Sistemas Operativos

Primer Cuatrimestre de 2007

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

Trabajo Práctico (parte I)

Integrante	LU	Correo electrónico
Freijo, Diego	4/05	giga.freijo@gmail.com
Giusto, Maximiliano	486/05	maxi.giusto@gmail.com
Iacobacci, Ignacio	322/02	iiacobac@gmail.com

Palabras Clave

Minix, Kernell, Scheduler, Memoria, Perifericos, Drivers

Índice

1. Soluciones

1.1. Ejercicio 1

1.1.1. Layout Minix

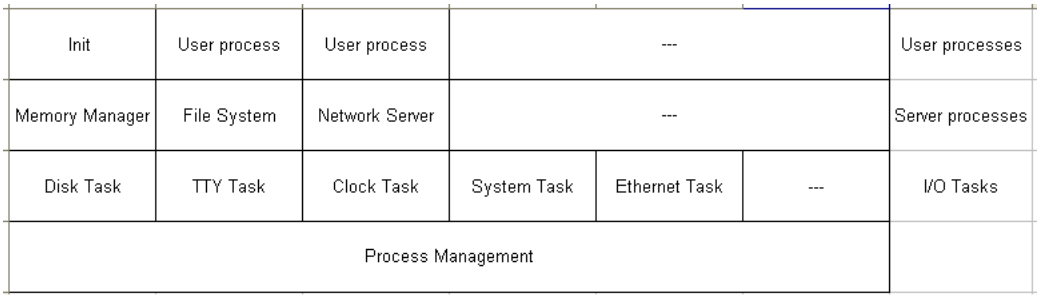

Figura 1: Los niveles de MINIX

1.1.2. Kernel

- SYS_FORK: Informa al kernel que un proceso es forkeado
- SYS_NEWMAP: Permite al Memory Manager setear una porción de memoria para un proceso
- SYS_GETMAP: Permite al Memory Manager tomar una porción de memoria de un proceso
- SYS_EXEC: Setea el contador de programa y el puntero al stack luego de realizar un EXEC
- SYS_XIT: Informa al kernel que un proceso ha terminado
- SYS_GETSP: El llamador pide un el puntero al stack de un proceso
- SYS_TIMES: El llamador pide contar las veces de un proceso
- SYS_ABORT: Si el File System o el Memory Manager no pueden continuar. Aborta Minix
- SYS_FRESH: Empieza con una imagen de proceso nueva durante EXEC
- SYS_SENDSIG: Envia una seal a un proceso
- SYS_SIGRETURN: Sealización estilo POSIX
- SYS_KILL: Mata un proceso, seal enviada via Memory Manager
- SYS_ENDSIG: Finaliza luego de una seal KILL
- SYS_COPY: Pide un bloque de datos para ser copiado entre procesos
- SYS_VCOPY: Pide una serie de bloques de datos para ser copiados entre procesos

- SYS_GBOOT: Copia el parámetro de booteo al un proceso
- SYS_MEM: Retorna el próximo bloque libre de memoria física
- SYS_UMAP: Computa la direccion física de una dirección virtual.
- SYS_TRACE: Retorna una operación de traza

1.1.3. Memory Manager

- FORK: Crea un nuevo proceso
- EXIT: Termina un proceso
- WAIT: Detiene un proceso a la espera de una seal
- WAITPID: Detiene un proceso a la espera de una seal
- BRK: Cambia el tamao del segmento data
- EXEC: Ejecuta un archivo
- KILL: Envia seal a proceso, usualmente para matar al proceso
- ALARM: Programa seal luego de un tiempo especificado
- PAUSE: Detiene hasta seal
- SIGACTION: Administra las seales
- SIGSUSPEND: Suspende hasta seal
- SIGPENDING: Reporta pendientes de seal
- SIGMASK: Manipula la máscara de las seales
- SIGRETURN: Se ejecuta cuando el MM termina manejo de seales. Sirve para restablecer contextos.
- GETUID: Retorna id del usuario
- GETGID: Retorna id del grupo
- GETPID: Retorna id del proceso
- SETUID: Setea id del usuario
- SETGID: Setea id del grupo
- SETSID: Crea una nueva sesión de un proceso
- GETPGRP: Retorna el id del grupo del proceso
- PTRACE: Traza del proceso
- REBOOT: Apaga el sistema o reinicia
- KSIG: Acepta seal que se origina en el Kernel

1.1.4. File System

- ACCESS: Determina accesibilidad hacia un archivo
- CHDIR: Cambia el directorio del trabajo actual
- CHMOD: Cambia el modo de un archivo
- CHOWN: Cambia el dueo de un archivo
- CHROOT: Cambia el directorio del usuario root
- CLOSE: Borra un descriptor
- CREAT: Crea un nuevo archivo
- DUP: Duplica un descriptor
- FCNTL: Ejecuta varias funciones relacionadas con el archivo descriptor
- FSTAT: Retorna estado de un archivo abierto
- IOCTL: Ejecuta varias funciones relacionadas con archivos especiales de tipo carácter (como son las terminales)
- LINK: Crea un link a un archivo
- LSEEK: Mueve puntero de lectura/escritura
- MKDIR: Crea un directorio
- MKNOD: Crea un nuevo inodo
- MOUNT: Monta un file system
- OPEN: Abre un archivo para lectura o escritura o crea un nuevo archivo
- PIPE: Crea un canal de comunicación entre procesos
- READ: Lee entrada
- RENAME: Renombra un archivo
- RMDIR: Remueve un directorio
- STAT: Retorna estado de la ruta de un archivo
- STIME: Setea fecha y hora
- SYNC: Actualiza buffers sucios y super-bloques
- TIME: Obtiene fecha y hora
- TIMES: Retorna información relacionada con los tiempos de un proceso
- UMASK: Setea la máscara de un archivo de un proceso
- UMOUNT: Desmonta un file system
- UNLINK: Remueve una entrada de directorio

- UTIME: Actualiza información relacionada con los tiempos de un proceso
- WRITE: Escribe salida
- UNPAUSE: Envía seal a proceso para ver si está suspendido
- REVIVE: Marca un proceso suspendido como ejecutable

b) Instalamos Minix sobre una imagen .vhd. Ésta fue creada con el entorno Virtua Pc 2004 de Microsoft. La intensión era generar una imagen que pueda ser corrida en dicha aplicación dado que en sucesivos intentos de instalación, el Minix acusaba un "File System Panic" y abortaba la misma. Se usó pues para la instalacion Qemu 4 con el Qemu Manager de Interfaz Gráfica. Sin embargo, esta versión presentó problemas al cargar el último disco de instalación y fue necesario utilizar la versión que presenta la página de la materia para concluir exitosamente la misma. El resto de los puntos fueron desarrollados ntegramente sobre el Qemu 4.

Pásos para la instalación

1. Bajar el Qemu 4 + Qemu Manager, versión para pendrives.

   `http://www.davereyn.co.uk/qem/qemumanager40.zip`

2. Bajar e instalar el Microsoft Virtual PC 2004 (versión gratuita)

3. Microsoft Virtual PC → Archivo → Asistente para disco virtual → Siguiente → Crear un nuevo equipo virtual → Un disco duro virtual → Setear nombre y ubicación → Tamao Fijo → Setear tamao 80MB → Finalizar.

4. Qemu Manager 4 → Crear una nueva máquina virtual → Setear nombre de máquina virtual → Sistema Operativo: `Linux Distribution`, RAM requerida 4MB → Checkear `Usar unidad virtual existente` → Seleccionar la imagen creada en 3 → Salvar maquina virtual.

5. Opcional (crear máquina virtual en Virtual PC) → Asistente para nuevo equipo virtual → Crear un nuevo equipo virtual → Setear nombre y ubicación → Setear Sistema Operativo: `Otro` → Ajustar RAM: 4MB → Utilizar un disco virtual existente → Seleccionar la imagen creada en 3 → Finalizar

6. Usar el archivo 144M.dsk proveido por la cátedra. Configurar máquina virtual (doble click en máquina creada en 4) → Configuración de disco → Disco Floppy A: (fda), selecionar 144M.dsk → Checkear: `Bootear desde disco floppy` → Lanzar maquina virtual.

7. Ingresar `=` en monitor de Minix.

8. Ingresar usuario root.

9. Modificar el archivo setup: `mined /usr/bin/setup`

   Reemplazar líneas:

```
root = ${primary}a
usr  = ${primary}c
```

Por

```
root = hd1
usr  = hd2
```

10. Ejecutar Setup → seleccionar layout → select device, presionar q̈(no modifica nada) → seleccionar hd0 → Seleccionar las particiones de hd1 y hd2 → Enter.

11. Finalizada la instalación, apagar la máquina virtual →

    Deschequear: `Bootear desde disco floppy`

12. con el comando partir, generar las imagenes de `USR.TAZ, SYS.TAZ Y CMD.TAZ`. En línea de comando ejecutar en el directorio donde estan los archivos y el ejecutable: partir [u] usr.taz partir [s] sys.taz partir [c] cmd.taz Ubicar los archivos generados en carpeta media del Qemu

13. Reiniciar máquina virtual.

14. Con `ctrl+alt+2` se abre consola de Qemu. change fda u1.img. Con `ctrl+alt+1` se vuelve a Minix.

15. Ejecutar `setup /usr`. Repetir 14 para cada vez que se pida el siguiente disco.

16. Repertir 14 y 15 para s1.img y con c1.img

    Nota: La instalación de archivo cmd.taz nunca se pudo completar con la versión de Qemu usada. Se intentó varias veces con resultados adversos. Se utilizó el Qemu proveido por la cátedra para la instalación de este archivo.

1.2. Ejercicio 2: Herramientas

Indique que hace el comando make y mknode. Cómo se utilizan éstos 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.

El comando make sirve para compilar archivos. Make es un programa que es normalmente usado para desarrollar programas grandes que consisten en muchos archivos. Lleva cuenta de cuales archivos objeto dependen de cuales archivos fuentes y de cabecera. Cuando es ejecutado, hace la mínima cantidad de recompilaciones para obtener el archivo destino actualizado. El comando make se utiliza en la creación de un nuevo kernel. Luego de modificar algún archivo fuente del kernel, uno debe reconstruir el kernel. En el directorio **/usr/src/tools** está el archivo Makefile, necesario para reconstruir el kernel.

El comando mknod sirve para crear archivos especiales ya sea archivos asociados a dispositivos de entrada/salida o bien directorios. El uso de éste comando está reservado a los superusuarios.

A pesar de que en la instalación de MINIX nosotros no tuvimos que, explícitamente, usar estos comandos, el comando mknod se utilizó para crear los nodos device en el nodo raíz: **/dev/tty**; **/dev/tty[0-2]**; **/dev/hd[0-9]**.

1.3. Ejercicio 3: Comandos básicos MINIX/UNIX

1.3.1. Pongale password root a root.

Utilizando el comando passwd:

```
# passwd
Changing the shadow password of root
New password:
Retype password:
# exit
Minix Release 2.0 Version 0
noname login: root
Password:
#
```

1.3.2. pwd

Indique qu directorio pasa a ser su *current directory* si ejecuta:

1.3.2.1. ♯ cd /usr/src

```
# cd /usr/src
# pwd
/usr/src
```

Utilizando el comando **pwd** podemos ver en que directorio nos encontramos. En éste caso el *current directory* es **/usr/src**.

1.3.2.2. ♯ cd

```
# cd
# pwd
/
#
```

Nuevamente utilizamos el comando **pwd** y el directorio actual es /.

1.3.2.3. Cómo explica el punto 1.3.2.2.?

Ejecutar el comando **cd** sin argumentos cambia el directorio actual al directorio home del usuario. Para el usuario root éste directorio es el directorio raíz del sistema.

1.3.3. cat

Cuál es el contenido del archivo **/usr/src/.profile** y para qué sirve.

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

```
# Login shell profile.

# Environment.
umask 022
PATH=/usr/local/bin:/bin:/usr/bin
PS1="! "
export PATH

# 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
#
```

Cada usuario tendrá éste script en su home, y cada vez que se "loguee"se ejecutará.

1.3.4. find

En qué directorio se encuentra el archivo **proc.c**

```
# find / -name proc.c
/usr/src/kernel/proc.c
```

1.3.5. mkdir

Genere un directorio **/usr/<nombregrupo>**

```
#mkdir /usr/grupo8
# cd /usr
# ls
adm bin grupo8  lib   man  preserve src
ast etc include local mdec spool    tmp
```

1.3.6. cp

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

```
# cp /etc/passwd /usr/grupo8/passwd
```

```
# cd /usr/grupo8
# ls
passwd
#
```

1.3.7. chgrp

Cambie el grupo del archivo **/usr/<grupo>/passwd** para que sea other.

```
# pwd
/usr/grupo8
# ls -l
total 1
-rw-r--r-- 1 root operator 285 Jun 17 22:05 passwd
# chgrp other passwd
# ls -l
total 1
-rw-r--r-- 1 root other 285 Jun 17 22:05 passwd
#
```

1.3.8. chown

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

```
# chown ast passwd
# ls -l
total 1
-rw-r--r-- 1 ast other 285 Jun 17 22:06 passwd
#
```

1.3.9. chmod

Cambie los permisos del archivo **/usr/<grupo>/passwd** para que:

- el propietario tenga permisos de lectura, escritura y ejecución
- el grupo tenga solo permisos de lectura y ejecución
- el resto tenga solo permisos de ejecución

```
# chmod 751 passwd
# ls -l
total 1
-rwxr-x--x 1 ast other 285 Jun 17 22:11 passwd
#
```

1.3.10. grep

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

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

```
# grep POSIX /usr/src/kernel/*
/usr/src/kernel/kernel.h:#define _POSIX_SOURCE 1 /* tell headers to inclu
de POSIX stuff */
/usr/src/kernel/rs232.c: if ((tp->tty_termios.c_lflag & IXON) && rs->oxoff !=
_POSIX_VDISABLE)
/usr/src/kernel/system.c: * SYS_SENDSIG send a signal to a process (POSIX style)
/usr/src/kernel/system.c: * SYS_SIGRETURN complete POSIX-style signalling
/usr/src/kernel/system.c:/* Handle sys_sendsig, POSIX-style signal */
/usr/src/kernel/system.c:/* POSIX style signals require sys_sigreturn to put
things in order before the
/usr/src/kernel/tty.c:/* These Posix functions are allowed to fail if _POSIX_JOB
_CONTROL is
/usr/src/kernel/tty.c: /* _POSIX_VDISABLE is a normal character value, so better
escape it. */
/usr/src/kernel/tty.c: if (ch == _POSIX_VDISABLE) ch |= IN_ESC;
#
```

1.3.11. su

1.3.11.1. Para qué sirve?

Permite convertir un usuario en otro sin tener que desconectarse del sistema. Por defecto lo convierte en superuser. Le pedirá la password correspondiente.

1.3.11.2. Qué sucede si ejecuta el comando su estando logueado como root?

Se genera un shell nuevo. Como estabamos logueados como root, no pidió password.

1.3.11.3. Genere una cuenta de <usuario>

```
# adduser pepe other /usr/pepe
cpdir /usr/ast /usr/pepe
```

```
chown -R 10:3 /usr/pepe
echo pepe::0:0::: >>/etc/shadow
echo pepe:##pepe:10:3:pepe:/usr/pepe: >>/etc/passwd
The new user pepe has been added to the system. Note that the password,
full name, and shell may be changed with the commands passwd(1), chfn(1),
and chsh(1). The password is now empty, so only console logins are possible.
```

1.3.11.4. Entre a la cuenta <usuario> generada

```
Minix Release 2.0 Version 0
noname login: pepe
$
```

No pidió Password pues el usuario pepe no tiene aún una seteada.

