00	50.85	0.03

Figura 2: Resultados Modificación Quantum

Los números confirman nuestras sospechas: si aumentamos el valor del quantum, los procesos pueden ejecutar más tiempo consecutivamente sin desalojo y, debido a que son programas que usan únicamente procesador - no hay E/S - vemos que el tiempo real y de usuario disminuyen considerablemente comparados con el tiempo original que nos ofrece Minix con su scheduler nativo.
Asímismo confirmamos también que achicando el quantum el tiempo real y de usuario disminuyen como lo habíamos predicho.

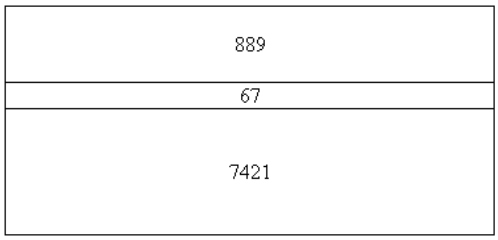
Como conclusión, podemos decir que si bien este programa, especialmente diseñado para mostrar la modificación del *scheduler*, anduvo mejor agrandando el *quantum*; esto no aplica para cualquier tipo de programa: se puede encontrar un tipo de programa dónde el agrandar el quantum funcione peor que la administración natural de Minix, aunque para el caso que elegimos nosotros, la mejora es indudable.

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.:
Para realizar el testeo de la modificación en la administración de memoria realizamos una pequeña modificación adicional, para que sea más sencillo conseguir un caso de prueba. Dicha modificación consistió en cómo asignar los huecos de memoria, originalmente Minix cuando encuentra un hueco libre en memoria en donde quepa el proceso, le otorga memoria a partir del comienzo del hueco, y modifica su posición inicial sumándole el tamaño requerido por el proceso. Nosotros le daremos la parte final del hueco libre al proceso, restándole a la longitud del hueco el tamaño de memoria requerido.

Observamos que la memoria comienza con los siguientes bloques libres:

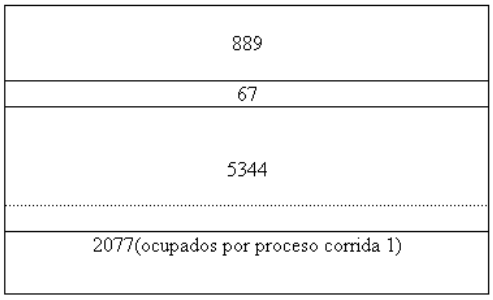
Como no pudimos crear un proceso que ocupe más de 67 clicks, y si creábamos alguno que sea mayor que 67 y menor que 889, tanto con el algoritmo de mejor ajuste como con el de primer ajuste se ubicaría en el primer hueco libre. Por lo que decidimos crear un proceso que solo pueda entrar en el último:



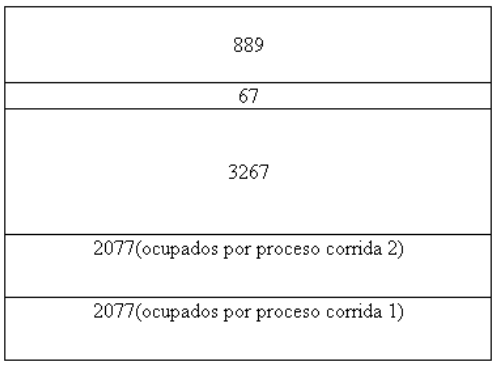

```
int a[100000]

int main(){
   while(1){
}
return 0;
}
```

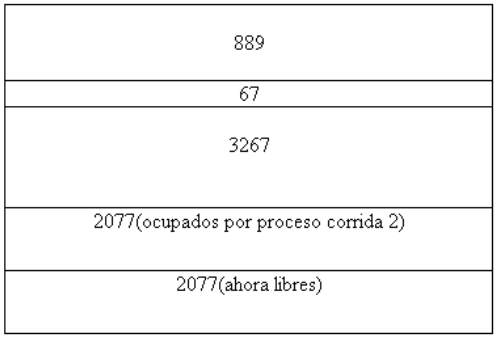
Este proceso requiere un total de 2077 clicks. Con la modificación realizada, los huecos libres quedarán distribuidos de la siguiente manera:



Luego corremos nuevamente el proceso(tener en cuenta que realizan bucles infinitos) y obtenemos lo siguiente:



Luego matamos la primer corrida del proceso, creando un hueco libre que estará al final de la lista:



Ahora, al correr nuevamente el proceso que ocupa 2077 clicks vemos que se ubica en el último hueco, a pesar de haber encontrado antes al hueco de 3267 clicks, donde hubiese entrado con el algoritmo de primer ajuste.

A continuación vemos un *dumping* de pantalla de la prueba realizada en Minix:

```
# ./proceso1 &
# Bloques:     736,  153,  67,  7421
Requerido:     2077  Desperdicio:  5344

# ./proceso1 &
# Bloques:     736,  153,  67,  5344
Requerido:     2077  Desperdicio:  3267
Bloques:       736,  153,  67,  3267
Requerido:     377   Desperdicio:  359
PID TTY  TIME CMD
33  co  0:00 -sh
34  c1  0:00 getty
39  co  0:00 ash
41  co  0:05 ./proceso1
42  co  0:02 ./proceso1
43  co  0:00 ps

# kill 41
Bloques:       736,  153,  67,  3267
Requerido:      97   Desperdicio:  56
[1] 41 Terminated          ./proceso1

# ./proceso1 &
# Bloques:     736,  153,  67,  3267,  2077,
Requerido:     2077  Desperdicio:  0
Borrando slot de tamanio           0
```

El código para implementar el método de mejor zona es el siguiente:

```
PUBLIC phys_clicks alloc_mem(clicks)
phys_clicks clicks; /* amount of memory requested */
{
/* Allocate a block of memory from the free list using first fit. The block
 * consists of a sequence of contiguous bytes, whose length in clicks is
 * given by 'clicks'.  A pointer to the block is returned.  The block is
 * always on a click boundary.  This procedure is called when memory is
 * needed for FORK or EXEC.
 */

  register struct hole *hp, *prev_ptr;
  phys_clicks old_base;

  register struct hole* anterior_mejor_zona;
  register struct hole* mejor_zona;
  phys_clicks mejor_diferencia;

  mejor_zona = NIL_HOLE;
  mejor_diferencia = 999999999;
  hp = hole_head;

  while (hp != NIL_HOLE)
  {
if (hp->h_len >= clicks && hp->h_len - clicks < mejor_diferencia)
{
anterior_mejor_zona = prev_ptr;
mejor_zona = hp;
mejor_diferencia = hp->h_len - clicks;
}

prev_ptr = hp;
hp = hp->h_next;
  }

  if (mejor_zona != NIL_HOLE)
  {
/* We found a hole that is big enough. Use it. */
old_base = mejor_zona->h_base + mejor_zona->h_len - clicks; /* Damos la parte final del hue
```

```
/*mejor_zona->h_base += clicks;*/ /* No movemos la base del hueco libre encontrado */
mejor_zona->h_len -= clicks; /* sino que le quitamos el final del mismo */

/* If hole is only partly used, reduce size and return. */
if (mejor_zona->h_len != 0) return (old_base);

/* The entire hole has been used up. Manipulate free list. */
del_slot(anterior_mejor_zona, mejor_zona);
return(old_base);
  }

  return(NO_MEM);
}
```

9. Estudio del problema de la Sección Crítica

Construya un programa principal, que utilice el recurso "pantalla" continuamente (realice la función que usted quiera, por ejemplo que imprima su nombre iterativamente, que dibuje una pelota moviéndose por la pantalla, etc.), pero que cada 4 segundos de forma repetitiva lance otro programa que deberá mostrar la fecha y hora actual (date), en el recurso "pantalla".