1.3.11.5. Repita los comandos de 1.3.11.2

Estando logueado como pepe ejecutamos el comando **su**.

```
$ su
Password:
#
```

Nos pide la password de root pues el usuario pepe es parte del grupo other. Si fuera parte del grupo operator no la pediría.

1.3.12. passwd

1.3.12.1. Cambie la password del usuario nobody

```
# passwd nobody
Changing the password of nobody
New password:
Retype password:
#
```

1.3.12.2. Presione las teclas ALT-F2 y verá otra sesión MINIX. Loguearse como nobody

```
Minix Release 2.0 Version 0
noname login: nobody
Password:
$
```

Nos logueamos con password nobody.

1.3.12.3. Ejecutar el comando su

```
$ su
Password:
#
```

1.3.12.3.1. Qué le solicita?

Solicita la password de root.

1.3.12.3.2. Sucede lo mismo que en 1.3.11.2?Por qué?

No, cuando se realizó lo mismo con el usuario **root** no fue necesario ingresar la password. Esto se debe a que el usuario **nobody** pertenece al grupo **nogroup**, quienes no tienen permisos para loguearse como **root** y, de hecho, casi no poseen ningun tipo de privilegios.

1.3.13. rm

Suprima el archivo **/usr/<grupo>/passwd**.

```
# ls /usr/grupo8
passwd
# rm /usr/grupo8/passwd
# ls /usr/grupo8
#
```

1.3.14. ln

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

```
# ls -l /etc/passwd
-rw-r--r-- 1 root operator 314 Apr 15 2007 /etc/passwd
# ln /etc/passwd /tmp/contra1
# ln /etc/passwd /tmp/contra2
# ls -l /etc/passwd
-rw-r--r-- 3 root operator 314 Apr 15 2007 /etc/passwd
#
```

En el listado del archivo, la segunda columna (luego de los permisos) indica la cantidad de links que tiene ese archivo.

1.3.15. mkfs

Genere un Filesystem MINIX en un diskette

```
# mkfs /dev/fd0
```

1.3.16. mount

Móntelo en el directorio /mnt.

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

Presente los filesystems que tiene montados.

```
# mount
/dev/hd1 is root device
/dev/hd2 is mounted on /usr
/dev/fd0 is mounted on /mnt
#
```

1.3.17. df

Qué espacio libre y ocupado tienen todos los filesystems montados?(en KBYTES)

```
# df
Device    Inodes Inodes Inodes  Blocks Blocks Blocks  Mounted   V Pr
           total  used   free    total  used   free      on
           -----  -----  -----   -----  -----  -----    ------   - --
/dev/hd1    496    212    284     1480    700    780    /        2 rw
/dev/hd2  12528   3309   9219    75096  27443   47653   /usr     2 rw
/dev/fd0    480      1    479     1440     35    1405   /mnt     2 rw
#
```

En todos los dispositivos, el tamaño de bloque es 1KB, por lo tanto el espacio disponible en KB es el indicado por la columna **Blocks free** y el espacio ocupado es el indicado por la columna **Blocks used**.

1.3.18. ps

1.3.18.1. Cuntos procesos de usuario tiene ejecutando?

```
# ps -a
PID TTY  TIME CMD
 41  co  0:00 -sh
 42  c1  0:00 getty
 47  c0  0:00 ash
 48  c0  0:00 ps -a
```

Los procesos de usuario que se están ejecutando son 4 (incluyendo al **ps**).

1.3.18.2. Indique cuántos son del sistema

```
# ps -ax
PID TTY  TIME CMD
  0   ?  0:00 TTY
  0   ?  0:00 SCSI
  0   ?  0:00 WINCH
  0   ?  0:00 SYN_AL
  0   ?  8:35 IDLE
  0   ?  0:00 PRINTER
  0   ?  0:00 FLOPPY
  0   ?  0:00 MEMORY
  0   ?  0:00 CLOCK
  0   ?  0:01 SYS
  0   ?  0:06 HARDWAR
  0   ?  0:00 MM
  0   ?  0:04 FS
  1   ?  0:00 INIT
 40  co  0:00 -sh
 27   ?  0:00 update
 41  c1  0:00 getty
 47  co  0:00 ash
 57  co  0:00 ps -ax
#
```

Aquí se listan todos los procesos incluyendo también a los de usuario. En total hay 18 procesos ejecutándose, 4 son de usuario y 15 del sistema.

1.3.19. umount

1.3.19.1. Desmonte el Filesystem del directorio /mnt

```
# umount /dev/fd0
/dev/fd0 unmounted from /mnt
#
```

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

```
# mount /dev/fd0 /mnt -r
/dev/fd0 is read-only mounted on /mnt
#
```

1.3.19.3. Desmonte el Filesystem del directorio /mnt

```
# umount /dev/fd0
/dev/fd0 unmounted from /mnt
#
```

1.3.20. fsck

Chequee la consistencia de Filesystem del diskette

```
# fsck /dev/fd0

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
#
```

1.3.21. dosdir

Tome un diskette formateado en DOS con archivos y ejecute *dosdira*

```
#dosdir a
dosdir: cannot open /dev/dosA: no such file or directory
#
```

Ejecute los comandos necesarios para que funcione correctamente el comando anterior

El diskette con formato DOS está en **/dev/fd1**

```
#dosdir fd1
HOLADOS.TXT
ABMUSR
GETNEXTG
MACHAQUE
#
```

1.3.22. dosread

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

```
#dosread fd1 HOLADOS.TXT
```

```
hola mundo!

#dosread -a fd1 HOLADOS.TXT > /tmp/holaminix.txt
#cat /tmp/holaminix.txt
hola mundo!

#
```

1.3.23. doswrite

Copie el archivo /etc/passwd al diskette DOS

```
#doswrite -a fd1 passwd < /etc/passwd
#dosdir fd1
HOLADOS.TXT
ABMUSR
GETNEXTG
MACHAQUE
PASSWD
#
```

1.4. Ejercicio 4: Uso de STDIN, STDOUT, STDERR y PIPES

1.4.1. STDOUT

1.4.1.1. Conserve 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

```
# cd /usr/grupo8
# ls -al /usr/src > /usr/grupo8/fuentes.txt
#
```

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

```
# wc /usr/grupo8/fuentes.txt
     25    218   1455 /usr/grupo8/fuentes.txt
#
```

El archivo **/usr/grupo8/fuentes.txt** tiene 25 lneas, 218 palabras y 1455 caracteres.

1.4.2. STDOUT

1.4.2.1. Agregue el contenido, ordenado alfabéticamente, del archivo /etc/passwd al final del archivo /usr/<grupo>/fuentes.txt

```
# sort /etc/passwd >> /usr/grupo8/fuentes.txt
```

1.4.2.2. Presente cuántas lineas, palabras y caracteres tiene usr/<grupo>/fuentes.txt

```
# wc /usr/grupo8/fuentes.txt
     34    234   1769 /usr/grupo8/fuentes.txt
#
```

El archivo **/usr/grupo8/fuentes.txt** tiene 34 lneas, 234 palabras y 1769 caracteres.

1.4.3. STDIN

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

```
# echo 2355 > /usr/grupo8/hora.txt
```

1.4.3.2. Cambie la hora del sistema usando el archivo /usr/<grupo>/hora.txt generado en 1.4.3.1

```
# date -q < /usr/grupo8/hora.txt
```

1.4.3.3. Presente la fecha del sistema

```
# date
Mon Aug 20 23:55:07 MET DST 2007
#
```

1.4.4. STDERR

Guarde el resultado de ejecutar el comando dosdir k en el archivo /usr/<grupo>/error.txt. Muestre el contenido de /usr/¡grupo¿/error.txt.

```
# dosdir k > /usr/grupo8/error.txt 2>&1
# cat /usr/grupo8/error.txt
dosdir: cannot open /dev/dosK: No such file or directory
#
```

1.4.5. PIPES

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

1.4.5.1. 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 informela 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 sesión, haga un kill sobre los procesos update y getty.

```
# pwd
/
# ls -lc /usr/bin/s* | grep sync | wc
      2     18    140
#
```

2 lneas, 18 palabras y 140 caracteres.

1.5. Ejercicio 5

El comando que realiza las tareas es **abmusr** y su uso es

Para agregar al usuario arturo: `abmusr -a arturo grupoarturo /usr/arturo`

Para borrar al usuario arturo: `abmusr -b arturo`

1.6. Ejercicio 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.

Creamos el programa y lo guardamos en el directorio **/usr/src**. Para compilar el programa realizamos los siguientes pasos:

```
# pwd
/usr/src
# cc loop.c -oloop
#
```

(a) Correrlo en foreground Qué sucede? Mate el proceso con el comando kill.

Ejecutamos el programa en foreground y notamos que la pantalla se llena de números, es decir, se produce una impresión indefinida de números en pantalla. La consola queda inutilizable ya que el programa está en un loop infinito.

```
# ./loop
25354554849505152535455484950515253545548495051525354554849505152535455484950515253545
25354554849505152535455484950515253545548495051525354554849505152535455484950515253545
25354554849505152535455484950515253545548495051525354554849505152535455484950515253545
25354554849505152535455484950515253545548495051525354554849505152535455484950515253545
25354554849505152535455484950515253545548495051525354554849505152535455484950515253545
25354554849505152535455484950515253545548495051525354554849505152535455484950515253545
```

Como necesitamos saber el PID del proceso para usar kill, creamos una nueva consola mediante Alt+F2 y en ella ejecutamos los siguientes comandos:

```
Minix Release 2.0 Version 0
noname login: root
Password:
# ps
PID TTY  TIME CMD
 40  co  0:00 -sh
 41  c1  0:00 -sh
 47  co  0:00 ash
 49  co  0:46 ./loop
 53  c1  0:00 ash
 54  c1  0:00 ps
# kill 49
#
```

Esto causa que en la primer consola el programa finalice con la leyenda **Terminated**.

```
24849505152484950515248495051524849505152484950515248495051524849505152484950515
24849505152484950515248495051524849505152484950515248495051524849505152484950515
24849505152484950515248495051524849505152484950515248495051524849505152484950515
24849505152484950515248495051524849505152484950515248495051524849505152484950515
24849505152484950515248Terminated
```

(b) Ahora ejecútelo en background

/usr/src/loop > /dev/null &

Qué se muestra en la pantalla?

Al ejecutarlo en background, el shell devuelve el ID del proceso. Al estar siendo redirigida la salida a null no imprime nada en pantalla.

```
# pwd
/usr/src
# ./loop > /dev/null &
#
```

Cabe aclarar que si el usuario logueado cambió su shell, por ejemplo cambió a **ash**, no visualizará el ID del proceso y deberá realizar un **ps -a** para obtenerlo.

Qué sucede si presiona la tecla F1? Qu significan esos datos?

La tecla F1, muestra una tabla con los procesos del sistema. Ésta tabla contiene datos del proceso y datos del sistema. Presenta las siguientes columnas:

- **pid**: El identificador del proceso. Puede ser el PID asignado por el MM, o puede ser el spot en la tabla de procesos, es decir el p nr, si este es menor a 0.
- **pc**: Indica el valor del Program Counter en el momento en que el proceso fue bloqueado.
- **sp**: Indica la dirección del puntero al tope del stack del proceso.
- **flag**: Indica el valor del word de los flags. Si no hay ningún flag activado, es decir, si este valor es 0, el proceso puede ser ejecutado, en caso contrario se encuentra bloqueado o aún no fue inicializado.
- **user**: Tiempo que el proceso estuvo utilizando el procesador. Está medido en "ticks", aproximadamente hay 60 ticks por segundo (59,7 medido, pero bajo Bochs por lo que es inexacto).
- **sys**: Tiempo de ejecución de rutinas de sistema relacionadas a la administración de éste proceso. También medido en "ticks"
- **text**: Dirección física donde comienza el segmento TEXT, o de código, asignado al proceso.
- **data**: Dirección física donde comienza el segmento DATA, o de datos, asignado al proceso.
- **size**: Tamaño del espacio ocupado en memoria por el proceso en KB.
- **recv**: Si el proceso se encuentra bloqueado en espera de el envio o recepción de un mensaje, este campo indica quien es el receptor o emisor de dicho mensaje. Caso contrario, está en blanco.
- **command**: Indica el nombre del proceso.

Qué sucede si presiona la tecla F2? Qué significan esos datos?

Al presionar F2 se obtiene información sobre el mapa de memoria de los procesos (estructura mem map en **/usr/include/minix/type.h**). La tabla presenta las siguientes columnas:

- **PROC**: Indica el slot en la tabla de procesos, es decir el p nr.

- **NAME**: Indica el nombre del proceso.
- **TEXT**: Para el segmento de código indica la dirección virtual, la dirección física y el tamaño, en ese orden.
- **DATA**: Para el segmento de datos indica la dirección virtual, la dirección física y el tamaño, en ese orden.
- **STACK**: Para el segmento de stack, o pila, indica la dirección virtual, la dirección física y el tamaño, en ese orden.
- **SIZE**: Tamaño del espacio ocupado en memoria por el proceso.