Ayúdese de las llamadas `FORK`, `WAITPID`, `EXEC`, `EXIT`, para ejecutar un programa que lanza la hora y las llamadas `SIGACTION`, y `ALARM`, para manejo de señales.

Plantee una solución basada en Sección Crítica, para que no exista conflicto en la utilización del recurso "pantalla" por parte de los programas, el principal y el que lanza la hora. El recurso pantalla debe utilizarse en modo exclusivo, para ello plantee una solución basada en, semáforos, o pipe, o archivo en modo exclusivo, u otra.
Indique cual utilizó y por qué. No olvide los test de prueba. (Tests o pruebas mencionados en Forma de entrega).

Rta.:
Dado que en el ejercicio 11 se nos pide implementar semáforos, aprovecharemos este ejercicio para testear su implementación. Por lo tanto, para proveer exclusión mutua del recurso pantalla utilizaremos semáforos.

El ciclo principal de nuestro programa escribe por pantalla los números del 0 al 1999 inclusive. El ciclo se ejecuta 4 veces (un número arbitrario para poner un fin al programa). Paralelamente, se activará una alarma cada 4 segundos, aunque en el programa de ejemplo lo haremos cada 1 segundo por razones de tiempo y calidad de la

prueba.

La alarma se activa la primera vez antes del ciclo principal, y el manejador de la señal de alarma (señal `SIGALARM`) es la encargada de activar una nueva alarma cada vez que se cumplió el tiempo de una. El manejador de dicha señal es una función, y su registro en el sistema se realiza llenando una estructura de tipo `sigaction` y pasándola como parámetro a la función de igual nombre.

El manejador de la señal de alarma tiene la tarea principal de mostrar la fecha por pantalla. Sin embargo, no queremos que se utilice el recurso "pantalla" para mostrar la fecha si el ciclo principal no concluyó la utilización de dicho recurso. Para lograr la exclusión mutua utilizaremos semáforos.

Antes de iniciar el ciclo principal que escribe los valores por pantalla creamos un semáforo inicializado con el valor 1. Si hacemos P de dicho semáforo antes del ciclo que imprime los números y antes de escribir la fecha en el handler, y además hacemos V del semáforo después de dichas operaciones, conseguiremos exclusión entre las dos partes que compiten por el recurso.

Sin embargo, hay ciertos detalles de implementación que agregan cierta complejidad al problema. Por un lado, queremos que la ejecución del comando que escribe la fecha se realice en un proceso distinto al proceso principal. Esto es así porque la ejecución del comando `execve`, que utilizamos para ejecutar el comando `date` TRANSFORMA el proceso llamador en el proceso que se le pidió ejecutar, y al finalizarse termina el proceso. Nosotros no queremos finalizar el proceso principal después de escribir una fecha.

Por otra parte, este comportamiento de la función `execve` hace que si colocamos un P(semáforo) antes del llamado, y un V(semáforo) después de él, esta última operación nunca se ejecutará. Por lo tanto, necesitaremos un proceso más (además del creado para que no se finalice el proceso principal junto al que escribe la fecha) que espere a que se termine el `execve` para poder hacer V(semáforo) y permitir, si corresponde, la continuación de la ejecución del ciclo principal.

El proceso que ejecuta el operador V sobre el semáforo debe ser finalizado, ya que de otra forma continuaría ejecutándose como el proceso principal, y no es lo que deseamos. Con el comando `exit` logramos terminar el proceso. Sin embargo, para poder completar el proceso de terminado, dicha operación exige que el proceso padre reconozca de alguna manera su fallecimiento (mediante el llamado a alguna función de tipo `wait`). Sin embargo, es necesario hacer una llamada asíncrona (mediante la señal `SIGCHLD`) para que el proceso principal no se quede esperando (pensar si el proceso que ejecuta la fecha se bloquea, se bloquearía también el proceso principal!). Esta señal, sin embargo, no está implementada en esta versión de Minix.

Por lo tanto, no tenemos otra opción que dejar que los procesos hijos que terminan queden como procesos *zombie* (en inglés, *defunct*, como se los ve con el comando `ps`), al menos hasta que el proceso principal concluya su ejecución.

Pasemos a la prueba. Para probar que esto realmente funciona decidimos mostrar un mensaje cuando cada alarma llega a su time-out. De esta forma, podemos ver por pantalla (o en un archivo, quizás sea más conveniente) el momento exacto en el que se produce esta acción. Es muy probable que esta señal se genere durante una iteración del ciclo principal, y esto se podrá ver dado que una iteración completa escribe todos los números entre 0 y 1999 inclusive, y el mensaje "ALARMA!!!" aparecerá entre dos números consecutivos del rango.

Sin embargo, las fechas solo se mostrarán luego de la finalización de una iteración del ciclo principal, pero nunca durante las mismas.

El código para el ejercicio es el siguiente:

```
void handler(int);

int main(int argc, char* argv[])
{
int i,j,var;
struct sigaction estr;

int semaforo = crear_sem(1, 1);

/* Inicializamos el handler para la alarma */
estr.sa_handler = &handler;
estr.sa_mask = 0;
estr.sa_flags = 0;

/* Activamos la alarma */
alarm(1);
sigaction(SIGALRM, &estr, 0);

/* Vamos a realizar cuatro veces una impresion larga en pantalla. */
/* La idea es que el codigo del handler que muestra la fecha no */
/* se ejecute en el medio de una iteracion. */
j=4;

while (j-->0){
p_sem(semaforo);
```

```
for (i = 0; i < 2000; i++)
{
printf("%i - ", i);
fflush(0);
}

v_sem(semaforo);
}

/* Esperamos dos segundos para que se muestren las fechas pendientes */
sleep(2);

liberar_sem(semaforo);
return 0;
}


void handler(int signal)
{
pid_t pid_hijo;
pid_t pid_nieto;

/* Mostramos que se genero una senal de alarma, para que se vea el */
/* momento exacto de su generacion (lo mas probable es que sea en el */
/* medio del ciclo que ejecuta el main, se quiere mostrar que el */
/* date no se ejecuta hasta que no se libere el semaforo. */
printf("\nALARMA!!!\n");
fflush(0);

/* Lanzamos una nueva alarma */
alarm(1);

/* Hacemos un fork, ya que por un lado queremos continuar con el */
/* hilo de ejecucion principal. */
pid_hijo = fork();

if (pid_hijo == 0)
{
/* Hacemos otro fork, ya que execve termina el proceso y */
```

```
/* nosotros queremos que antes de terminar se haga V del */
/* semaforo asociado. */
pid_nieto = fork();

if (pid_nieto == 0)
{
/* Adquirimos el semaforo por nombre */
int semaforo = crear_sem(1, 1);

/* Cremos la estructura para llamar a execve */
char* argv[2] = { "date", 0 };
char* envp[1] = { 0 };

p_sem(semaforo);

/* Desasociamos al proceso del semaforo */
desasociar_sem(semaforo);

execve("/usr/bin/date", argv, envp);
}

else
{
/* Adquirimos el semaforo por nombre */
int semaforo = crear_sem(1, 1);

/* Esperamos a que se termine de ejecutar date */
waitpid(pid_nieto,0,0);

v_sem(semaforo);

/* Desasociamos al proceso del semaforo */
desasociar_sem(semaforo);

/* Terminamos el proceso. Ojo, el proceso queda */
/* zombie porque el padre deberia 'enterarse' de */
/* la muerte de su hijo, pero la senal que permite */
/* hacer esto de forma asincrona (SIGCHLD) no esta */
/* implementada en esta version de Minix. De todas */
/* formas, los procesos zombies se liberan al */
```

```
/* terminar el hilo principal */
exit(0);
}
}
}
```

Los resultados obtenidos son los siguientes (analizaremos los fragmentos relevantes de la salida):

```
Ciclo 1:
ALARMA: ... - 1802 - 1803 - 1804 - ALARMA!!! - 1805 - 1806 - 1807
FECHA:  ... - 1997 - 1998 - 1999 - Tue Jul  5 12:40:43 GMT 2005 - 0-1-2-3-4-5-...

Ciclo 2:
ALARMA: ... - 1400 - 1401 - 1402 - ALARMA!!! - 1403 - 1404 - 1405 - ...
FECHA:  ... - 1997 - 1998 - 1999 - Tue Jul  5 12:40:44 GMT 2005 - 0-1-2-3-4-5-...

Ciclo 3:
ALARMA 1: ... - 759 - 760 - 761 - ALARMA!!! - 762 - 763 - 764 - ...
ALARMA 2: ... - 1913 - 1914 - 1915 - ALARMA!!! - 1916 - 1917 - 1918 - ...
FECHA  1: ... - 1997 - 1998 - 1999 - Tue Jul  5 12:40:46 GMT 2005
FECHA  2: Tue Jul  5 12:40:46 GMT 2005 - 0 - 1 - 2 - 3 - 4 - 5 - ...

Ciclo 4:
ALARMA: ... - 1437 - 1438 - 1439 - ALARMA!!! - 1440 - 1441 - 1442 - ...
FECHA:  ... - 1997 - 1998 - 1999 - Tue Jul  5 12:40:47 GMT 2005
```

Notar que durante el tercer ciclo se ejecutaron dos alarmas, y al final del ciclo se escribe dos veces la fecha antes de iniciar el próximo ciclo.

10. System Calls

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.:
Los pasos necesarios para acometer dicha tarea fueron:

- Creamos el archivo `/usr/src/lib/syscall/newcall.s`, archivo de lenguaje ensamblador que contendrá la llamada al sistema. El archivo contiene el siguiente código:

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

_newcall:
jmp     __newcall
```

- Luego, modificamos el archivo `Makefile` en dicha carpeta para que se incluya posteriormente la compilación del archivo `newcall.s`. En primer lugar, incluímos la línea
  `$(LIBRARY)(newcall.o) \`
  y luego, la línea: `$(LIBRARY)(newcall.o):  newcall.s`
  `$(CC1) newcall.s`

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

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

PUBLIC int newcall()
{
   message m;

  return(_syscall(MM, NEWCALL, &m));
}
```

- Tal como lo hicimos en el segundo punto, modificamos el archivo `Makefile` en dicha carpeta para que se incluya la compilación. Incluímos las siguientes líneas:
  `$(LIBRARY)(_newcall.o) \ ...`
  `$(LIBRARY)(_newcall.o):  _newcall.c`
  `$(CC1) _newcall.c`

- Incorporamos un nuevo prototipo al archivo
  `/usr/include/unistd.h` agregando la línea

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

- En el archivo `include/callnr.h` le asignamos el numero a nuestro *system call* mediante la línea

```
#define NEWCALL 77
```

Además, incrementamos la constante `NCALLS` en uno para que la numeración de system calls quede consistente.

- Modificamos el `archivo /usr/src/mm/table.c` agregando la línea

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

- También insertamos el elemento

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

en la tabla de funciones de atención de System Calls del MM/FS ubicada en `/usr/src/mm/proto.h`

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

- por ultimo, editamos el archivo `/usr/src/mm/Makefile` para que incluya la compilación de `newcall.c`:

```
newcall.o:      $a
newcall.o:      $h/callnr.h
newcall.o:      $i/signal.h
newcall.o:      mproc.h
```

- Ahora, luego de haber *tocado* tantos archivos, necesitamos consolidar este trabajo compilando el file system, el memory manager, librerías, e incluso el kernel mismo de nuevo. A ese fin existen en Minix dos Makefiles muy útiles, a saber:
`/usr/src/Makefile` y `/usr/src/tools/Makefile`

El primero, invocado con el parámetro `world` (previamente logueado como `bin`), compila e instala recursivamente TODO lo que se encuentre bajo el directorio `/usr/src` salvo el kernel. Para compilar e instalar el kernel sirve el segundo script, simplemente invocando `make hdboot`, crea y copia una imagen del kernel a la carpeta `\minix`.

Los comandos ejecutados fueron los siguientes:

```
su - bin
cd /usr/src
make clean
make world
cd tools
rm /minix/*
make hdboot
```

11. Primitivas para manejo de semáforos

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:
Tenemos dos procesos *P1* y *P2* que se ejecutan en paralelo y en cuyo código aparecen las siguientes operaciones:

$P1$	$P2$
cosas particulares	cosas particulares
(*) región crítica	(*) región crítica
cosas particulares	cosas particulares

En (*) los dos procesos quieren entrar a la misma región crítica. Queremos conseguir exclusión mutua sobre dicha región crítica.
SE PIDE:
Escriba una solución en código C al problema anterior utilizando su implementación y en la que el proceso que no pueda lograr el acceso a la región crítica quede en espera hasta que ésta se libere.
No olvide los test de prueba. (Tests o pruebas mencionados en Forma de entrega).

Rta.:
Introducción

Para implementar semáforos en MINIX debemos primero evaluar las estructuras a utilizar. Podemos identificar tres niveles de información:

- Información propia del semáforo (por ejemplo, su valor o los procesos que están en espera por ese semáforo);
- Información de semáforos a nivel global (por ejemplo, qué procesos usan qué semáforos);
- Información de procesos bloqueados en espera de algún semáforo a nivel sistema.

11.1. Información del semáforo

Los semáforos que implementamos son semáforos contadores. Además, para facilitar la comunicación entre procesos distintos, se permite asignar un nombre al semáforo creado de tal forma que varios procesos puedan obtener acceso al mismo semáforo.