1.7. Ejercicio 7

1.7.1. Inciso a

Ejecutar

```
# make install
```

Se compilan todo los archivos del kernel, mm o fs que no tengan su archivo .o generado.

```
cd ../kernel && exec make - install
cd keymaps && make - install
make: 'install' is up to date
cd ../mm && exec make - install
make: 'install' is up to date
cd ../fs && exec make - install
make: 'install' is up to date
cd ../inet && exec make - install
make: 'install' is up to date
```

Ejecutar

```
# make hdboot
```

Para generar una nueva imagen de Minix

```
cd ../kernel && exec make -
make: 'kernel' is up to date
cd ../mm && exec make -
exec cc -c -I/usr/include main.c
exec cc -c -I/usr/include forkexit.c
exec cc -c -I/usr/include break.c
exec cc -c -I/usr/include exec.c
exec cc -c -I/usr/include signal.c
exec cc -c -I/usr/include alloc.c
exec cc -c -I/usr/include utility.c
exec cc -c -I/usr/include table.c
```

```
exec cc -c -I/usr/include putk.c
exec cc -c -I/usr/include trace.c
exec cc -c -I/usr/include getset.c
exec cc -c -I/usr/include newcall.c
exec cc -c -I/usr/include llamsistema.c
exec cc -c -I/usr/include semaf.c
exec cc -o mm -i main.o forkexit.o break.o exec.o \
signal.o alloc.o utility.o table.o \
putk.o trace.o getset.o newcall.o \
llamsistema.o semaf.o
install -S 256w mm
cd ../fs && exec make -
exec cc -c -I/usr/include main.c
exec cc -c -I/usr/include open.c
exec cc -c -I/usr/include read.c
exec cc -c -I/usr/include write.c
exec cc -c -I/usr/include pipe.c
exec cc -c -I/usr/include device.c
exec cc -c -I/usr/include path.c
exec cc -c -I/usr/include mount.c
exec cc -c -I/usr/include link.c
exec cc -c -I/usr/include super.c
exec cc -c -I/usr/include inode.c
exec cc -c -I/usr/include cache.c
exec cc -c -I/usr/include cache2.c
exec cc -c -I/usr/include filedes.c
exec cc -c -I/usr/include stadir.c
exec cc -c -I/usr/include protect.c
exec cc -c -I/usr/include time.c
exec cc -c -I/usr/include misc.c
exec cc -c -I/usr/include utility.c
exec cc -c -I/usr/include table.c
exec cc -c -I/usr/include putk.c
exec cc -c -I/usr/include othercall.c
exec cc -o fs -i main.o open.o read.o write.o pipe.o \
device.o path.o mount.o link.o super.o inode.o \
cache.o cache2.o filedes.o stadir.o protect.o time.o \
lock.c misc.o utility.o table.o putk.o othercall.o
install -S 512w fs
installboot -image image ../kernel/kernel ../mm/mm ../fs/fs  init
    text    data     bss     size
   53248    8964   40076   102288  ../kernel/kernel
   14096    1244   30736    46076  ../mm/mm
   29184    2228  108084   139496  ../fs/fs
    6828    2032    1356    10216  init
  ------  ------  ------  -------
  103356   14468  180252   298076  total
exec sh mkboot hdboot
rm /dev/hd1:/minix/2.0.2SISTEMASO
```

```
cp image /dev/hd1:/minix/2.0.2SISTEMASOPERATIVOS-UBA-FCENr1
Done.
```

Ejecutar

```
# reboot
```

para reiniciar con la nueva version de kernel

1.7.2. Inciso a

Se crea directorio `/minix2` y se copia desde `/minix` el archivo 2.0.0. Al iniciar minix presionar ESC. Escribir

```
> image=minix
```

Si se quiere iniciar con la nueva versin del kernel

Escribir

```
> image= minix2
```

Si se quiere iniciar con la versin original.

```
> boot
```

Para iniciar

1.8. Ejercicio 8

En MINIX tenemos las siguientes características:

- Administración de la memoria: se maneja por política de memoria segmentada variable con primera zona. Es decir, se asigna el primer lugar donde exista el espacio necesario.
- Administración del procesador: la administración de MINIX es una triple cola de prioridad (es decir, es una multicola) donde se utiliza FIFO para la del Kernell (de mayor prioridad), FIFO para la de System Tasks como el MM y FS (de prioridad menor) y Round Robin para las colas del usuario (prioridad última).
- Administración de E/S: en la 1er capa (la de drivers) hay un proceso (driver) por cada dispositivo en el sistema. Cuando un proceso de usuario quiere acceder a un dispositivo, éste lo realiza a travez de la 2da capa por alguno de los servicios alli expuestos (MM, FS o red) y éstos se comunican con los drivers a travez de mensajes (los cuales se comunican con el kernell en la capa 0 por otros mensajes).
- Administración de FS: el sistema de archivos de MINIX posee seis componentes:
    - El bloque de booteo, que esta siempre almacenado en el primer bloque. Contiene la información sobre como iniciar el sistema al encenderse.

- El segundo bloque es el Superbloque y almacena informacion sobre el FS que permite al SO localizar y entender otras estructuras de sistemas de archivos (número de inodos y zonas, el tamaño de dos bitmaps y el bloque de inicio del área de información).
- El bitmap de inodo es un simple mapa de inodos que localiza cuales están en uno y cuales están libres representándolos con un bit.
- El bitmap de zona trabaja de la misma forma que el bitmap de inodo, excepto que localiza las zonas.
- Los inodos de área. Cada archivo o directorio está representado como un inodo, el cual almacena metadata incluyendo tipo (archivo, directorio, bloque, char, pipe), ids de usuario y grupo, tres timestamps que graban la fecha y hora de último acceso, última modificación y último cambio de estado. Un inodo además contiene una lista de direcciones que apuntan a las zonas en el área de información donde el archivo o directorio es'ta ubicado.
- El área de datos es el componente mas grande del sistema de archivos, usando la mayoría del espacio.

Y en LINUX tenemos lo siguiente:

- Administración de la memoria: utiliza paginación por demanda, por lo que brinda:
  - Grandes espacios de direccionamiento El sistema operativo hace aparentar al sistema como si tuviese mas memoria que la que realmente tiene. La memoria virtual puede ser varias veces mas grandes que la memoria fisica en el sistema.
  - Protección Cada proceso en el sistema tiene su propio espacio de direccionamiento. Éstos están completamente separados entre sí y por eso un proceso corriendo una aplicacion no puede afectar otro. Además, los mecanismos de hardware de memoria virtual permite que áreas de memoria sean protegidas contra escritura para proteger el código y los datos de ser sobre escritos por otras aplicaciones.
  - Mapeo de memoria El mapeo de memoria es usado para mapear archivos al espacio de direccionamiento de un proceso. Es decir, los contenidos de un archivo son enlazados directamente en el espacio de direccionamiento virtual del proceso.
  - Asignación justa de la memoria física El subsistema de administración de memoria le permite obtener a cada proceso en ejecución en el sistema una parte justa de la memoria física.
  - Memoria virtual compartida Although virtual memory allows processes to have separate (virtual) address spaces, there are times when you need processes to share memory. For example there could be several processes in the system running the bash command shell. Rather than have several copies of bash, one in each processes virtual address space, it is better to have only one copy in physical memory and all of the processes running bash share it. Dynamic libraries are another common example of executing code shared between several processes. Shared memory can also

be used as an Inter Process Communication (IPC) mechanism, with two or more processes exchanging information via memory common to all of them. Linux supports the Unix TM System V shared memory IPC.

A pesar que memoria virtual permite a los procesos tener separados espacios (virtuales) de direccionamiento, hay veces que necesitan compartir memoria (por ejemplo, librerías dinámicas). Por lo que, en lugar de duplicar datos, se utiliza un mecanismo de comunicación entre procesos (IPC) para que dos o más procesos intercambien información a travéz de memoria en común. El soporte en Linux se llama TM System V shared memory IPC.

- Administración del procesador: se basa en una política Round-Robin (en su nomenclatura, *time-sharing technique*) donde para cada proceso existe una cuota de tiempo (quantum, o también llamado *timeslice*) que puede utilizar del procesador por época. Al comienzo de cada época, el timeslice es recomputado por proceso y cada uno de ellos podrá utilizar el procesador éste tiempo asignado a cada uno durante la época. Notar que si un proceso utiliza demasiadas E/S y por ende no consume todo su timeslice al final de la época, éste sobrante es agregado en la próxima época.

- Administración de E/S: Linux soporta tres tipos de dispositivo de hardware: caracter, bloque y red. Los dispositivos en modo caracter se leen y se escriben directamente sin buffering, por ejemplo los puertos seriales /dev/cua0 y /dev/cua1 del sistema. Los dispositivos en modo bloque pueden ser solamente escritos leídos en en los múltiplos del tamaño de bloque, de típicamente 512 o 1024 bytes. Los dispositivos en modo bloque son accedidos por el buffer y sede forma aleatoria, es decir, cualquier bloque puede ser leído o ser escrito no importa dónde está en el dispositivo. Los dispositivos en modo bloque se pueden acceder vía un archivo especial del dispositivo pero se suele utilizar el sistema de archivos. Solamente un dispositivo en modo bloque puede soportar un sistema de archivos montado. Los dispositivos de la red son accedidos a travéz de los subsistemas de red.

  Hya muchos y diferentes drivers de dispositivos en el kernell de Linux, lo cual es una de sus fuerzas, pero todos comparten algos atributos en común:

  - Código de kernell Los drivers son parte del kernell y, como todo otro código en el kernell, si funciona mal puede daar el sistema incluso perdiendo información del FS.
  - Interfaces del kernell Los drivers deben prooveer una interfaz estándar al kernell de Linuz o al subsistema del que son parte.
  - Mecanismos del kernell y servicios Los drivers usan los servicios estándar del kernell como administración de memoria, manejo de interrupciones y colas de espera para operar.
  - Cargable Muchos drivers pueden ser cargados a demanda como modulos del kernell cuando son necesitados y descargados cuando no son más usados. Ésto hace al kernell flexible y eficiente con los recursos.
  - Configurable Los drivers pueden ser construidos en el kernell, y se puede configurar cuales cuando se compila el kernell.

    - Dinámico Mientras es sistema bootea y cada driver es inicializado, éste mira por cada dispositivo que está controlando. No importa si el dispositivo que se está controlando por un dispositivo en particular no existe. En éste caso el driver es simplemente redundante y no causa ningún daño aparte de ocupar un poquito de memoria del sistema.

- Administración de FS: por el momento, Linux soporta 15 sistemas de archivo: ext, ext2, xia, minix, umsdos, msdos, vfat, proc, smb, ncp, iso9660, sysv, hpfs, affs y ufs. Aunque el principal, en principio (1992), era ext éste carecia de buen rendimiento. Por eso es que fue reemplazado por ext2 en 1993. Éste sistema, como muchos otros, es construido en la premisa que la información en los archivos es conservada en bloques. Éstos bloques son todos de la misma longitud y, a pesar que la longitud puede variar entre diferentes sistemas EXT2, el tamaño del bloque de un sistema particular es establecido cuando es creado. Cada tamaño de archivo es redondeado hacia arriba hasta un número entero de bloques. Si el tamaño de bloque es de 1024 bytes entonces un archivo de 1025 bytes ocupará 2 bloques. Desafortunadamente esto significa que en promedio se desperdicia la mitad de un bloque por archivo. Pero para favorecer el desempeño del CPU éste ineficiente *tradeoff* es utilizado. No todos los bloques en el FS contienen información, algunos deben ser usados para contener la información que describe la estructura del sistema de archivos. EXT2 define la topología del sistema describiendo cada archivo en el sistema con una estrucuta de datos inodo. Un inodo describe que bloques de información dentro de un archivo ocupa al igual que los derechos de acceso del archivo, las veces que se modificó y el tipo de archivo. Cada archivo en el FS EXT2 es descripto por un único inodo y cada inodo tiene un único número identificándolo. Los inodos para el sistema de archivos estan contenidos en tablas de inodos. Los directorios EXT2 son simples archivos especiales (descriptos como inodos) que contienen punteros a los inodos de sus entradas dentro del directorio.

Notar que ambos sistemas operativos poseen ciertas pequeñas similitudes como los inodos en el sistema de archivos y round robin para los procesos de usuario, pero varias características transforman a Linux en un sistema de producción a diferencia de Minix que se límita bastante. Por ejemplo, el tamaño máximo de los archivos en Minix (64MB) hace que no pueda ejecutar aplicaciones tales como una base de datos decente de hoy en día, y ni hablar de archivos multimedia grandes como películas. Además, la administración de memoria de Minix es muy rudimentaria, sin prácticamente protección asegurada y con un espacio de direcciones limitado a la memoria física, a diferencia de Linux que posee memoria virtual a travéz de memoria paginada con todas las ventajas que se describieron como la proteccion y el espacio mayor de dirección. Es por eso que no sorprende la cantidad de actualizaciones que recibió Linux (al igual que su popoularidad) y el uso didáctico que se le dá a Minix (debido a la simpleza de sus administraciones).

1.9. Ejercicio 9

Los dispositivos de E/S pueden dividirse en 2 categorias: dispositivos tipo bloque y dispositivos tipo caracter. Uno tipo bloque es el cual guarda infomación en bloques de tamaño fijo, cada uno con su propia dirección. Los bloques comunes van del rango

de los 128 bytes a los 1024 bytes. La propiedad esencial de este tipo de dispositivo es hacer posible la lectura o escritura de cada bloque independientemente de los otros. En otras palabas, instantaneamente el programa puede leer o escribir cualquiera de sus bloques. Un ejemplo de ellos son los discos.

Los dispositivos de E/S tipo caracter entrega o acepta una cadena de caracteres, sin respetar ninguna estructura de bloques. No es direccionable y no tiene ningún tipo de operación de búsqueda. La terminal, impresoras, cintas de paepl, tarjetas perfordas, interfaces de red, mouses y muchos otros dipositivos no que no son como discos pueden verse como dispositivos tipo caracter. Este modelo es en general suficiente para ser usado para hacer que el software del S.O. pueda trabajar con dispositivos de E/S independientes. El file system, por ejemplo, trabaja solo con dispositivos de bloques abstractos y deja la parte dependiente del dispositivo a software de bajo nivel llamado drivers de dispositivos.

El trabajo de los drivers de dispositivos es aceptar pedidos abstractos de un software independiente del dispositivo sobre él, y ver que el pedido sea realizado.

En MINIX, para cada clase de dispositivo de E/S, esta presente una tarea de E/S (I/O task) o driver de dispositivo. Estos drivers son procesos terminados, cada uno con su propio estado, mapa de memoria y más. Estos driverse se comunican entre si, si es necesario, con el file system usando el mecanismo de pasaje de mensajes estandar usado por todos los procesos de MINIX. Además, cada driver de dispositivo esta escrito en un solo archivo fuente, como floppy.c o clock.c. La única diferencia entre los drivers y otros procesos es que los primeros estas linkeados con el kernel, y comparten todo el espacio de de direccionamiento.

El programa principal para cada driver de dispositivo tipo bloque es estructuralmente el mismo. Posee un ciclo infinito, el cual espera la llegada de un mensaje invocandolo. Si esto sucede, se llamar a la operacin del tipo entregado por el mensaje.

La estructura es la siguiente:

```
message mess;                          /* buffer del mensaje */

void io_task() {
    initialize();                      /* solo se inicializa una vez */
    while(TRUE){
        receive(ANY, &mess);           /* espera a un pedido de trabajo */
        caller = mess.source;          /* proceso desde donde vino el mensaje */
        switch(mess.type){
            case READ: rcode = dev_read(&mess);break;
            case WRITE: rcode = dev_write(&mess);break;
            /* Other cases go here, e.g., OPEN, CLOSE, IOTCTL */
            default: rcode = ERROR;
        }
        mess.type = TASK_REPLY;
        mess.status = rcode;           /* respuesta */
        send(caller, &mess);           /* Se envia respuesta al proceso llamador */
    }
```

```
}
```

Cuando el sistema inicia, cada uno de los drivers se inicializa para definir datos internos como tablas y similares cosas. Cada task intenta recibir un mensaje. Cuando alguno llega, se graba la identidad del llamador, y se ejecuta un procedimiento en particular para realizar el trabajo. Luego de finalizado, una respuesta es enviada al llamador, y el task vuelve a el tope del ciclo esperando un nuevo pedido.

Para generar un driver de dispositivo tipo bloque hay que hacer lo siguiente:

1. hay que alterar el /usr/include/minix/com.h. Este archivo es el que guarda el id de cada uno de los I/O task. Este id es llamado "major number"del dispositivo. Este número especifica la clase de dispositivo, como los floppy, disco rígido, o terminal. Todos los dispositivos con el mismo "major numberçomparten el mismo codigo de driver dentro del S.O.

2. luego hay que alterar /usr/include/minix/config.h para determinar cuales son los dispositivos con los cuales el kernel va a ser compilado. Incluir todos estaria mal. Esto es para que sea más facil excluir el nuevo dispositivo.

3. alterar el /usr/include/minix/const.h Se encarga de las constantes usadas por el MINIX. Se la usa para agregar en nuevo proceso a las NR_TASKS. Setea el numero de las tareas predefinidas del kernel incluyendo los drivers de dispositivo.

4. /usr/src/kernel/proto.h Hay que incluir el header o prototipo de la llamada al driver. Eso es, predefinir el punto de entrada del nuevo driver. Este punto es la primer función que se ejecuta en el driver, como el main() es el punto de entrada de un archivo de C. En el ejemplo, el task va a manejar un disco, se llamara en tal caso, `disco_task`.