Respecto de los procesos bloqueados en espera de un semáforo, decidimos utilizar una cola de procesos, implementada como una lista simplemente enlazada.

Por lo tanto, nuestros semáforos contendrán:

- Nombre, para facilitar la comunicación entre procesos distintos
- Valor, ya que son semáforos contadores
- Cantidad de procesos bloqueados en espera del semáforo
- Cola de procesos bloqueados en espera del semáforo

Veamos la estructura de un semáforo escrita en C (`kernel/sem.h`):

```
struct Semaforo
{
bool estaEnUso;

int id;
int valor;

int cantProcesosQueMeUsan;
int cantProcesosBloqueados;
};
```

Dado que la cola de procesos bloqueados es una lista simplemente enlazada de procesos, debemos almacenar un puntero al primero y al último proceso de la cola. Además, se debe agregar en la estructura `proc` un puntero al próximo proceso bloqueado por el semáforo (`kernel/proc.h`):

```
struct proc {
...
struct proc* proximoProcesoBloqueado;
...
};
```

Dado que en la implementación que elegimos se utiliza la estructura `proc` en la representación de un semáforo, y dicha estructura se encuentra solo disponible a nivel del núcleo, la estructura se encontrará del lado del kernel, al igual que la implementación de todas las operaciones sobre semáforos. Las llamadas serán realizadas por el usuario a través del File System, con lo cual deberemos proveer un mecanismo para comunicar las operaciones implementadas en el kernel con el File System. Hablaremos de ello más adelante.

11.2. Información de semáforos a nivel global

Necesitamos un arreglo global de semáforos que contenga la información de todos los semáforos del sistema. Definimos una constante `MAX_SEMAFOROS` con el valor arbitrario 50 (`include/minix/config.h`):

```
#define MAX_SEMAFOROS 50
```

Y declaramos el arreglo (`kernel/sem.c`):

```
Semaforo g_semaforos[MAX_SEMAFOROS];
```

La operación `is_sem` requiere tener el conocimiento de qué procesos están utilizando qué semáforos. Para proveer esta información utilizaremos una matriz de enteros de tamaño `NR_PROCS * MAX_SEMAFOROS`, donde `NR_PROCS` es la cantidad máxima de procesos en el sistema. En la posición (i, j) se tiene un 1 si el proceso i está utilizando el semáforo j, y un 0 en caso contrario (`kernel/sem.c`):

```
int g_procesosQueUsanSemaforos[NR_PROCS][MAX_SEMAFOROS];
```

11.3. Información de procesos bloqueados en espera de algún semáforo

Los operadores P y V tienen la capacidad de bloquear o desbloquear procesos, respectivamente, mediante las operaciones `signal` y `wait`. La estrategia que seguimos para realizar estos bloqueos consiste en utilizar un conjunto de procesos bloqueados por semáforo que será recorrido por el sistema en cada selección del próximo proceso a ejecutar (operación `pick_proc`). Si un candidato se encuentra dentro de este conjunto, no será ejecutado.

Dado que la cantidad de procesos bloqueados no puede superar la cantidad máxima de procesos del sistema, utilizaremos un arreglo para representar este conjunto. Una forma de implementar esto sería con un arreglo de booleanos, donde cada posición indica un número de proceso y hay un "verdadero" en la posición i si el proceso i está bloqueado por algún semáforo, y hay un "falso" en caso contrario. Sin embargo, esta implementación nos obliga a recorrer todo el arreglo cada vez que querramos verificar si un proceso determinado está o no bloqueado. Ejemplo:

Arreglo: [0,0,0,1,0,0,0,1,1,0] $\Rightarrow$ Identifica que los procesos 4, 8 y 9 (si se numera desde 1) están bloqueados, los demás no.

Otra posible implementación consiste en un arreglo de enteros donde las posiciones no identifican nada, pero cada valor del arreglo tiene un número de proceso. Se tiene un contador de cuántos procesos bloqueados hay. Al bloquearse un proceso, se lo agrega

después del último valor válido y se incrementa el contador. Al liberarse un proceso, se "saca" el valor del último proceso en el arreglo y se lo coloca en la posición que tenía el proceso liberado, pisando el viejo valor. Luego se decrementa en contador en uno. De esta forma no es necesario recorrer toda la lista, solo basta con recorrer tantas posiciones como procesos bloqueados haya. Ejemplo.

Valores iniciales:

Arreglo: [1,4,2,0,0,0,0,0,0,0] Cantidad de procesos bloqueados: 3

Bloqueamos el proceso 3:

Arreglo: [1,4,2,3,0,0,0,0,0,0] Cantidad de procesos bloqueados: 4

Liberamos el proceso 1:

Arreglo: [3,4,2,3,0,0,0,0,0,0] Cantidad de procesos bloqueados: 3

Notar que en este último paso no es necesario poner un cero en la cuarta posición porque el contador identifica la cantidad de números válidos en el arreglo.

Finalmente, la estructura es la que sigue (`kernel/sem.c`):

```
int g_cantidadProcesosBloqueados;
int g_procesosBloqueados[NR_PROCS];
```

11.4. Inicialización de las estructuras

Para inicializar las estructuras se deben realizar las siguientes acciones:

a) Marcar todos los semáforos como libres (es decir, disponibles para su uso).

b) Llenar la matriz que indica qué procesos utilizan qué semáforos con valores 'falso'.

c) Indicar para todos los procesos que no hay ningún próximo proceso bloqueado.

d) Inicializar el conjunto de procesos bloqueados por semáforo, poniendo el contador en cero (no es necesario inicializar los valores del arreglo).

Todo ello lo realizaremos en la función 'main'(`kernel/main.c`):

a)
```
for (i = 0; i < MAX_SEMAFOROS; i++)
    g_semaforos[i].estaLibre = 0;
```

b)
```
for (i = 0; i < NR_PROCS; i++)
    for (j = 0; j < MAX_SEMAFOROS; j++)
       g_procesosQueUsanSemaforos[i][j] = 0;
```

c)
```
for (rp = BEG_PROC_ADDR, t = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++t) {
    ...
    rp->proximoProcesoBloqueado = NULL;
    ...
}
```

d)
```
g_cantidadProcesosBloqueados = 0;
```

Además, cuando se ejecuta `do_fork` se realiza una copia de la estructura `proc` del padre al hijo, modificando luego algunos elementos de la nueva estructura. Debemos poner el puntero al próximo proceso bloqueado (del proceso hijo) en NULL (`kernel/system.c`):

rpc-¿proximoProcesoBloqueado = NULL;

11.5. Bloqueo y desbloqueo de procesos

La operación `pick_proc` se modificó de la siguiente manera (`kernel/proc.c`):

```
if ( (rp = rdy_head[USER_Q]) != NIL_PROC)) {
-->
if ( (rp = rdy_head[USER_Q]) != NIL_PROC && !bloqueado(rp)) {
```