   ```
   PUBLIC _PROTOTYPE( void disco_task, (void) );
   ```

5. /usr/src/kernel/table.c En este archivo debe existir una entrada para las tasks. En ella se especifica el tamaño del stack que va a usar el nuevo driver.

   ```
   #define DISCO_STACK (4 * SMALL_STACK * ENABLE_DISCO)
   ```

   Además se debe agregar dentro del struct tasktab una línea donde se relacionan el task(el nombred del driver), la pila y el programa del driver propiamente dicho en ese orden.

   ```
   #if ENABLE_DISCO

   { disco_task, DISCO_STACK , "DISCO" },

   #endif
   ```

   Donde *DISCO* es el nombre del proceso.

6. `/usr/src/kernel/table.c` También hay que agregar a este archivo una entrada para el nombre de la task en particular. Hay que agregar un archivo especial, en nuestro caso "DISCO.[en] /dev y ligar el proceso del nuevo driver con él. La nueva línea se agrega en el struct dmap. (como muesta se agrega el encabezado que aparece en el fuente).

```
?    Open        Read/Write  Close       Task #      Device  File
-    ----        ----------  -----       -------     ------  ----

DT(1, dev_opcl,   call_task,  dev_opcl,   DISCO)      /* 7 = /dev/disco */
```

7. Hay que generar el driver mismo. Depende de que es lo que maneje, la estructura del mismo. Para el ejemplo citado, el archivo se llamar disco.c y se encontrar en la ruta `/usr/src/kernel/disco.c`

8. Es necesario alterar el Makefile del kernel para que el nuevo driver sea compilado con él.

9. Hay que generar el iNodo:

   `mknod /dev/disco Flags MayorNumber MinorNumber\verb`

1.10. Ejercicio 10: Modificación de códigos

a. Modifique el "scheduler" original del MINIX para el nivel de usuarios.

b. Modifique la administración de memoria original del MINIX

En ambos casos deberá describir en el informe cuáles fueron las decisiones tomadas, cuáles fueron las espectativas y cuáles fueron los resultados obtenidos e informar el juego de programas utilizados con los cuales se llegó a alguna conclusión. (test o pruebas mencionados en Forma de entrega).

1.10.1. Modique el "scheduler" original del MINIX para el nivel de usuarios

Decisiones tomadas

La administración de procesos de Minix divide a los procesos en tres categorías: procesos de usuario, procesos SERVER (como ser MM, o FS) y procesos TASK (kernel). Asimismo, para los procesos de usuario, utiliza una administración *Round−Robin*, y un proceso sólo podrá perder el recurso procesador por alguna de las tres siguientes razones:

- El proceso finaliza.
- Se bloquee por una operación de E/S o sincronización.
- El uso exceda un cierto *Quantum*.

Decidimos modificar la administración de los procesos de usuario para que cambie de $Round - Robin$ a $FIFO$,

Expectativas

Como estamos permitiendo un uso más prolongado y continuo del recurso procesador, es de esperar que los procesos de alto uso de CPU se vean beneficiados, quizás en detrimento de procesos que tengan, en cambio, mayor uso de E/S, ya que éstos se bloquearán rápidamente y estarán un largo tiempo en espera del recurso.

Resultados

Nuestra propuesta es modificar el archivo **/usr/src/kernel/clock.c**. De la siguiente manera: si esta definido UBA_FCEN, entonces en la función *do_clocktick* no se ejecutará el código que realiza la selección de un proceso de usuario nuevo, en el caso en que no haya ni procesos SERVER ni TASK para elegir y además que el proceso anterior haya excedido su *Quantum*.

El código que no se ejecuta si UBA_FCEN está definido está en el Apéndice A.

Se generaron dos procesos de prueba **highCPU.c** y **lowCPU.c**. El primero consiste en la repetición de un ciclo que contiene a su vez otro ciclo, en total clica aproximadamente 5000050000 veces. El segundo consiste en la repetición de un ciclo 10000 veces. Ambos al terminar muestran por pantalla la etiqueta: "Final de highCPU" y "Final de lowCPU" respectivamente.

A continuacin se muestran los resultados obtenidos con la administración *Round−Robin*:

```
# cd /usr/grupo8/ej10
# ls
highCPU highCPU.c lowCPU lowCPU.c
#./highCPU & ./lowCPU
Final de lowCPU
# Final de highCPU
#
#./lowCPU & ./highCPU
Final de lowCPU
# Final de highCPU
```

Si cambiamos a la administración a $FIFO$ obtenemos los siguientes valores:

```
# cd /usr/grupo8/ej10
# ls
highCPU highCPU.c lowCPU lowCPU.c
#./highCPU & ./lowCPU
Final de lowCPU
# Final de highCPU
#
#./lowCPU & ./highCPU
Final de highCPU
# Final de lowCPU
```

Con la administración $Round-Robin$ lowCPU siempre termina primero. Debido a que hace poco uso del procesador (menos que highCPU) y además tiene su " cuota" de procesador correspondiente debido al tipo de administración. Esto hace que termine de ejecutar antes que highCPU.

Con la administración $FIFO$ termina primero el que obtiene el recurso procesador primero. Ninguno de los procesos se bloquea, dando lugar a ejecutar al otro, ya que no realizan E/S.

1.10.2. Modifique la administración de memoria original del MINIX

Decisiones tomadas

MINIX utiliza administración de memoria particionada variable, con *primer zona* (first fit) como algoritmo de selección. Se cambió el procedimiento de alocación de memoria para que utilice el algoritmo de selección *mayor zona*.

El código modificado se encuentra en el Apéndice A.

Expectativas

Éstas administraciones fueron vistas durante el curso. Tanto la administración *mayor zona* como la administración "golosa"de $first\ fit$ tienen casos en la que resultan buenas y casos en las que no. Con lo cual no se espera algún comportamiento en particular, salvo porque se debería de estar tomando siempre el segmento de mayor tamaño.

Resultados

Para realizar las modificaciones, cambiamos la sección de alocación de memoria que ocurre en la función *alloc_mem* del archivo **/usr/src/mm/alloc.c**. En ella se hace un while recorriendo los segmentos libres en la memoria del sistema en busca del de mayor tamaño. Cuando se encuentra (en el caso en que cubra el requerimiento) nos quedamos con ese lugar, caso contrario se devuelve un mensaje de error (Ver Apéndice).

Para poder ver los resultados agregamos dentro de la función código que nos muestra el tamaño de los segmentos libres antes y después de la alocación. Para que dicho código se ejecute debe estar definida la variable DEBUGG. Dicha variable se encuentra en **/usr/include/minix/config.h**.

Usamos el programa de test 4000Clicks.c que puede encontrarse en la carpeta **/usr/grupo8/ej10**:

```
static char datos [4000*256]; /* reserva 1000k */
int main ( void)
{
return 0 ;
}
```

Este programa lo único que hace es reservar 4000 clicks (de 256 Bytes) de memoria que es equivalente a 1000 KB. Lo ejecutamos y observamos lo siguiente:

```
# ls
Se quiere reservar 306 espacios de memoria
1936 514 67 5860
Los segmentos quedaron de la siguiente manera
1936 514 67 5554
Se quiere reservar 440 espacios de memoria
1936 514 67 5860
Los segmentos quedaron de la siguiente manera
1936 514 67 5420
4000Clicks 4000Clicks.c
# ./4000Clicks
Se quiere reservar 306 espacios de memoria
1936 514 67 5860
Los segmentos quedaron de la siguiente manera
1936 514 67 5554
Se quiere reservar 4514 espacios de memoria
1936 514 67 5860
Los segmentos quedaron de la siguiente manera
1936 514 67 1346
#
```

Notar que el cambio fue el deseado, ya que al pedir memoria siempre se ocupa el segmento de mayor tamaño. Usando la administración anterior se ubiese ocupado el primer bloque.

Aclaración: El click es la unidad básica de tamaño de memoria. Si el procesador es Intel está definida como 256 Bytes y varía para otros procesadores. Estos $define$ se encuentran en *const.h*. Para poder ver como se va modificando la memoria hay que bootear con la imagen $imagMemChck$ que está en el directorio **/minix**.

1.11. Ejercicio 11

Se realizaron los siguientes cambios

`/usr/include/minix/callnr.h`

Se incrementó en 1 el define NCALLS Se agreg `#define LLAMSISTEMA` con el nro correspondiente

`/usr/src/mm/proto.h` Se agrega linea:

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

`/usr/src/mm/table.c` Se agrega linea:

```
do_llamsistema,  /\textasteriskcentered 79 = LLAMSISTEMA \textasteriskcentered/
```

`/usr/src/fs/table.c` Se agrega linea:

```
no_sys,  /\textasteriskcentered 79 = LLAMSISTEMA \textasteriskcentered/
```

Se implementa System Call en: `/usr/src/mm/llamsistema.c` Se actualiza Makefile de `/usr/src/mm` para incluirlo.

Se agrega la linea al archivo `/usr/include/minix/com.h`

```
#define OPC_NEWCALL mi_i1
```

Para renombrar el parámetro que se usa para la elección de la consulta que se hace por medio del System Call.

Se recompila kernel

Ejecutar

```
# /usr/src/tools/make install
# /usr/src/tools/make hdboot
```

Nota: no se usó el nombre pedido x la cátedra (newcall), ya que este fue usado para otras pruebas. Se realizó una version del getpid desde MM y desde FS, sendas llamadas a sistema con nombres newcall y othercall respectivamente.

1.11.1. Pruebas

En imagen minix para ejercicio 11

Fuente: `/usr/ej11/llam.c`

Ejecutable: `/usr/ej11/llam`

Modo de prueba

```
./llam [opcion]
```

Opcion es del 1 al 5

1. Entrega el pid del programa
2. Entrega el pid del padre, que, al estar implementado en MM, es el pid del mm o sea 0
3. Entrega el puntero al segmento text, en hexadecimal
4. Entrega el puntero al segmento data, en hexadecimal
5. Entrega el puntero al segmento stack, en hexadecimal

Para las opciones 3, 4 y 5, para verificar la correctitud de la misma se agregó un ciclo while para evitar que el programa termine. Al ejecutar el programa con algunas de estas opciones, al presionar F2 veremos la informacion de los segmentos correspondientes. Para matar al proceso, abrimos otra consola, ejecutamos ps para ver el nro del mismo y con el comando `kill [nro proc]` lo eliminamos.

En el mismo directorio el newcall.c y el othercall.c son source para probar dichas funciones

```
./newcall
./othercall
```

Tienen la misma funcionalidad que `./llam 1` La función `NEWCALL` utiliza la información del Memory Manager (igual que `LLAMSISTEMA`) para averiguar el pid y `OTHERCALL`, el File System.

1.12. Ejercicio 12

Para el diseo de semforos se eligió la esctructura que se ve en el archivo semaf.h. El typedef semaforo se creó para que el usuario del semforo no posea ms información del mismo que el id. La estructura semaf es la que guarda toda la información sensible para el manejo de los semforos y el arreglo semaforos hace las veces de objeto contenedor de los mismos. El nombre del semforo se usa para identificarlo, y tiene un largo mximo dado por el largo de `M3_STRING`, que es un arreglo de char de 15 posiciones. Se tomó esta decisión porque fue mas simple manejar el nombre del semforo a travs de un arreglo que con punteros. El flag semafEnUso sirve para saber si un semaforo tiene asignado algn proceso. Estos se agregan en la lista procEnUso, en el primer lugar vacio que encuentren. El campo valor, indica el valor del semforo (ya que son semforos contadores). La cola de procesos bloqueados est implementada sobre un arreglo y 2 apuntadores, al inicio y al fin de la lista respectivamente. La lista va haciendose circular por el arreglo, una vez que se llega al final del mismo, vuelve a comenzar.

El usuario posee las siguientes funciones para el manejo de semforos:

`semaforo crear_sem(char* nombre, int valor)` : A partir de un nombre y un valor crea un semforo, le asigna nombre, valor y el proceso que lo creó. Retorna un "semaforo" (int). Si el nombre ya exista en la lista de semforos, asigna al proceso creador e ignora el valor entregado. Si no hay ms semforos disponibles retorna mensaje de error.

`int p_sem(semaforo x)` : Ejecuta P al semforo x. Decrementa el valor del semforo. Si el valor resulta quedar en 0 o menor, el proceso se bloquea, se agrega a la cola de bloqueados. Si el semforo no perteneca al proceso se retorna mensaje de error.

`int v_sem(semaforo x)` : Ejecuta V al semforo x. Incrementa el valor del semforo. Si el valor resulta quedar en 0 o menor, el primer proceso de la cola de bloqueados se libera. Si el semforo no perteneca al proceso se retorna mensaje de error.

`int liberar_sem(semaforo x)` : Dado un semforo x libera a todos los procesos bloqueados en l, borra los procesos de la lista e inicializa al semforo. Este mtodo viola la exclusión mutua.

`void inicializar()` : Vacia las listas de procesos, flags, valor y nombre de todos los semforos

Cambios para ejercicio 12

`/usr/include/minix/callnr.h`

Se incrementó en 5 el define `NCALLS` (Se definieron 5 llamadas a sistema)

Se agregó `#define CREAR_SEM` con el nro correspondiente

Se agregó `#define P_SEM` con el nro correspondiente

Se agregó `#define V_SEM` con el nro correspondiente

Se agregó `#define LIBERAR_SEM` con el nro correspondiente

Se agregó `#define INIT_ALL_SEM` con el nro correspondiente

`/usr/src/mm/proto.h`

Se agrega linea:

```
_PROTOTYPE( int do_crear_sem, (void) );
_PROTOTYPE( int do_p_sem, (void) );
_PROTOTYPE( int do_v_sem, (void) );
_PROTOTYPE( int do_liberar_sem, (void) );
_PROTOTYPE( int do_init_all_sem, (void) );
```

`/usr/src/mm/table.c`

Se agregan lineas:

```
do_crear_sem,  /* 80 = CREAR_SEM */
do_p_sem,  /* 81 = P_SEM  */
do_v_sem,  /* 82 = V_SEM */
do_liberar_sem,  /* 83 = LIBERAR_SEM */
do_init_all_sem,  /* 84 = INIT_ALL_SEM */
```

`/usr/src/fs/table.c`

Se agregan lineas:

```
no_sys,  /* 80 = CREAR_SEM */
no_sys,  /* 81 = P_SEM  */
no_sys,  /* 82 = V_SEM */
no_sys,  /* 83 = LIBERAR_SEM */
no_sys,  /* 84 = INIT_ALL_SEM */
```

Se define estructura del semforo en: `/usr/src/mm/semaf.h` (archivo source)

Se implementan system call en: `/usr/src/mm/semaf.c` (archivo source)

Se actualiza Makefile de `/usr/src/mm` para incluirlos.

Se crea `/usr/include/minix/constsemaf.h` (archivo source) Posee unos defines para renombrar los parmetros del mensaje

Se actualiza archivo system.c para agregar los system call encargados de bloquear y desbloquear un proceso. `/usr/src/kernel/system.c` (archivo source)

Se recompila kernel

Ejecutar

```
# /usr/src/tools/make install
# /usr/src/tools/make hdboot
```

Se crean archivos para poder llamar el system call desde una librera y cumplir el standard posix:

```
/usr/src/lib/syscall/crear_sem.s (archivo source)
/usr/src/lib/syscall/p_sem.s (archivo source)
/usr/src/lib/syscall/v_sem.s (archivo source)
/usr/src/lib/syscall/liberar_sem.s (archivo source)
/usr/src/lib/syscall/init.s (archivo source)
```

Se actualiza Makefile de `/usr/src/lib/syscall` para incluirlo.

`/usr/src/lib/posix/_sem.c` (archivo source) Se actualiza Makefile de `/usr/src/lib/posix` para incluirlo.

Se crean archivos para poder manejar como función los system call al kernel:

```
/usr/src/lib/syslib/sys_block.c (archivo source)
/usr/src/lib/syslib/sys_unblock.c (archivo source)
```

Se actualiza Makefile de `/usr/src/lib/syslib` para incluirlo.

`/usr/include/minix/syslib.h`

Se agregan lineas:

```
_PROTOTYPE( int sys_block, (int_proc) );
_PROTOTYPE( int sys_block, (int_proc) );
```

Ejecutar

```
# /usr/src/lib/make all
# /usr/src/lib/make install
```

1.12.1. Pruebas ejercicio 12

En imagen minix para ejercicio 12

Directorio `/usr/ej12`:

Estn implementados productor consumidor y secuencia de procesos A-B-B-A-C

Productor-Consumidor

Ejecutar

```
# ./init --> inicializa semforos
# ./prod-cons --> define semforos x e y para pruebas productor consumidor

# ./productor  > /dev/null &   --> ejecuta productor en 2do plano
# ./consumidor > /dev/null &  --> ejecuta consumidor en 2do plano
```

Ejecutar indistintamente productor y consumidor. Cada vez que se produzca algo se mostrar en consola "Produce" y cada vez que se consuma, "Consume"

Para finalizar la prueba ejecutar:

```
# ./libsem
```

que libera todos los procesos bloqueados

Secuencia A-B-B-A-C

Ejecutar

```
# ./init --> inicializa semforos
# ./sem --> define todos los semforos necesarios
```

Los procesos involucrados son ./A ./B o ./C . Cada vez que se ejecutan se muestra en consola "este es A", "este es B" o "este es C" respectivamente.

```
# ./A > /dev/null &   --> ejecuta A en 2do plano
# ./B > /dev/null &   --> ejecuta B en 2do plano
# ./C > /dev/null &  --> ejecuta C en 2do plano
```

Hay un script (pru.sh) que tiene un ejemplo:

```
 ./A & ./A & ./A & ./B & ./B & ./B & ./C & ./C & ./C & >  /dev/null &
```

Que debera mostrar lo siguiente:

```
este es A
este es B
este es B
este es A
este es C
este es A
este es B
```

Ejecutando el comando

```
# ps
```

Podemos corroborar que existen 2 procesos C que estn bloqueados.

Ejecutar indistintamente productor y consumidor. Cada vez que se produzca algo se mostrar en consola "Produce" y cada vez que se consuma, "Consume"

Para finalizar la prueba ejecutar:

```
# ./libsem
```

que libera todos los procesos bloqueados

2. Apendice

2.1. Fuentes del ejercicio 11

llamsistema.c

```
#include "mm.h"
#include <minix/callnr.h>
#include <signal.h>
#include "mproc.h"
#include <stdlib.h>
#include <minix/com.h>
#include <minix/type.h>

PUBLIC int do_llamsistema( void ) {

register struct mproc *rmp = mp;

register int r;

switch( mm_in.OPC_NEWCALL ) {

case 1: /* pid */

r = mproc[who].mp_pid;
break;

case 2: /* ppid */

break;

case 3: /* text */

r = (int) mproc[who].mp_seg[T].mem_phys;
break;

case 4: /* data */

r = (int) mproc[who].mp_seg[D].mem_phys;
break;

case 5: /* stack */

r = (int) mproc[who].mp_seg[S].mem_phys;
break;

default:

break;
```

```
}

return r;

}
```

_llamsistema.c

```
#include <lib.h>
#include <unistd.h>
#include <minix/com.h>
#define llamsistema _llamsistema

PUBLIC int llamsistema( int opcion ) {

message m;

switch( opcion ) {

case 1: /* pid */
case 3: /* text */
case 4: /* data */
case 5: /* stack */

m.OPC_NEWCALL = opcion;

return(_syscall( MM, LLAMSISTEMA, &m ));

case 2: /* ppid */

_syscall( MM, LLAMSISTEMA, &m );
return m.m2_i1;

default:

return -1;

}
}
```

llamsistema.s

```
.sect .text
.extern __llamsistema
.define _llamsistema
```

```
.align 2

_llamsistema:
jmp __llamsistema
```

2.2. Fuentes del ejercicio 12

constsemaf.h

```
#define PROC1  m1_i1
#define NOMBRE_SEM m3_ca1
#define VALOR m3_i1
#define SEMAFORO m1_i2
```

crear_sem.s

```
.sect .text
.extern __crear_sem
.define _crear_sem

.align 2

_crear_sem:
jmp __crear_sem
```

init.s

```
.sect .text
.extern __inicializar
.define _inicializar

.align 2

_liberar_sem:
jmp __inicializar
```

liberar_sem.s

```
.sect .text
.extern __liberar_sem
.define _liberar_sem

.align 2

_liberar_sem:
jmp __liberar_sem
```

p_sem.s

```
.sect .text
.extern __p_sem
.define _p_sem

.align 2

_p_sem:
jmp __p_sem
```

semaf.c

```
#include "mm.h"
#include <minix/callnr.h>
#include <signal.h>
#include "mproc.h"
#include <stdlib.h>
#include "semaf.h"
#include <minix/constsemaf.h>
#include <minix/com.h>
#include <minix/type.h>
#include <string.h>

FORWARD _PROTOTYPE( int do_is_sem, (void) );
FORWARD _PROTOTYPE( int do_val_sem, (void) );
FORWARD _PROTOTYPE( int do_get_next_bloq_proc, (void) );
FORWARD _PROTOTYPE( void do_add_bloq_proc, (int) );
FORWARD _PROTOTYPE( void do_init_sem, (void) );

/*===========================================================================*
 * do_crear_sem      *
 *===========================================================================*/

PUBLIC int do_crear_sem(void) {

register struct semaf *sp = semaforos;

semaforo s;

pid_t procID = mproc[who].mp_pid;
char* nombre = mm_in.NOMBRE_SEM;
int valor = mm_in.VALOR;

int i, j;

i = 0;

/* printf("este es el proc en crear %d\n",procID); */
```

```
for(i=0;i<MAX_SEM;i++) {
/* Busco si el nombre corresponde a un semaforo ya creado */
if (!strcmp(semaforos[i].nombre, nombre)) {
/* printf("semaforo encontrado!!\n"); */
break;
}
}

/* i se incrementa en uno mas, no se xq, entonces lo decremento */

if ( i < MAX_SEM ) {
/* Si el semaforo ya existe lo selecciono */
s = i;

} else {
/* Sino busco el primer semaforo sin uso */
for(i=0;i<MAX_SEM;i++) {
if (semaforos[i].semafEnUso == 0) {

s = i;
/* Defino el semaforo */

strcpy(semaforos[s].nombre, nombre);
semaforos[s].valor = valor;
semaforos[s].semafEnUso = 1;
semaforos[s].cant_proc  = 0;
semaforos[s].inicio_cola_bloq = 0;
semaforos[s].fin_cola_bloq    = 0;

break;

}
}
}


if (i == MAX_SEM) {

/* si no hay semaforos disponibles retorno error */
return -1;

} else {

semaforos[s].cant_proc++;

/* Asigno al proceso al primer lugar vacio de la lista de procesos */
for(j=0;j<MAX_PROC;j++) {
```

```
if (semaforos[s].procEnUso[j] == 0) {

semaforos[s].procEnUso[j] = procID;
break;

}
}

return s;

}

}


/*===========================================================================*
 * do_is_sem           *
 *===========================================================================*/

PRIVATE int do_is_sem(void) {

register int i;

register struct semaf *sp = semaforos;

pid_t procID = mproc[who].mp_pid;
semaforo s = mm_in.SEMAFORO;

/* printf("el proc es: %d\n", procID); */

for(i=0;i<MAX_PROC;i++) {
/*
printf("proc en uso actual: %d\n", semaforos[s].procEnUso[i]);
*/
if (semaforos[s].procEnUso[i] == procID) {
return 1;
}
}

return 0;

}

/*===========================================================================*
 * do_p_sem        *
 *===========================================================================*/

PUBLIC int do_p_sem(void) {
```

```
message m;

register struct semaf *sp = semaforos;

register int proc_nr;
register semaforo s = mm_in.SEMAFORO;

       /**
 * Calculo la posicion del proceso en la tabla.
 * Esta es el puntero al proceso menos el puntero a la lista
 * de procesos.
   */

proc_nr = (int) (mp - mproc);

if (do_is_sem()) {

/* decremento el valor del semaforo */
semaforos[s].valor--;

if(do_val_sem() < 0 ) {

/* Agrego el proceso a bloqueados */
do_add_bloq_proc(proc_nr);

/*  debo bloquear el proceso */
sys_block(proc_nr);

}

return 0;

} else {

return -1;

}

}

/*===========================================================================*
 * do_v_sem        *
 *===========================================================================*/

PUBLIC int do_v_sem(void) {

message m;

register struct semaf *sp = semaforos;
```

```
register int proc_nr;
semaforo s = mm_in.SEMAFORO;

if (do_is_sem()) {

/* incremento el valor del semaforo */
semaforos[s].valor++;

if(do_val_sem() <= 0 ) {

/* Busco si existe algun proceso bloqueado */
proc_nr = do_get_next_bloq_proc();

if (proc_nr > 0) {

sys_unblock(proc_nr);

}

}

return 0;

} else {

return -1;

}

}

/*===========================================================================*
 * do_val_sem      *
 *===========================================================================*/

PRIVATE int do_val_sem(void) {

register struct semaf *sp = semaforos;

register semaforo s = mm_in.SEMAFORO;

return semaforos[s].valor;

}

/*===========================================================================*
 * do_liberar_sem         *
 *===========================================================================*/
```

```
PUBLIC int do_liberar_sem(void) {

message m;

register struct semaf *sp = semaforos;

register int proc_nr;

pid_t procID = mproc[who].mp_pid;
semaforo s = mm_in.SEMAFORO;

int i;

if (do_is_sem()) {

semaforos[s].cant_proc--;

if (semaforos[s].cant_proc > 0) {
/* si el semaforo posee procesos bloqueados los desbloquea */

for(i=semaforos[s].inicio_cola_bloq;i<=semaforos[s].inicio_cola_bloq;i++) {

proc_nr = semaforos[s].procBloqueados[i];
sys_unblock(proc_nr);

}

/* Luego elimino todos los procesos asociados referencia al proceso */

for(i=0;i<MAX_PROC;i++) {

semaforos[s].procEnUso[i] = 0;

}

}

/* finalmente inicializo el semaforo */

do_init_sem();

return 0;

} else {

return -1;

}
```

```
}

/*===========================================================================*
 * do_get_next_bloq_proc      *
 *===========================================================================*/

PRIVATE int do_get_next_bloq_proc(void) {

register struct semaf *sp = semaforos;

semaforo s = mm_in.SEMAFORO;

register int r = 0;

if (semaforos[s].inicio_cola_bloq != semaforos[s].inicio_cola_bloq) {

/* si existe algun proceso bloqueado lo elijo, y adelanto el inicio de la cola */

semaforos[s].cant_proc--;

/* primer proceso bloqueado */
r = semaforos[s].procBloqueados[semaforos[s].inicio_cola_bloq];

semaforos[s].inicio_cola_bloq++;
     semaforos[s].inicio_cola_bloq %= MAX_PROC;

}
/* sino retorna 0 */

return r;

}

/*===========================================================================*
 * do_add_bloq_proc      *
 *===========================================================================*/

PRIVATE void do_add_bloq_proc(int proc_nr) {

register struct semaf *sp = semaforos;

semaforo s = mm_in.SEMAFORO;

semaforos[s].cant_proc++;

semaforos[s].procBloqueados[semaforos[s].fin_cola_bloq] = proc_nr;

semaforos[s].fin_cola_bloq++;
```

```
semaforos[s].fin_cola_bloq %= MAX_PROC;

}


/*===========================================================================*
 * do_init_sem      *
 *===========================================================================*/

PRIVATE void do_init_sem(void) {

register struct semaf *sp = semaforos;

register semaforo s = mm_in.SEMAFORO;

semaforos[s].semafEnUso = 0;

semaforos[s].valor  = 0;
semaforos[s].cant_proc  = 0;

semaforos[s].inicio_cola_bloq = 0;
semaforos[s].fin_cola_bloq = 0;

}



/*===========================================================================*
 * do_init_all_sem      *
 *===========================================================================*/

PUBLIC int do_init_all_sem(void) {

int i, j;
register struct semaf *sp = semaforos;

for(i=0;i<MAX_SEM;i++) {

strcpy(semaforos[i].nombre,"");
semaforos[i].semafEnUso = 0;
semaforos[i].valor  = 0;
semaforos[i].cant_proc  = 0;
semaforos[i].inicio_cola_bloq = 0;
semaforos[i].fin_cola_bloq = 0;


for(j=0;j<MAX_PROC;j++) {

semaforos[i].procBloqueados[j] = 0;
```

```
semaforos[i].procEnUso[j]  = 0;

}

}

return 0;
}
```

semaf.h

```
#define MAX_SEM  100
#define MAX_PROC  100

#include <minix/type.h>

typedef int semaforo;

struct semaf {

char nombre[M3_STRING];

int semafEnUso; /* 0 o 1 */

int valor;
int cant_proc;

int procEnUso[MAX_PROC];

int inicio_cola_bloq;
int fin_cola_bloq;
int procBloqueados[MAX_PROC];

};

struct semaf semaforos[MAX_SEM];
```

semaforo.h

```
/* El header <semaforos.h> contiene constantes para la definicion de semaforos */

#ifndef _SEMAFOROS_H_
#define _SEMAFOROS_H_

#define ERROR -1
typedef int semaforo;

_PROTOTYPE( semaforo crear_sem, (char* nombre, int valor)  );
```

```
_PROTOTYPE( int p_sem, (semaforo)  );

_PROTOTYPE( int v_sem, (semaforo) );

_PROTOTYPE( int liberar_sem, (semaforo) );

_PROTOTYPE( void inicializar, (void)  );

#endif
```