La operación "bloqueado" analiza si un proceso está o no en el conjunto de procesos bloqueados en espera de algún semáforo. Por lo tanto, con esta línea estamos diciendo que sólo elegiremos al proceso en caso de que no pertenezca al conjunto de bloqueados. La función "bloqueado" se muestra a continuación (`kernel/proc.c`):

```
PRIVATE int bloqueado(rp)
register struct proc *rp;
{
  int i;
  int estaBloqueado = 0;

  for (i = 0; i < g_cantidadProcesosBloqueados; i++)
  {
    if (g_procesosBloqueados[i] == rp->p_nr)
    {
      estaBloqueado = 1;
      break;
    }
  }
```

```
  return estaBloqueado;
}
```

Las funciones para bloquear y desbloquear procesos son `proc_wait` y `proc_signal`, respectivamente (`kernel/sem.c`):

```
PRIVATE void proc_wait(nproc)
int nproc;
{
  /* Insertamos el numero de proceso a bloquear al final del arreglo */
  g_procesosBloqueados[g_cantidadProcesosBloqueados] = nproc;
  g_cantidadProcesosBloqueados++;
}


PRIVATE int proc_signal(nproc)
int nproc;
{
  int i;

  for (i = 0; i < g_cantidadProcesosBloqueados; i++)
  {
    if (g_procesosBloqueados[i] == nproc)
    {
      /* Pisamos el numero del proceso liberado con el último del arreglo */
      g_procesosBloqueados[i] = g_procesosBloqueados[g_cantidadProcesosBloqueados];

      g_cantidadProcesosBloqueados--;
      break;
    }
  }
}
```

11.6. Operaciones sobre semáforos

Las operaciones implementadas son las siguientes:

- `crear_sem`: Dado un nombre N y un valor X, se busca un semáforo ya inicializado cuyo nombre sea N. Si se encuentra, se devuelve el identificador asociado a ese

semáforo; si no se encuentra, genera un nuevo semáforo buscando el primero que esté libre en el arreglo se semáforos y devuelve el identificador.
NOTA: Este llamado debe ser realizado por TODOS los procesos que deseen utilizar el semáforo (se realiza una asociación entre proceso y semáforo). Si no se hiciera así un proceso podría utilizar un semáforo que no le corresponde. Además, si el semáforo fue identificado por nombre (es decir, ya fue inicializado por alguien) el valor inicial se deja como estaba.

- `val_sem`: Devuelve el valor del semáforo.
- `nproc_sem`: Indica la cantidad de procesos que utilizan un semáforo dado.
- `is_sem`: Indica si el semáforo está asociado al proceso.
- `p_sem`: El operador P. Actualiza la cola de procesos bloqueados asociada al semáforo en caso de ser necesario.
- `v_sem`: El operador V. Actualiza la cola de procesos bloqueados asociada al semáforo en caso de ser necesario.
- `desasociar_sem`: Esta operación desasocia al proceso de un semáforo que adquirió mediante la operación `crear_sem`. Esta operación se provee para aumentar la consistencia general del sistema de semáforos. Veámoslo con un ejemplo: Consideremos un proceso que hace `fork()` y este proceso hijo adquiere un semáforo. Si no lo desasociara antes de terminar, y luego de terminar entra otro proceso con el mismo número (no PID, que es único), este último proceso podría utilizar el semáforo sin haberlo adquirido.
- `liberar_sem`: Libera a todos los procesos bloqueados en el semáforo y lo establece como libre.

Como se puede ver, decidimos no implementar la función `set_sem`. Esto se debe a que una función de este tipo permitiría al usuario de estas operaciones establecer el valor de un semáforo en valores arbitrarios y se producirían inconsistencias. Al asignar el valor inicial en el momento de la creación, solo se permite alterar el valor de los semáforos mediante P, V y `liberar_sem`, manteniendo la consistencia.