system.c

```
/* This task handles the interface between file system and kernel as well as
 * between memory manager and kernel.  System services are obtained by sending
 * sys_task() a message specifying what is needed.  To make life easier for
 * MM and FS, a library is provided with routines whose names are of the
 * form sys_xxx, e.g. sys_xit sends the SYS_XIT message to sys_task.  The
 * message types and parameters are:
 *
 *   SYS_FORK  informs kernel that a process has forked
 *   SYS_NEWMAP  allows MM to set up a process memory map
 *   SYS_GETMAP  allows MM to get a process' memory map
 *   SYS_EXEC  sets program counter and stack pointer after EXEC
 *   SYS_XIT  informs kernel that a process has exited
 *   SYS_GETSP  caller wants to read out some process' stack pointer
 *   SYS_TIMES  caller wants to get accounting times for a process
 *   SYS_ABORT  MM or FS cannot go on; abort MINIX
 *   SYS_FRESH  start with a fresh process image during EXEC (68000 only)
 *   SYS_SENDSIG send a signal to a process (POSIX style)
 *   SYS_SIGRETURN complete POSIX-style signalling
 *   SYS_KILL  cause a signal to be sent via MM
 *   SYS_ENDSIG  finish up after SYS_KILL-type signal
 *   SYS_COPY  request a block of data to be copied between processes
 *   SYS_VCOPY   request a series of data blocks to be copied between procs
 *   SYS_GBOOT  copies the boot parameters to a process
 *   SYS_MEM  returns the next free chunk of physical memory
 *   SYS_UMAP  compute the physical address for a given virtual address
 *   SYS_TRACE  request a trace operation
 *
 * Message types and parameters:
 *
 *    m_type       PROC1     PROC2      PID     MEM_PTR
 * ------------------------------------------------------
 * | SYS_FORK   | parent  |  child  |   pid   |         |
 * |------------+---------+---------+---------+---------|
 * | SYS_NEWMAP | proc nr |         |         | map ptr |
 * |------------+---------+---------+---------+---------|
 * | SYS_EXEC   | proc nr | traced  | new sp  |         |
```

```
* |------------+---------+---------+---------+---------|
* | SYS_XIT    | parent  | exitee  |         |         |
* |------------+---------+---------+---------+---------|
* | SYS_GETSP  | proc nr |         |         |         |
* |------------+---------+---------+---------+---------|
* | SYS_TIMES  | proc nr |         | buf ptr |         |
* |------------+---------+---------+---------+---------|
* | SYS_ABORT  |         |         |         |         |
* |------------+---------+---------+---------+---------|
* | SYS_FRESH  | proc nr | data_cl |         |         |
* |------------+---------+---------+---------+---------|
* | SYS_GBOOT  | proc nr |         |         | bootptr |
* |------------+---------+---------+---------+---------|
* | SYS_GETMAP | proc nr |         |         | map ptr |
* ---------------------------------------------------
*
*    m_type          m1_i1     m1_i2     m1_i3       m1_p1
* ----------------+---------+---------+---------+--------------
* | SYS_VCOPY     |  src p  |  dst p  | vec siz | vc addr     |
* |---------------+---------+---------+---------+-------------|
* | SYS_SENDSIG   | proc nr |         |         | smp         |
* |---------------+---------+---------+---------+-------------|
* | SYS_SIGRETURN | proc nr |         |         | scp         |
* |---------------+---------+---------+---------+-------------|
* | SYS_ENDSIG    | proc nr |         |         |             |
* ------------------------------------------------------------
*
*    m_type       m2_i1     m2_i2     m2_l1     m2_l2
* ------------------------------------------------------
* | SYS_TRACE  | proc_nr | request |  addr   |  data   |
* ------------------------------------------------------
*
*
*    m_type       m6_i1     m6_i2     m6_i3     m6_f1
* ------------------------------------------------------
* | SYS_KILL   | proc_nr  |  sig    |         |         |
* ------------------------------------------------------
*
*
*    m_type      m5_c1   m5_i1    m5_l1   m5_c2   m5_i2    m5_l2   m5_l3
* ------------------------------------------------------------------------
* | SYS_COPY   |src seg|src proc|src vir|dst seg|dst proc|dst vir| byte ct |
* ------------------------------------------------------------------------
* | SYS_UMAP   |  seg  |proc nr |vir adr|       |        |       | byte ct |
* ------------------------------------------------------------------------
*
*
*    m_type      m1_i1      m1_i2      m1_i3
* |------------+----------+----------+----------
```

```
 * | SYS_MEM    | mem base | mem size | tot mem |
 * ---------------------------------------------
 *
 *
 *    m_type        m1_i1      m1_i2      m1_i3
 * ---------------+----------+----------+----------
 * | SYS_BLOCK    | proc_nr  |     |   |
 * |---------------------------------------------|
 * | SYS_UNBLOCK  | proc_nr  |     |   |
 * ---------------------------------------------
 *
 *
 * In addition to the main sys_task() entry point, there are 5 other minor
 * entry points:
 *   cause_sig: take action to cause a signal to occur, sooner or later
 *   inform: tell MM about pending signals
 *   numap: umap D segment starting from process number instead of pointer
 *   umap: compute the physical address for a given virtual address
 *   alloc_segments: allocate segments for 8088 or higher processor
 */

#include "kernel.h"
#include <signal.h>
#include <unistd.h>
#include <sys/sigcontext.h>
#include <sys/ptrace.h>
#include <minix/boot.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"
#if (CHIP == INTEL)
#include "protect.h"
#endif

/* PSW masks. */
#define IF_MASK 0x00000200
#define IOPL_MASK 0x003000

PRIVATE message m;

FORWARD _PROTOTYPE( int do_abort, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_copy, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_exec, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_fork, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_gboot, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_getsp, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_kill, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_mem, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_newmap, (message *m_ptr) );
```

```
FORWARD _PROTOTYPE( int do_sendsig, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_sigreturn, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_endsig, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_times, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_trace, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_umap, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_xit, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_vcopy, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_getmap, (message *m_ptr) );

FORWARD _PROTOTYPE( int do_block, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_unblock, (message *m_ptr) );

#if (SHADOWING == 1)
FORWARD _PROTOTYPE( int do_fresh, (message *m_ptr) );
#endif

/*===========================================================================*
 * sys_task      *
 *===========================================================================*/
PUBLIC void sys_task()
{
/* Main entry point of sys_task.  Get the message and dispatch on type. */

  register int r;

  while (TRUE) {
receive(ANY, &m);

switch (m.m_type) { /* which system call */
    case SYS_FORK: r = do_fork(&m); break;
    case SYS_NEWMAP: r = do_newmap(&m); break;
    case SYS_GETMAP: r = do_getmap(&m); break;
    case SYS_EXEC: r = do_exec(&m); break;
    case SYS_XIT: r = do_xit(&m); break;
    case SYS_GETSP: r = do_getsp(&m); break;
    case SYS_TIMES: r = do_times(&m); break;
    case SYS_ABORT: r = do_abort(&m); break;
#if (SHADOWING == 1)
    case SYS_FRESH: r = do_fresh(&m); break;
#endif
    case SYS_SENDSIG: r = do_sendsig(&m); break;
    case SYS_SIGRETURN: r = do_sigreturn(&m); break;
    case SYS_KILL: r = do_kill(&m); break;
    case SYS_ENDSIG: r = do_endsig(&m); break;
    case SYS_COPY: r = do_copy(&m); break;
            case SYS_VCOPY: r = do_vcopy(&m); break;
    case SYS_GBOOT: r = do_gboot(&m); break;
    case SYS_MEM: r = do_mem(&m); break;
```

```
    case SYS_UMAP: r = do_umap(&m); break;
    case SYS_TRACE: r = do_trace(&m); break;
    case SYS_BLOCK: r = do_block(&m); break;
    case SYS_UNBLOCK: r = do_unblock(&m); break;
    default: r = E_BAD_FCN;
}

m.m_type = r; /* 'r' reports status of call */
send(m.m_source, &m); /* send reply to caller */
  }
}

/*===========================================================================*
 * do_block      *
 *===========================================================================*/
 PRIVATE int do_block(m_ptr)
 register message *m_ptr; /* pointer to request message */
{
/* Handle sys_block().  A process is blocked by semaphore. */

register struct proc *rp;

rp = proc_addr(m_ptr->PROC1);

if (rp->p_flags == 0) lock_unready(rp);

rp->p_flags |= BLOCK_X_SEM; /* bloquea x semaforo */

return(OK);
}

/*===========================================================================*
 * do_unblock      *
 *===========================================================================*/
  PRIVATE int do_unblock(m_ptr)
  register message *m_ptr; /* pointer to request message */
 {
/* Handle sys_unblock().  Process is unblocked by semaphore. */

  register struct proc *rp;

  rp = proc_addr(m_ptr->PROC1); /* process to unblocks id */

  rp->p_flags &= ~BLOCK_X_SEM; /* desbloquea x semaforo */

  if (rp->p_flags == 0) lock_ready(rp);

  return(OK);
}
```

```
/*===========================================================================*
 * do_fork      *
 *===========================================================================*/
PRIVATE int do_fork(m_ptr)
register message *m_ptr; /* pointer to request message */
{
/* Handle sys_fork().  m_ptr->PROC1 has forked.  The child is m_ptr->PROC2. */

#if (CHIP == INTEL)
  reg_t old_ldt_sel;
#endif
  register struct proc *rpc;
  struct proc *rpp;

  if (!isoksusern(m_ptr->PROC1) || !isoksusern(m_ptr->PROC2))
return(E_BAD_PROC);
  rpp = proc_addr(m_ptr->PROC1);
  rpc = proc_addr(m_ptr->PROC2);

  /* Copy parent 'proc' struct to child. */
#if (CHIP == INTEL)
  old_ldt_sel = rpc->p_ldt_sel; /* stop this being obliterated by copy */
#endif

  *rpc = *rpp; /* copy 'proc' struct */

#if (CHIP == INTEL)
  rpc->p_ldt_sel = old_ldt_sel;
#endif
  rpc->p_nr = m_ptr->PROC2; /* this was obliterated by copy */

#if (SHADOWING == 0)
  rpc->p_flags |= NO_MAP; /* inhibit the process from running */
#endif

  rpc->p_flags &= ~(PENDING | SIG_PENDING | P_STOP);

  /* Only 1 in group should have PENDING, child does not inherit trace status*/
  sigemptyset(&rpc->p_pending);
  rpc->p_pendcount = 0;
  rpc->p_pid = m_ptr->PID; /* install child's pid */
  rpc->p_reg.retreg = 0; /* child sees pid = 0 to know it is child */

  rpc->user_time = 0; /* set all the accounting times to 0 */
  rpc->sys_time = 0;
  rpc->child_utime = 0;
  rpc->child_stime = 0;
```

```
#if (SHADOWING == 1)
  rpc->p_nflips = 0;
  mkshadow(rpp, (phys_clicks)m_ptr->m1_p1); /* run child first */
#endif

  return(OK);
}


/*===========================================================================*
 * do_newmap      *
 *===========================================================================*/
PRIVATE int do_newmap(m_ptr)
message *m_ptr; /* pointer to request message */
{
/* Handle sys_newmap().  Fetch the memory map from MM. */

  register struct proc *rp;
  phys_bytes src_phys;
  int caller; /* whose space has the new map (usually MM) */
  int k; /* process whose map is to be loaded */
  int old_flags; /* value of flags before modification */
  struct mem_map *map_ptr; /* virtual address of map inside caller (MM) */

  /* Extract message parameters and copy new memory map from MM. */
  caller = m_ptr->m_source;
  k = m_ptr->PROC1;
  map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
  if (!isokprocn(k)) return(E_BAD_PROC);
  rp = proc_addr(k); /* ptr to entry of user getting new map */

  /* Copy the map from MM. */
  src_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, sizeof(rp->p_map));
  if (src_phys == 0) panic("bad call to sys_newmap", NO_NUM);
  phys_copy(src_phys, vir2phys(rp->p_map), (phys_bytes) sizeof(rp->p_map));

#if (SHADOWING == 0)
#if (CHIP != M68000)
  alloc_segments(rp);
#else
  pmmu_init_proc(rp);
#endif
  old_flags = rp->p_flags; /* save the previous value of the flags */
  rp->p_flags &= ~NO_MAP;
  if (old_flags != 0 && rp->p_flags == 0) lock_ready(rp);
#endif

  return(OK);
}
```

```
/*===========================================================================*
 * do_getmap      *
 *===========================================================================*/
PRIVATE int do_getmap(m_ptr)
message *m_ptr; /* pointer to request message */
{
/* Handle sys_getmap().  Report the memory map to MM. */

  register struct proc *rp;
  phys_bytes dst_phys;
  int caller; /* where the map has to be stored */
  int k; /* process whose map is to be loaded */
  struct mem_map *map_ptr; /* virtual address of map inside caller (MM) */

  /* Extract message parameters and copy new memory map to MM. */
  caller = m_ptr->m_source;
  k = m_ptr->PROC1;
  map_ptr = (struct mem_map *) m_ptr->MEM_PTR;

  if (!isokprocn(k))
panic("do_getmap got bad proc: ", m_ptr->PROC1);

  rp = proc_addr(k); /* ptr to entry of the map */

  /* Copy the map to MM. */
  dst_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, sizeof(rp->p_map));
  if (dst_phys == 0) panic("bad call to sys_getmap", NO_NUM);
  phys_copy(vir2phys(rp->p_map), dst_phys, sizeof(rp->p_map));

  return(OK);
}


/*===========================================================================*
 * do_exec      *
 *===========================================================================*/
PRIVATE int do_exec(m_ptr)
register message *m_ptr; /* pointer to request message */
{
/* Handle sys_exec().  A process has done a successful EXEC. Patch it up. */

  register struct proc *rp;
  reg_t sp; /* new sp */
  phys_bytes phys_name;
  char *np;
#define NLEN (sizeof(rp->p_name)-1)
```

```
  if (!isoksusern(m_ptr->PROC1)) return E_BAD_PROC;
  /* PROC2 field is used as flag to indicate process is being traced */
  if (m_ptr->PROC2) cause_sig(m_ptr->PROC1, SIGTRAP);
  sp = (reg_t) m_ptr->STACK_PTR;
  rp = proc_addr(m_ptr->PROC1);
  rp->p_reg.sp = sp; /* set the stack pointer */
#if (CHIP == M68000)
  rp->p_splow = sp; /* set the stack pointer low water */
#ifdef FPP
  /* Initialize fpp for this process */
  fpp_new_state(rp);
#endif
#endif
  rp->p_reg.pc = (reg_t) m_ptr->IP_PTR; /* set pc */
  rp->p_alarm = 0; /* reset alarm timer */
  rp->p_flags &= ~RECEIVING; /* MM does not reply to EXEC call */
  if (rp->p_flags == 0) lock_ready(rp);

  /* Save command name for debugging, ps(1) output, etc. */
  phys_name = numap(m_ptr->m_source, (vir_bytes) m_ptr->NAME_PTR,
(vir_bytes) NLEN);
  if (phys_name != 0) {
phys_copy(phys_name, vir2phys(rp->p_name), (phys_bytes) NLEN);
for (np = rp->p_name; (*np & BYTE) >= ' '; np++) {}
*np = 0;
  }
  return(OK);
}


/*===========================================================================*
 * do_xit      *
 *===========================================================================*/
PRIVATE int do_xit(m_ptr)
message *m_ptr; /* pointer to request message */
{
/* Handle sys_xit().  A process has exited. */

  register struct proc *rp, *rc;
  struct proc *np, *xp;
  int parent; /* number of exiting proc's parent */
  int proc_nr; /* number of process doing the exit */
  phys_clicks base, size;

  parent = m_ptr->PROC1; /* slot number of parent process */
  proc_nr = m_ptr->PROC2; /* slot number of exiting process */
  if (!isoksusern(parent) || !isoksusern(proc_nr)) return(E_BAD_PROC);
  rp = proc_addr(parent);
  rc = proc_addr(proc_nr);
```

```
  lock();
  rp->child_utime += rc->user_time + rc->child_utime; /* accum child times */
  rp->child_stime += rc->sys_time + rc->child_stime;
  unlock();
  rc->p_alarm = 0; /* turn off alarm timer */
  if (rc->p_flags == 0) lock_unready(rc);

#if (SHADOWING == 1)
  rmshadow(rc, &base, &size);
  m_ptr->m1_i1 = (int)base;
  m_ptr->m1_i2 = (int)size;
#endif

  strcpy(rc->p_name, "<noname>"); /* process no longer has a name */

  /* If the process being terminated happens to be queued trying to send a
   * message (i.e., the process was killed by a signal, rather than it doing an
   * EXIT), then it must be removed from the message queues.
   */
  if (rc->p_flags & SENDING) {
/* Check all proc slots to see if the exiting process is queued. */
for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
if (rp->p_callerq == NIL_PROC) continue;
if (rp->p_callerq == rc) {
/* Exiting process is on front of this queue. */
rp->p_callerq = rc->p_sendlink;
break;
} else {
/* See if exiting process is in middle of queue. */
np = rp->p_callerq;
while ( ( xp = np->p_sendlink) != NIL_PROC)
if (xp == rc) {
np->p_sendlink = xp->p_sendlink;
break;
} else {
np = xp;
}
}
}
  }
#if (CHIP == M68000) && (SHADOWING == 0)
  pmmu_delete(rc); /* we're done remove tables */
#endif

  if (rc->p_flags & PENDING) --sig_procs;
  sigemptyset(&rc->p_pending);
  rc->p_pendcount = 0;
  rc->p_flags = P_SLOT_FREE;
  return(OK);
```

```
}


/*===========================================================================*
 * do_getsp      *
 *===========================================================================*/
PRIVATE int do_getsp(m_ptr)
register message *m_ptr; /* pointer to request message */
{
/* Handle sys_getsp().  MM wants to know what sp is. */

  register struct proc *rp;

  if (!isoksusern(m_ptr->PROC1)) return(E_BAD_PROC);
  rp = proc_addr(m_ptr->PROC1);
  m_ptr->STACK_PTR = (char *) rp->p_reg.sp; /* return sp here (bad type) */
  return(OK);
}


/*===========================================================================*
 * do_times      *
 *===========================================================================*/
PRIVATE int do_times(m_ptr)
register message *m_ptr; /* pointer to request message */
{
/* Handle sys_times().  Retrieve the accounting information. */

  register struct proc *rp;

  if (!isoksusern(m_ptr->PROC1)) return E_BAD_PROC;
  rp = proc_addr(m_ptr->PROC1);

  /* Insert the times needed by the TIMES system call in the message. */
  lock(); /* halt the volatile time counters in rp */
  m_ptr->USER_TIME   = rp->user_time;
  m_ptr->SYSTEM_TIME = rp->sys_time;
  unlock();
  m_ptr->CHILD_UTIME = rp->child_utime;
  m_ptr->CHILD_STIME = rp->child_stime;
  m_ptr->BOOT_TICKS  = get_uptime();
  return(OK);
}


/*===========================================================================*
 * do_abort      *
 *===========================================================================*/
PRIVATE int do_abort(m_ptr)
```

```
message *m_ptr; /* pointer to request message */
{
/* Handle sys_abort.  MINIX is unable to continue.  Terminate operation. */
  char monitor_code[64];
  phys_bytes src_phys;

  if (m_ptr->m1_i1 == RBT_MONITOR) {
/* The monitor is to run user specified instructions. */
src_phys = numap(m_ptr->m_source, (vir_bytes) m_ptr->m1_p1,
(vir_bytes) sizeof(monitor_code));
if (src_phys == 0) panic("bad monitor code from", m_ptr->m_source);
phys_copy(src_phys, vir2phys(monitor_code),
(phys_bytes) sizeof(monitor_code));
reboot_code = vir2phys(monitor_code);
  }
  wreboot(m_ptr->m1_i1);
  return(OK); /* pro-forma (really EDISASTER) */
}


#if (SHADOWING == 1)
/*===========================================================================*
 * do_fresh      *
 *===========================================================================*/
PRIVATE int do_fresh(m_ptr)     /* for 68000 only */
message *m_ptr; /* pointer to request message */
{
/* Handle sys_fresh.  Start with fresh process image during EXEC. */

  register struct proc *p;
  int proc_nr; /* number of process doing the exec */
  phys_clicks base, size;
  phys_clicks c1, nc;

  proc_nr = m_ptr->PROC1; /* slot number of exec-ing process */
  if (!isokprocn(proc_nr)) return(E_BAD_PROC);
  p = proc_addr(proc_nr);
  rmshadow(p, &base, &size);
  do_newmap(m_ptr);
  c1 = p->p_map[D].mem_phys;
  nc = p->p_map[S].mem_phys - p->p_map[D].mem_phys + p->p_map[S].mem_len;
  c1 += m_ptr->m1_i2;
  nc -= m_ptr->m1_i2;
  zeroclicks(c1, nc);
  m_ptr->m1_i1 = (int)base;
  m_ptr->m1_i2 = (int)size;
  return(OK);
}
#endif /* (SHADOWING == 1) */
```

```
/*===========================================================================*
 *       do_sendsig      *
 *===========================================================================*/
PRIVATE int do_sendsig(m_ptr)
message *m_ptr; /* pointer to request message */
{
/* Handle sys_sendsig, POSIX-style signal */

  struct sigmsg smsg;
  register struct proc *rp;
  phys_bytes src_phys, dst_phys;
  struct sigcontext sc, *scp;
  struct sigframe fr, *frp;

  if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
  rp = proc_addr(m_ptr->PROC1);

  /* Get the sigmsg structure into our address space.  */
  src_phys = umap(proc_addr(MM_PROC_NR), D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
  (vir_bytes) sizeof(struct sigmsg));
  if (src_phys == 0)
panic("do_sendsig can't signal: bad sigmsg address from MM", NO_NUM);
  phys_copy(src_phys, vir2phys(&smsg), (phys_bytes) sizeof(struct sigmsg));

  /* Compute the usr stack pointer value where sigcontext will be stored. */
  scp = (struct sigcontext *) smsg.sm_stkptr - 1;

  /* Copy the registers to the sigcontext structure. */
  memcpy(&sc.sc_regs, &rp->p_reg, sizeof(struct sigregs));

  /* Finish the sigcontext initialization. */
  sc.sc_flags = SC_SIGCONTEXT;

  sc.sc_mask = smsg.sm_mask;

  /* Copy the sigcontext structure to the user's stack. */
  dst_phys = umap(rp, D, (vir_bytes) scp,
  (vir_bytes) sizeof(struct sigcontext));
  if (dst_phys == 0) return(EFAULT);
  phys_copy(vir2phys(&sc), dst_phys, (phys_bytes) sizeof(struct sigcontext));

  /* Initialize the sigframe structure. */
  frp = (struct sigframe *) scp - 1;
  fr.sf_scpcopy = scp;
  fr.sf_retadr2= (void (*)()) rp->p_reg.pc;
  fr.sf_fp = rp->p_reg.fp;
  rp->p_reg.fp = (reg_t) &frp->sf_fp;
```

```
  fr.sf_scp = scp;
  fr.sf_code = 0; /* XXX - should be used for type of FP exception */
  fr.sf_signo = smsg.sm_signo;
  fr.sf_retadr = (void (*)()) smsg.sm_sigreturn;

  /* Copy the sigframe structure to the user's stack. */
  dst_phys = umap(rp, D, (vir_bytes) frp, (vir_bytes) sizeof(struct sigframe));
  if (dst_phys == 0) return(EFAULT);
  phys_copy(vir2phys(&fr), dst_phys, (phys_bytes) sizeof(struct sigframe));

  /* Reset user registers to execute the signal handler. */
  rp->p_reg.sp = (reg_t) frp;
  rp->p_reg.pc = (reg_t) smsg.sm_sighandler;

  return(OK);
}

/*===========================================================================*
 *       do_sigreturn      *
 *===========================================================================*/
PRIVATE int do_sigreturn(m_ptr)
register message *m_ptr;
{
/* POSIX style signals require sys_sigreturn to put things in order before the
 * signalled process can resume execution
 */

  struct sigcontext sc;
  register struct proc *rp;
  phys_bytes src_phys;

  if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
  rp = proc_addr(m_ptr->PROC1);

  /* Copy in the sigcontext structure. */
  src_phys = umap(rp, D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
  (vir_bytes) sizeof(struct sigcontext));
  if (src_phys == 0) return(EFAULT);
  phys_copy(src_phys, vir2phys(&sc), (phys_bytes) sizeof(struct sigcontext));

  /* Make sure that this is not just a jmp_buf. */
  if ((sc.sc_flags & SC_SIGCONTEXT) == 0) return(EINVAL);

  /* Fix up only certain key registers if the compiler doesn't use
   * register variables within functions containing setjmp.
   */
  if (sc.sc_flags & SC_NOREGLOCALS) {
rp->p_reg.retreg = sc.sc_retreg;
rp->p_reg.fp = sc.sc_fp;
```

```
rp->p_reg.pc = sc.sc_pc;
rp->p_reg.sp = sc.sc_sp;
return (OK);
  }
  sc.sc_psw  = rp->p_reg.psw;

#if (CHIP == INTEL)
  /* Don't panic kernel if user gave bad selectors. */
  sc.sc_cs = rp->p_reg.cs;
  sc.sc_ds = rp->p_reg.ds;
  sc.sc_es = rp->p_reg.es;
#if _WORD_SIZE == 4
  sc.sc_fs = rp->p_reg.fs;
  sc.sc_gs = rp->p_reg.gs;
#endif
#endif

  /* Restore the registers. */
  memcpy(&rp->p_reg, (char *)&sc.sc_regs, sizeof(struct sigregs));

  return(OK);
}

/*===========================================================================*
 * do_kill      *
 *===========================================================================*/
PRIVATE int do_kill(m_ptr)
register message *m_ptr; /* pointer to request message */
{
/* Handle sys_kill(). Cause a signal to be sent to a process via MM.
 * Note that this has nothing to do with the kill (2) system call, this
 * is how the FS (and possibly other servers) get access to cause_sig to
 * send a KSIG message to MM
 */

  if (!isokusern(m_ptr->PR)) return(E_BAD_PROC);
  cause_sig(m_ptr->PR, m_ptr->SIGNUM);
  return(OK);
}


/*===========================================================================*
 *       do_endsig      *
 *===========================================================================*/
PRIVATE int do_endsig(m_ptr)
register message *m_ptr; /* pointer to request message */
{
/* Finish up after a KSIG-type signal, caused by a SYS_KILL message or a call
 * to cause_sig by a task
```

```
 */

  register struct proc *rp;

  if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
  rp = proc_addr(m_ptr->PROC1);

  /* MM has finished one KSIG. */
  if (rp->p_pendcount != 0 && --rp->p_pendcount == 0
      && (rp->p_flags &= ~SIG_PENDING) == 0)
lock_ready(rp);
  return(OK);
}

/*===========================================================================*
 * do_copy      *
 *===========================================================================*/
PRIVATE int do_copy(m_ptr)
register message *m_ptr; /* pointer to request message */
{
/* Handle sys_copy().  Copy data for MM or FS. */

  int src_proc, dst_proc, src_space, dst_space;
  vir_bytes src_vir, dst_vir;
  phys_bytes src_phys, dst_phys, bytes;

  /* Dismember the command message. */
  src_proc = m_ptr->SRC_PROC_NR;
  dst_proc = m_ptr->DST_PROC_NR;
  src_space = m_ptr->SRC_SPACE;
  dst_space = m_ptr->DST_SPACE;
  src_vir = (vir_bytes) m_ptr->SRC_BUFFER;
  dst_vir = (vir_bytes) m_ptr->DST_BUFFER;
  bytes = (phys_bytes) m_ptr->COPY_BYTES;

  /* Compute the source and destination addresses and do the copy. */
#if (SHADOWING == 0)
  if (src_proc == ABS)
src_phys = (phys_bytes) m_ptr->SRC_BUFFER;
  else {
if (bytes != (vir_bytes) bytes)
/* This would happen for 64K segments and 16-bit vir_bytes.
 * It would happen a lot for do_fork except MM uses ABS
 * copies for that case.
 */
panic("overflow in count in do_copy", NO_NUM);
#endif

src_phys = umap(proc_addr(src_proc), src_space, src_vir,
```

```
(vir_bytes) bytes);
#if (SHADOWING == 0)
}
#endif

#if (SHADOWING == 0)
  if (dst_proc == ABS)
dst_phys = (phys_bytes) m_ptr->DST_BUFFER;
  else
#endif
dst_phys = umap(proc_addr(dst_proc), dst_space, dst_vir,
(vir_bytes) bytes);

  if (src_phys == 0 || dst_phys == 0) return(EFAULT);
  phys_copy(src_phys, dst_phys, bytes);
  return(OK);
}


/*===========================================================================*
 * do_vcopy      *
 *===========================================================================*/
PRIVATE int do_vcopy(m_ptr)
register message *m_ptr; /* pointer to request message */
{
/* Handle sys_vcopy(). Copy multiple blocks of memory */

  int src_proc, dst_proc, vect_s, i;
  vir_bytes src_vir, dst_vir, vect_addr;
  phys_bytes src_phys, dst_phys, bytes;
  cpvec_t cpvec_table[CPVEC_NR];

  /* Dismember the command message. */
  src_proc = m_ptr->m1_i1;
  dst_proc = m_ptr->m1_i2;
  vect_s = m_ptr->m1_i3;
  vect_addr = (vir_bytes)m_ptr->m1_p1;

  if (vect_s > CPVEC_NR) return EDOM;

  src_phys= numap (m_ptr->m_source, vect_addr, vect_s * sizeof(cpvec_t));
  if (!src_phys) return EFAULT;
  phys_copy(src_phys, vir2phys(cpvec_table),
(phys_bytes) (vect_s * sizeof(cpvec_t)));

  for (i = 0; i < vect_s; i++) {
src_vir= cpvec_table[i].cpv_src;
dst_vir= cpvec_table[i].cpv_dst;
bytes= cpvec_table[i].cpv_size;
```

```
src_phys = numap(src_proc,src_vir,(vir_bytes)bytes);
dst_phys = numap(dst_proc,dst_vir,(vir_bytes)bytes);
if (src_phys == 0 || dst_phys == 0) return(EFAULT);
phys_copy(src_phys, dst_phys, bytes);
  }
  return(OK);
}


/*===========================================================================*
 * do_gboot     *
 *===========================================================================*/
PUBLIC struct bparam_s boot_parameters;

PRIVATE int do_gboot(m_ptr)
message *m_ptr; /* pointer to request message */
{
/* Copy the boot parameters.  Normally only called during fs init. */

  phys_bytes dst_phys;

  dst_phys = umap(proc_addr(m_ptr->PROC1), D, (vir_bytes) m_ptr->MEM_PTR,
(vir_bytes) sizeof(boot_parameters));
  if (dst_phys == 0) panic("bad call to SYS_GBOOT", NO_NUM);
  phys_copy(vir2phys(&boot_parameters), dst_phys,
(phys_bytes) sizeof(boot_parameters));
  return(OK);
}


/*===========================================================================*
 * do_mem      *
 *===========================================================================*/
PRIVATE int do_mem(m_ptr)
register message *m_ptr; /* pointer to request message */
{
/* Return the base and size of the next chunk of memory. */

  struct memory *memp;

  for (memp = mem; memp < &mem[NR_MEMS]; ++memp) {
m_ptr->m1_i1 = memp->base;
m_ptr->m1_i2 = memp->size;
m_ptr->m1_i3 = tot_mem_size;
memp->size = 0;
if (m_ptr->m1_i2 != 0) break; /* found a chunk */
  }
  return(OK);
}
```

```
/*===========================================================================*
 * do_umap     *
 *===========================================================================*/
PRIVATE int do_umap(m_ptr)
register message *m_ptr; /* pointer to request message */
{
/* Same as umap(), for non-kernel processes. */

  m_ptr->SRC_BUFFER = umap(proc_addr((int) m_ptr->SRC_PROC_NR),
                           (int) m_ptr->SRC_SPACE,
                           (vir_bytes) m_ptr->SRC_BUFFER,
                           (vir_bytes) m_ptr->COPY_BYTES);
  return(OK);
}


/*===========================================================================*
 * do_trace     *
 *===========================================================================*/
#define TR_PROCNR (m_ptr->m2_i1)
#define TR_REQUEST (m_ptr->m2_i2)
#define TR_ADDR ((vir_bytes) m_ptr->m2_l1)
#define TR_DATA (m_ptr->m2_l2)
#define TR_VLSIZE ((vir_bytes) sizeof(long))

PRIVATE int do_trace(m_ptr)
register message *m_ptr;
{
/* Handle the debugging commands supported by the ptrace system call
 * The commands are:
 * T_STOP stop the process
 * T_OK enable tracing by parent for this process
 * T_GETINS return value from instruction space
 * T_GETDATA return value from data space
 * T_GETUSER return value from user process table
 * T_SETINS set value from instruction space
 * T_SETDATA set value from data space
 * T_SETUSER set value in user process table
 * T_RESUME resume execution
 * T_EXIT exit
 * T_STEP set trace bit
 *
 * The T_OK and T_EXIT commands are handled completely by the memory manager,
 * all others come here.
 */

  register struct proc *rp;
```

```
  phys_bytes src, dst;
  int i;

  rp = proc_addr(TR_PROCNR);
  if (rp->p_flags & P_SLOT_FREE) return(EIO);
  switch (TR_REQUEST) {
  case T_STOP: /* stop process */
if (rp->p_flags == 0) lock_unready(rp);
rp->p_flags |= P_STOP;
rp->p_reg.psw &= ~TRACEBIT; /* clear trace bit */
return(OK);

  case T_GETINS: /* return value from instruction space */
if (rp->p_map[T].mem_len != 0) {
if ((src = umap(rp, T, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
phys_copy(src, vir2phys(&TR_DATA), (phys_bytes) sizeof(long));
break;
}
/* Text space is actually data space - fall through. */

  case T_GETDATA: /* return value from data space */
if ((src = umap(rp, D, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
phys_copy(src, vir2phys(&TR_DATA), (phys_bytes) sizeof(long));
break;

  case T_GETUSER: /* return value from process table */
if ((TR_ADDR & (sizeof(long) - 1)) != 0 ||
    TR_ADDR > sizeof(struct proc) - sizeof(long))
return(EIO);
TR_DATA = *(long *) ((char *) rp + (int) TR_ADDR);
break;

  case T_SETINS: /* set value in instruction space */
if (rp->p_map[T].mem_len != 0) {
if ((dst = umap(rp, T, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
phys_copy(vir2phys(&TR_DATA), dst, (phys_bytes) sizeof(long));
TR_DATA = 0;
break;
}
/* Text space is actually data space - fall through. */

  case T_SETDATA: /* set value in data space */
if ((dst = umap(rp, D, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
phys_copy(vir2phys(&TR_DATA), dst, (phys_bytes) sizeof(long));
TR_DATA = 0;
break;

  case T_SETUSER: /* set value in process table */
if ((TR_ADDR & (sizeof(reg_t) - 1)) != 0 ||
```

```
     TR_ADDR > sizeof(struct stackframe_s) - sizeof(reg_t))
return(EIO);
i = (int) TR_ADDR;
#if (CHIP == INTEL)
/* Altering segment registers might crash the kernel when it
 * tries to load them prior to restarting a process, so do
 * not allow it.
 */
if (i == (int) &((struct proc *) 0)->p_reg.cs ||
    i == (int) &((struct proc *) 0)->p_reg.ds ||
    i == (int) &((struct proc *) 0)->p_reg.es ||
#if _WORD_SIZE == 4
    i == (int) &((struct proc *) 0)->p_reg.gs ||
    i == (int) &((struct proc *) 0)->p_reg.fs ||
#endif
    i == (int) &((struct proc *) 0)->p_reg.ss)
return(EIO);
#endif
if (i == (int) &((struct proc *) 0)->p_reg.psw)
/* only selected bits are changeable */
SETPSW(rp, TR_DATA);
else
*(reg_t *) ((char *) &rp->p_reg + i) = (reg_t) TR_DATA;
TR_DATA = 0;
break;

  case T_RESUME: /* resume execution */
rp->p_flags &= ~P_STOP;
if (rp->p_flags == 0) lock_ready(rp);
TR_DATA = 0;
break;

  case T_STEP: /* set trace bit */
rp->p_reg.psw |= TRACEBIT;
rp->p_flags &= ~P_STOP;
if (rp->p_flags == 0) lock_ready(rp);
TR_DATA = 0;
break;

  default:
return(EIO);
  }
  return(OK);
}

/*===========================================================================*
 * cause_sig      *
 *===========================================================================*/
PUBLIC void cause_sig(proc_nr, sig_nr)
```

```
int proc_nr; /* process to be signalled */
int sig_nr; /* signal to be sent, 1 to _NSIG */
{
/* A task wants to send a signal to a process.   Examples of such tasks are:
 *   TTY wanting to cause SIGINT upon getting a DEL
 *   CLOCK wanting to cause SIGALRM when timer expires
 * FS also uses this to send a signal, via the SYS_KILL message.
 * Signals are handled by sending a message to MM.  The tasks don't dare do
 * that directly, for fear of what would happen if MM were busy.  Instead they
 * call cause_sig, which sets bits in p_pending, and then carefully checks to
 * see if MM is free.  If so, a message is sent to it.  If not, when it becomes
 * free, a message is sent.  The process being signaled is blocked while MM
 * has not seen or finished with all signals for it.  These signals are
 * counted in p_pendcount, and the SIG_PENDING flag is kept nonzero while
 * there are some.  It is not sufficient to ready the process when MM is
 * informed, because MM can block waiting for FS to do a core dump.
 */

  register struct proc *rp, *mmp;

  rp = proc_addr(proc_nr);
  if (sigismember(&rp->p_pending, sig_nr))
return; /* this signal already pending */
  sigaddset(&rp->p_pending, sig_nr);
  ++rp->p_pendcount; /* count new signal pending */
  if (rp->p_flags & PENDING)
return; /* another signal already pending */
  if (rp->p_flags == 0) lock_unready(rp);
  rp->p_flags |= PENDING | SIG_PENDING;
  ++sig_procs; /* count new process pending */

  mmp = proc_addr(MM_PROC_NR);
  if ( ((mmp->p_flags & RECEIVING) == 0) || mmp->p_getfrom != ANY) return;
  inform();
}


/*===========================================================================*
 * inform      *
 *===========================================================================*/
PUBLIC void inform()
{
/* When a signal is detected by the kernel (e.g., DEL), or generated by a task
 * (e.g. clock task for SIGALRM), cause_sig() is called to set a bit in the
 * p_pending field of the process to signal.  Then inform() is called to see
 * if MM is idle and can be told about it.  Whenever MM blocks, a check is
 * made to see if 'sig_procs' is nonzero; if so, inform() is called.
 */
```

```
  register struct proc *rp;

  /* MM is waiting for new input.  Find a process with pending signals. */
  for (rp = BEG_SERV_ADDR; rp < END_PROC_ADDR; rp++)
if (rp->p_flags & PENDING) {
m.m_type = KSIG;
m.SIG_PROC = proc_number(rp);
m.SIG_MAP = rp->p_pending;
sig_procs--;
if (lock_mini_send(proc_addr(HARDWARE), MM_PROC_NR, &m) != OK)
panic("can't inform MM", NO_NUM);
sigemptyset(&rp->p_pending); /* the ball is now in MM's court */
rp->p_flags &= ~PENDING;/* remains inhibited by SIG_PENDING */
lock_pick_proc(); /* avoid delay in scheduling MM */
return;
}
}


/*===========================================================================*
 * umap      *
 *===========================================================================*/
PUBLIC phys_bytes umap(rp, seg, vir_addr, bytes)
register struct proc *rp; /* pointer to proc table entry for process */
int seg; /* T, D, or S segment */
vir_bytes vir_addr; /* virtual address in bytes within the seg */
vir_bytes bytes; /* # of bytes to be copied */
{
/* Calculate the physical memory address for a given virtual address. */

  vir_clicks vc; /* the virtual address in clicks */
  phys_bytes pa; /* intermediate variables as phys_bytes */
#if (CHIP == INTEL)
  phys_bytes seg_base;
#endif

  /* If 'seg' is D it could really be S and vice versa.  T really means T.
   * If the virtual address falls in the gap,  it causes a problem. On the
   * 8088 it is probably a legal stack reference, since "stackfaults" are
   * not detected by the hardware.  On 8088s, the gap is called S and
   * accepted, but on other machines it is called D and rejected.
   * The Atari ST behaves like the 8088 in this respect.
   */

  if (bytes <= 0) return( (phys_bytes) 0);
  vc = (vir_addr + bytes - 1) >> CLICK_SHIFT; /* last click of data */

#if (CHIP == INTEL) || (CHIP == M68000)
  if (seg != T)
```

```
seg = (vc < rp->p_map[D].mem_vir + rp->p_map[D].mem_len ? D : S);
#else
  if (seg != T)
seg = (vc < rp->p_map[S].mem_vir ? D : S);
#endif

  if((vir_addr>>CLICK_SHIFT) >= rp->p_map[seg].mem_vir+ rp->p_map[seg].mem_len)
return( (phys_bytes) 0 );
#if (CHIP == INTEL)
  seg_base = (phys_bytes) rp->p_map[seg].mem_phys;
  seg_base = seg_base << CLICK_SHIFT; /* segment origin in bytes */
#endif
  pa = (phys_bytes) vir_addr;
#if (CHIP != M68000)
  pa -= rp->p_map[seg].mem_vir << CLICK_SHIFT;
  return(seg_base + pa);
#endif
#if (CHIP == M68000)
#if (SHADOWING == 0)
  pa -= (phys_bytes)rp->p_map[seg].mem_vir << CLICK_SHIFT;
  pa += (phys_bytes)rp->p_map[seg].mem_phys << CLICK_SHIFT;
#else
  if (rp->p_shadow && seg != T) {
pa -= (phys_bytes)rp->p_map[D].mem_phys << CLICK_SHIFT;
pa += (phys_bytes)rp->p_shadow << CLICK_SHIFT;
  }
#endif
  return(pa);
#endif
}


/*===========================================================================*
 * numap     *
 *===========================================================================*/
PUBLIC phys_bytes numap(proc_nr, vir_addr, bytes)
int proc_nr; /* process number to be mapped */
vir_bytes vir_addr; /* virtual address in bytes within D seg */
vir_bytes bytes; /* # of bytes required in segment  */
{
/* Do umap() starting from a process number instead of a pointer.  This
 * function is used by device drivers, so they need not know about the
 * process table.  To save time, there is no 'seg' parameter. The segment
 * is always D.
 */

  return(umap(proc_addr(proc_nr), D, vir_addr, bytes));
}
```

```
#if (CHIP == INTEL)
/*===========================================================================*
 * alloc_segments     *
 *===========================================================================*/
PUBLIC void alloc_segments(rp)
register struct proc *rp;
{
/* This is called only by do_newmap, but is broken out as a separate function
 * because so much is hardware-dependent.
 */

  phys_bytes code_bytes;
  phys_bytes data_bytes;
  int privilege;

  if (protected_mode) {
data_bytes = (phys_bytes) (rp->p_map[S].mem_vir + rp->p_map[S].mem_len)
             << CLICK_SHIFT;
if (rp->p_map[T].mem_len == 0)
code_bytes = data_bytes; /* common I&D, poor protect */
else
code_bytes = (phys_bytes) rp->p_map[T].mem_len << CLICK_SHIFT;
privilege = istaskp(rp) ? TASK_PRIVILEGE : USER_PRIVILEGE;
init_codeseg(&rp->p_ldt[CS_LDT_INDEX],
     (phys_bytes) rp->p_map[T].mem_phys << CLICK_SHIFT,
     code_bytes, privilege);
init_dataseg(&rp->p_ldt[DS_LDT_INDEX],
     (phys_bytes) rp->p_map[D].mem_phys << CLICK_SHIFT,
     data_bytes, privilege);
rp->p_reg.cs = (CS_LDT_INDEX * DESC_SIZE) | TI | privilege;
#if _WORD_SIZE == 4
rp->p_reg.gs =
rp->p_reg.fs =
#endif
rp->p_reg.ss =
rp->p_reg.es =
rp->p_reg.ds = (DS_LDT_INDEX*DESC_SIZE) | TI | privilege;
  } else {
rp->p_reg.cs = click_to_hclick(rp->p_map[T].mem_phys);
rp->p_reg.ss =
rp->p_reg.es =
rp->p_reg.ds = click_to_hclick(rp->p_map[D].mem_phys);
  }
}
#endif /* (CHIP == INTEL) */
```