El código de las operaciones es el siguiente (`/usr/src/kernel/sem.c`):

```
/* es_valido: indica si el identificador del semaforo esta en rango */
/*==========================================================*/
PRIVATE int es_valido(int sem_id)
{
  return sem_id >= 0 && sem_id < MAX_SEMAFOROS;
}
```

```
/* me_corresponde: indica si el semaforo esta asociado a un proceso dado */
/*=========================================================*/
PRIVATE int me_corresponde(int sem_id, int nproc)
{
  return g_procesosQueUsanSemaforos[nproc][sem_id] != 0;
}


/* proc_signal: despierta a un proceso bloqueado por
semaforo sacandolo del conjunto de procesos bloqueados */
/*=========================================================*/
PRIVATE void proc_signal(nproc)
int nproc;
{
  int i;

  for (i = 0; i < g_cantidadProcesosBloqueados; i++)
  {
    if (g_procesosBloqueados[i] == nproc)
    {
      g_cantidadProcesosBloqueados--;

      /* Pisamos el numero del proceso liberado con el ultimo del arreglo */
      g_procesosBloqueados[i] =
                    g_procesosBloqueados[g_cantidadProcesosBloqueados - 1];

      break;
    }
  }
}


/* proc_wait: duerme un proceso insertandolo en el conjunto de procesos */
/* bloqueados por semaforo */
/*=========================================================*/
PRIVATE void proc_wait(nproc)
int nproc;
{
  /* Insertamos el numero del proceso a bloquear al final del arreglo */
```

```
  g_procesosBloqueados[g_cantidadProcesosBloqueados] = nproc;
  g_cantidadProcesosBloqueados++;
}


/*=========================================================*/
PUBLIC int do_crear_sem(m_ptr)
register message *m_ptr;
{
  int nombre = m_ptr->m1_i1;
  int valorInicial = m_ptr->m1_i2;
  int nproc = m_ptr->m1_i3;

  int encontreNombre = 0;

  int ret;

  int i = 0;
  int primeroLibre = -1;

  /* Buscamos el nombre y llevamos cuenta del primer semaforo
     libre para utilizarlo en caso de que el nombre no aparezca */
  while (i < MAX_SEMAFOROS && !encontreNombre)
  {
    if (g_semaforos[i].estaLibre != 0)
    {
      if (primeroLibre == -1)
        primeroLibre = i;
    }

    else
      encontreNombre = g_semaforos[i].nombre == nombre;

    if (!encontreNombre)
      i++;
  }

  if (encontreNombre)
  {
    /* Si lo pide un proceso que no habia adquirido el semaforo, */
```

```
    /* modificamos la matriz para reflejar los cambios */
    if (g_procesosQueUsanSemaforos[nproc][i] == 0)
      g_procesosQueUsanSemaforos[nproc][i] = 1;

    ret = i;
  }

  else
  {
    if (primeroLibre == -1)
      ret = E_NO_HAY_SEMAFOROS_DISPONIBLES;

    else
    {
      /* Construimos un nuevo semaforo */
      g_semaforos[primeroLibre].estaLibre = 0;
      g_semaforos[primeroLibre].nombre = nombre;
      g_semaforos[primeroLibre].valor = valorInicial;
      g_semaforos[primeroLibre].nProcesos = 0;
      g_semaforos[primeroLibre].colaDeBloqueados_head = NULL;
      g_semaforos[primeroLibre].colaDeBloqueados_tail = NULL;
      g_procesosQueUsanSemaforos[nproc][primeroLibre] = 1;
      ret = primeroLibre;
    }
  }

  return ret;
}


/*==========================================================*/
PUBLIC int do_val_sem(m_ptr)
register message *m_ptr;
{
  int sem_id = m_ptr->m1_i1;
  int nproc = m_ptr->m1_i2;
  int ret;

  if (!es_valido(sem_id) || !me_corresponde(sem_id, nproc))
    ret = E_SEMAFORO_INVALIDO;
```

```
  else
    ret = g_semaforos[sem_id].valor;

  return ret;
}


/*==========================================================*/
PUBLIC int do_nproc_sem(m_ptr)
register message *m_ptr;
{
  int sem_id = m_ptr->m1_i1;
  int nproc = m_ptr->m1_i2;
  int ret;

  if (!es_valido(sem_id) || !me_corresponde(sem_id, nproc))
    ret = E_SEMAFORO_INVALIDO;

  else
    ret = g_semaforos[sem_id].nProcesos;

  return ret;
}


/*==========================================================*/
PUBLIC int do_is_sem(m_ptr)
register message *m_ptr;
{
  int sem_id = m_ptr->m1_i1;
  int nproc = m_ptr->m1_i2;
  int ret;

  if (!es_valido(sem_id))
    ret = E_SEMAFORO_INVALIDO;

  else
    ret = g_procesosQueUsanSemaforos[nproc][sem_id] != 0;
```

```
  return ret;
}


/*==========================================================*/
PUBLIC int do_p_sem(m_ptr)
register message *m_ptr;
{
  int sem_id = m_ptr->m1_i1;
  int nproc = m_ptr->m1_i2;
  int ret;

  struct proc* pr;

  if (!es_valido(sem_id) || !me_corresponde(sem_id, nproc))
    ret = E_SEMAFORO_INVALIDO;

  else
  {
    g_semaforos[sem_id].valor--;

    if (g_semaforos[sem_id].valor < 0)
    {
      /* Colocamos el numero de proceso en el arreglo
      de procesos bloqueados por algun semaforo */
      proc_wait(nproc);

      /* Actualizamos la cola de procesos bloqueados,
      colocando al proceso al final.
      pr = proc_addr(nproc);

      if (g_semaforos[sem_id].colaDeBloqueados_head != 0)
        g_semaforos[sem_id].colaDeBloqueados_tail->proximoProcesoBloqueado = pr;

      else
        g_semaforos[sem_id].colaDeBloqueados_head = pr;

      g_semaforos[sem_id].colaDeBloqueados_tail = pr;

    }
```

```
    ret = 0;
  }

  return ret;
}


/*==========================================================*/
PUBLIC int do_v_sem(m_ptr)
register message *m_ptr;
{
  int sem_id = m_ptr->m1_i1;
  int nproc = m_ptr->m1_i2;
  int ret;

  struct proc* aux;

  if (!es_valido(sem_id) || !me_corresponde(sem_id, nproc))
    ret = E_SEMAFORO_INVALIDO;

  else
  {
    g_semaforos[sem_id].valor++;

    if (g_semaforos[sem_id].valor <= 0)
    {
      g_semaforos[sem_id].nProcesos--;
      aux = g_semaforos[sem_id].colaDeBloqueados_head;

      /* Obtenemos el numero del primer proceso en la cola y lo
      sacamos del arreglo de procesos bloqueados por algun semaforo */
      proc_signal(aux->p_nr);

      /* Actualizamos la cola de bloqueados, sacando el primer proceso en */
      /* la misma y poniendo en NULL el puntero al proximo bloqueado de ese */
      /* mismo proceso */
      g_semaforos[sem_id].colaDeBloqueados_head = aux->proximoProcesoBloqueado;
      aux->proximoProcesoBloqueado = NULL;
```

```
      if (g_semaforos[sem_id].colaDeBloqueados_head == NULL)
        g_semaforos[sem_id].colaDeBloqueados_tail = NULL;
    }

    ret = 0;
  }
  return ret;
}

/*==========================================================*/
PUBLIC int do_liberar_sem(m_ptr)
register message *m_ptr;
{
  int sem_id = m_ptr->m1_i1;
  int nproc = m_ptr->m1_i2;

  int ret;

  message m;

  if (!es_valido(sem_id) || !me_corresponde(sem_id, nproc))
    ret = E_SEMAFORO_INVALIDO;

  else
  {
    int i;

    /* Aplicamos el operador V al semaforo tantas veces
    como procesos en espera por ese semaforo haya */
    while (g_semaforos[sem_id].valor < 0)
    {
      m.m1_i1 = sem_id;
      m.m1_i2 = nproc;
      do_v_sem(&m);
    }

    /* Liberamos el semaforo, marcandolo como libre
    para poder reutilizarlo luego */
    g_semaforos[sem_id].estaLibre = 1;
```

```
    /* Desasociamos a todos los procesos del semaforo poniendo
    un cero en la posicion correspondiente de la matriz */
    for (i = 0; i < NR_PROCS; i++)
      g_procesosQueUsanSemaforos[i][sem_id] = 0;

    ret = 0;
  }

  return ret;
}
```

11.7. Implementación de las llamadas al FS y comunicación FS-Kernel

Las llamadas al File System se implementaron de la misma manera que en el ejercicio 10, con la diferencia de que no creamos archivos distintos para cada función en `/usr/src/lib/syscall` y `/usr/src/lib/posix` (en realidad hacerlo sería la forma correcta, pero trabajando desde la consola nos resultó mucho más eficiente agruparlas).

Sin embargo, dado que la implementación efectiva de todas las operaciones se encuentra a nivel del kernel, debemos habilitar el acceso de dichas operaciones al File System. Para ello se siguieron los siguientes pasos:

- `/usr/src/lib/syslib/sys_sem.c`: `/usr/src/lib/syslib/sys_sem.c`: De la misma manera que en `/usr/src/lib/posix`, se crean los llamados para las distintas funciones. Estas funciones son `sys_crear_sem`, `sys_val_sem`, `sys_nproc_sem`, `sys_is_sem`, `sys_p_sem`, `sys_v_sem`, `sys_desasignar_sem` y `sys_liberar_sem`. Estas funciones deben "empaquetar" los parámetros en un mensaje y hacer un `_taskcall` (en vez de un `_syscall`) mandando el código del llamado y el mensaje. Algunas de ellas reciben un parámetro adicional que es el número del proceso, que es pasado por el FS (variable global "who") al hacer cada llamado.
- `/usr/include/minix/syslib.h`: Agregar los prototipos de los llamados `sys_xxx`.
- `/usr/include/minix/com.h`: Agregar los códigos de los nuevos llamados.
- `/usr/src/kernel/system.c`: Agregar en la función `sys_task` las cláusulas *case* que permiten, dado un número de función, efectuar la llamada al kernel correspondiente (`do_crear_sem`, por ejemplo).

De esta forma, las operaciones `do_crear_sem`, `do_val_sem`, `do_nproc_sem`, `do_is_sem`, `do_p_sem`, `do_v_sem` y `do_liberar_sem` del lado del File System (`fs/sem.c`) delegan la

tarea al kernel, haciendo cada una el llamado a la función `sys_xxx` correspondiente.

11.8. Resumen general de archivos creados y modificados

Para implementar las llamadas al File System: (los pasos son los del ejercicio 10)

- Creado `/usr/src/lib/syscall/_sem.s`
- Creado `/usr/src/lib/posix/_sem.c`
- Modificado `/usr/src/fs/table.c`
- Modificado `/usr/src/fs/proto.h`
- Modificado `/usr/src/mm/table.c`
- Creado `/usr/src/fs/sem.c`
- Modificado `/usr/include/unistd.h`
- Modificado `/usr/include/minix/callnr.h`

Para implementar la comunicación entre Kernel y File System:

- Creado `/usr/src/lib/syslib/sys_sem.c` (funciones que serán llamadas desde el FS, empaquetan parámetros en mensajes y hacen un `_taskcall`)
- Modificado `/usr/include/minix/syslib.h` (prototipos de las funciones implementadas en `sys_sem`)
- Modificado `/usr/include/minix/com.h` (definición de los códigos de llamada)
- Modificado `/usr/src/kernel/system.c` (llamada desde `sys_task` a la funcion que corresponda con el código de llamada)

Implementación de semáforos:

- Creado `/usr/src/kernel/sem.c` (implementación de las operaciones sobre semáforos y definición de variables globales)
- Creado `/usr/src/kernel/sem.h` (definición de la estructura Semaforo y prototipos)
- Modificado `/usr/include/errno.h` (códigos de error)
- Modificado `/usr/include/minix/config.h` (`MAX_SEMAFOROS`)
- Modificado `/usr/src/kernel/proc.c` (implementación del conjunto de bloqueados)
- Modificado `/usr/src/kernel/main.c` (inicializacion de las estructuras)

Y casi todos los Makefiles ;-)

11.9. Pruebas

Para un testeo básico de la implementación vamos a realizar dos pruebas. En otros ejercicios se la pondrá a pruebas más sofisticadas, como lo son el caso del ejercicio 9 (para manejar exclusión mutua sobre una misma región crítica -recurso pantalla-) y el ejercicio 12 (el problema del productor-consumidor).

11.9.1. Primera Prueba

La primera prueba pretende analizar de forma muy básica el funcionamiento de las operaciones `crear_sem`, `is_sem`, `val_sem` y los operadores P y V, por ahora sin bloqueo de procesos. Para ello se realiza `is_sem` sobre semáforos no asignados al proceso y sobre los asignados también. Se aplican las primitivas P y V para verificar que los contadores se incrementan o decrementan según corresponda, y se muestran los valores finales de los semáforos.

El código del test es:

```
void es_semaforo(int sem_id)
{
if (is_sem(sem_id))
printf("%1d es un semaforo valido.\n", sem_id);

else
printf("%1d no es un semaforo valido.\n", sem_id);

fflush(0);
}

int main(int argc, char* argv[])
{
int sem0 = 0;
int sem1 = 1;
int sem2 = 2;

es_semaforo(sem0);
es_semaforo(sem1);
es_semaforo(sem2);

printf("Creando semaforo 0 con nombre 5 y valor inicial 3.\n");
fflush(0);
```

```
sem0 = crear_sem(5, 3);
es_semaforo(sem0);
es_semaforo(sem1);
es_semaforo(sem2);

printf("Creando semaforo 1 con nombre 2 y valor inicial 6.\n");
fflush(0);
sem1 = crear_sem(2, 6);
es_semaforo(sem0);
es_semaforo(sem1);
es_semaforo(sem2);

printf("Creando semaforo 2 con nombre 9 y valor inicial 1.\n");
fflush(0);
sem2 = crear_sem(9, 1);
es_semaforo(sem0);
es_semaforo(sem1);
es_semaforo(sem2);

printf("\n");

printf("Aplicando P al semaforo 1...\n");
p_sem(sem1);

printf("Aplicando V al semaforo 2...\n");
v_sem(sem2);

printf("\n");
printf("Nuevos valores:\n");

printf("* Valor(0): %i\n", val_sem(sem0));
printf("* Valor(1): %i\n", val_sem(sem1));
printf("* Valor(2): %i\n", val_sem(sem2));
fflush(0);

liberar_sem(sem0);
liberar_sem(sem1);
liberar_sem(sem2);

return 0;
```

```
}
```

NOTA: Este código asume que no hay otros procesos utilizando semáforos en el instante de su ejecución (por eso asumimos en el código que los identificadores de los semáforos serán 0, 1 y 2).

El resultado obtenido fue:

```
0 no es un semaforo valido.
1 no es un semaforo valido.
2 no es un semaforo valido.
Creando semaforo 0 con nombre 5 y valor inicial 3.
0 es un semaforo valido.
1 no es un semaforo valido.
2 no es un semaforo valido.
Creando semaforo 1 con nombre 2 y valor inicial 6.
0 es un semaforo valido.
1 es un semaforo valido.
2 no es un semaforo valido.
Creando semaforo 2 con nombre 9 y valor inicial 1.
0 es un semaforo valido.
1 es un semaforo valido.
2 es un semaforo valido.

Aplicando P al semaforo 1...
Aplicando V al semaforo 2...

Nuevos valores:
* Valor(0): 3
* Valor(1): 5
* Valor(2): 2
```

11.9.2. Segunda Prueba

La segunda prueba consiste en un productor-consumidor con un buffer que almacena un solo producto por vez. El proceso A escribe una serie de caracteres en pantalla. El proceso B escribe otra serie de caracteres en pantalla. Se quiere conseguir una secuencia de ejecución `BABABABA...BA` sin que las secuencias se mezclen. Aquí pretendemos ver cómo compartir semáforos entre distintos procesos y ver el funcionamiento del sistema de bloqueo de procesos.

Para esta segunda prueba realizamos tres programas: uno es solo el proceso A (que se quedará esperando a que el proceso B ejecute una iteración), otro es solo el proceso B, y uno tercero que ejecuta los procesos A y B como padre e hijo. Mostramos el código de esta tercer variante:

```
void proceso1()
{
int x = crear_sem(10, 1);
int y = crear_sem(11, 0);

int cantidad = 10;

while (cantidad > 0)
{
p_sem(y);
printf("A");
fflush(0);
printf("B");
fflush(0);
printf("C");
fflush(0);
v_sem(x);

cantidad--;
}

liberar_sem(x);
liberar_sem(y);
}

void proceso2()
{
int x = crear_sem(10, 1);
int y = crear_sem(11, 0);

int cantidad = 10;

while (cantidad > 0)
{
p_sem(x);
```

```
printf("1");
fflush(0);
printf("2");
fflush(0);
printf("3");
fflush(0);
v_sem(y);

cantidad--;
}
}

int main(int argc, char* argv[])
{
if (fork() == 0)
proceso1();

else
proceso2();

return 0;
}
```

El resultado obtenido fue:

```
123ABC123ABC123ABC123ABC123ABC123ABC123ABC123ABC123ABC123ABC
```

Como se puede observar, las secuencias no se solapan y hay sincronía entre los procesos.