sys_block.c

```
#include "syslib.h"

PUBLIC int sys_block(proc)
int proc; /* process to block */
{
  message m;

  m.m1_i1 = proc;
  return(_taskcall(SYSTASK, SYS_BLOCK, &m));
}
```

sys_unblock.c

```
#include "syslib.h"

PUBLIC int sys_unblock(proc)
int proc; /* process to unblock */
{
  message m;

  m.m1_i1 = proc;
  return(_taskcall(SYSTASK, SYS_UNBLOCK, &m));
}
```

v_sem.s

```
.sect .text
.extern __v_sem
.define _v_sem

.align 2

_v_sem:
jmp __v_sem
```

_sem.c

```
#include <lib.h>
#include <minix/semaforo.h>
#include <minix/com.h>
#include <minix/constsemaf.h>
#include <stdio.h>
#include <string.h>

#define crear_sem _crear_sem
#define p_sem _p_sem
#define v_sem  _v_sem
#define liberar_sem _liberar_sem
```

```
#define inicializar _inicializar

semaforo crear_sem(char* nombre, int valor) {

message m;
semaforo new_sem;

strcpy(m.NOMBRE_SEM,"");

if(strlen(nombre) < M3_STRING) {
strcpy(m.NOMBRE_SEM, nombre);
} else {
strncpy(m.NOMBRE_SEM, nombre, M3_STRING-1);
}

m.VALOR = valor;

new_sem = _syscall(MM, CREAR_SEM, &m);
if(new_sem==-1) {
printf("error: crear_sem\n");
return ERROR;
} else {
return new_sem;
}

}

int p_sem(semaforo x) {

message m;
int r;

m.SEMAFORO = x;

r = _syscall(MM, P_SEM, &m);

if(r==-1) {
printf("error: p_sem\n");
return ERROR;
} else {
return r;
}

}

int v_sem(semaforo x) {

message m;
int r;
```

```
m.SEMAFORO = x;

r = _syscall(MM, V_SEM, &m);

if(r==-1) {
printf("error: v_sem\n");
return ERROR;
} else {
return r;
}

}

int liberar_sem(semaforo x) {

message m;
int r;

m.SEMAFORO = x;

r = _syscall(MM, LIBERAR_SEM, &m);

if(r==-1) {
printf("error: liberar_sem");
return ERROR;
} else {
return r;
}

}

void inicializar() {

message m;

_syscall(MM, INIT_ALL_SEM, &m);

return 0;

}
```

3. Referencias

- Sistemas Operativos - Diseño e Implementación (Andrew Tanenbaum - Prentice Hall 1998)
- http://www.minix3.org/
- http://es.wikipedia.org/wiki/Chmod