12. Productor-Consumidor

Con las herramientas ya generadas resuelva un caso de productor - consumidor. No olvide los test de prueba. (Tests o pruebas mencionados en Forma de entrega). **Rta.:** Para resolver un caso de productor consumidor primero pensamos en un algoritmo típico, que consistía en que ambos procesos tengan un buffer en común, donde el productor pueda escribir lo que produce, y de donde el consumidor lo fuese leyendo.

El primer intento consistió en crear un proceso hijo que consuma mientras el padre produce, utilizando un buffer de memoria en común. Luego de algunos intentos nos dimos cuenta que en realidad el hijo no tenía acceso al buffer, ya que en minix (como en

Unix) los procesos hijos tienen memoria independiente, porque la comunicación solo es posible por medio de archivos.
Para subsanar este inconveniente, simplemente no utilizamos buffer(a pesar de no ser conceptualmente la mejor opción). Por otro lado agregamos un delay tanto al productor como al consumidor, para simular distintas velocidades de proceso y poder analizar distintos comportamientos. En particular, analizamos el comportamiento cuando el consumidor es más lento que el productor (en cuyo caso el consumidor siempre tendrá disponible algo para consumir), y también el caso inverso (en cuyo caso el consumidor deberá esperar a que se produzca algo).

El código para productor-consumidor es el siguiente:

```
#define BUFFER_SIZE 5
#define MAX_ITERATIONS BUFFER_SIZE*2

/* Delay del productor y el consumidor. Para realizar la otra prueba */
/* (es decir, consumidor más rápido que el productor) se dan vuelta */
/* los valores. */
#define DELAY_PROD 500000
#define DELAY_CONS 1000000

void productor();
void consumidor();

void delay(int valor){
while(valor-- > 0);
}

int main(int argc, char* argv[]) {
int exclusion = crear_sem(1,1);
int hay_lugar = crear_sem(2,BUFFER_SIZE);
int hay_algo = crear_sem(3,0);

if(fork() == 0){
/*Proceso hijo*/
consumidor();
}
else
{
/*Proceso padre*/
productor();
```

```
}

wait(0);
wait(0);

liberar_sem(hay_lugar);
liberar_sem(hay_algo);
liberar_sem(exclusion);
}

void productor(){
int exclusion = crear_sem(1,1);
int hay_lugar = crear_sem(2,BUFFER_SIZE);
int hay_algo = crear_sem(3,0);

int i;
int producido = 0;

for(i = 0; i < MAX_ITERATIONS; i++){
p_sem(hay_lugar);
p_sem(exclusion);

printf("Produciendo %i\n", producido);
fflush(0);
producido++;

v_sem(exclusion);
v_sem(hay_algo);

delay(DELAY_PROD);
}
}

void consumidor(){
int exclusion = crear_sem(1,1);
int hay_lugar = crear_sem(2,BUFFER_SIZE);
int hay_algo = crear_sem(3,0);

int h;
int consumido = 0;
```

```
for(h = 0; h < MAX_ITERATIONS; h++){
p_sem(hay_algo);
p_sem(exclusion);

printf("Consumiendo: %i\n", consumido);
fflush(0);
consumido++;

v_sem(exclusion);
v_sem(hay_lugar);

delay(DELAY_CONS);
}
}
```

Los resultados obtenidos son:

Para el productor más rápido que el consumidor:

```
Produciendo: 0
Consumiendo: 0
Produciendo: 1
Produciendo: 2
Consumiendo: 1
Produciendo: 3
Produciendo: 4
Consumiendo: 2
Produciendo: 5
Produciendo: 6
Consumiendo: 3
Produciendo: 7
Produciendo: 8
Consumiendo: 4
Produciendo: 9
Consumiendo: 5
Consumiendo: 6
Consumiendo: 7
Consumiendo: 8
Consumiendo: 9
```

Para el consumidor más rápido que el productor:

```
Produciendo 0
Consumiendo: 0
Produciendo 1
Consumiendo: 1
Produciendo 2
Consumiendo: 2
Produciendo 3
Consumiendo: 3
Produciendo 4
Consumiendo: 4
Produciendo 5
Consumiendo: 5
Produciendo 6
Consumiendo: 6
Produciendo 7
Consumiendo: 7
Produciendo 8
Consumiendo: 8
Produciendo 9
Consumiendo: 9
```

13. Lectores-Escritores(Opcional)

Resuelva el problema de Lectores/Escritores con prioridad Lectores. (Tests o pruebas mencionados en Forma de entrega).

14. Filósofos Chinos(Opcional)

Resuelva el problema de los Filósofos Chinos con la herramienta adecuada. (Tests o pruebas mencionados en Forma de entrega).

15. Aplicación que muestra un inodo(Opcional)

Genere una aplicación que muestre el contenido del inodo de un archivo cuyo `\$HOME/nombre` se pasa como parámetro.

A. Ubicación y breve descripción de las pruebas

Las pruebas se encuentran en el directorio `tests` de la imagen. Allí se encuentran los siguientes subdirectorios:

- 8a
- 9
- 11
- 12
- 14

Cada uno de ellos contiene los archivos de prueba de los ejercicios correspondientes.

8a:

- `test`: Prueba que ejecuta cinco procesos simultaneamente durante un intervalo de tiempo.

9:

- `test1seg`: Prueba que muestra exclusión mutua para el acceso a un recurso con semáforos. Alarma cada 1 segundo.
- `test4seg`: Idem a test1seg pero con alarmas cada 4 segundos.
- `res_test1seg`: Salida de test1seg.
- `res_test4seg`: Salida de test4seg.

11:

- `test1`: Muestra el funcionamiento de las operaciones más básicas de los semáforos.
- `procA` y `procB`: Procesos que actúan sincronizados, ejecutandose en secuencia `ABABAB...AB`
- `procsFork`: Ejecuta las funciones de procA y procB pero ejecutándolos como procesos padre e hijo (se usa fork).
- `res_test1`: Salida de test1.
- `res_procsFork`: Salida de procsFork

12:

- `prodcons1`: Ejemplo de productor-consumidor con productor más rápido que consumidor.
- `prodcons2`: Ejemplo de productor-consumidor con consumidor más rápido que productor.
- `res_prodcons1`: Salida de prodcons1.
- `res_prodcons2`: Salida de prodcons2.

Referencias

[1] Sistemas Operativos. Diseño e Implementación. A. Tanenbaum-A. Woodhull. Segunda Edición (Prentice Hall)

[2] Semáforos y memoria compartida en UNIX. Página de Internet. `http://www.infor.uva.es/~isaac/SO/`

[3] Smx-the Solaris port of MINIX - Paul Ashton, paper disponible en `http://www.infor.uva.es/~benja/smx-html/`

[4] Implementando semáforos. URL: `http://www.cs.lafayette.edu/~pfaffmaj/classes/F04/cs406/`

[5] Operating Systems-EricFreudenthal, cursos, material disponible en: `http://rlab.cs.utep.edu/~freudent/courses/osLectureNotes/`