Algoritmos y Estructuras de Datos III

Primer Cuatrimestre de 2007

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

Trabajo Práctico 2

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, Memoria, Perifericos, Drivers

Índice

1. Soluciones

1.1. Ejercicio 1

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** están todos los archivos necesarios 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**.

```
248495051524849505152484950515248495051524849505152484950515248495051524849505152484950515
248495051524849505152484950515248495051524849505152484950515248495051524849505152484950515
248495051524849505152484950515248495051524849505152484950515248495051524849505152484950515
248495051524849505152484950515248495051524849505152484950515248495051524849505152484950515
2484950515248495051524 8Terminated
```

(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

1.8. 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 increment en 1 el define NCALLS Se agreg `#define LLAMSISTEMA` con el nro correspondiente

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

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

El 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`

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 agreg 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.

1.12. Ejercicio 12

2. Apéndice A

A continuación se muestra el código de la función *do_clocktick* que no se ejecutará si está definido UBA_FCEN. Dicha función se encuentra en el archivo **/usr/src/kernel/clock.c**.

```
[...]
#ifndef UBA_FCEN
  /* If a user process has been running too long, pick another one. */

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

Aquí se verá como fue modificada la función *alloc_mem*, la cual se encuentra en el archivo **/usr/src/mm/alloc.c**.

```
[...]
/*===========================================================================*
* alloc_mem *
*===========================================================================*/
#ifdef UBA_FCEN

PUBLIC phys_clicks alloc_mem(clicks)
phys_clicks clicks;
{
  register struct hole *max, *anterior;
  phys_clicks old_base;
  register struct hole *actual, *debugg;
  actual = hole_head;
  max = hole_head;

#ifdef DEBUGG

  /*DEBUGER: para ver como es el estado inicial de la memoria*/
  debugg = hole_head;
  printf("Se quiere reservar %d espacios de memoria \n",clicks);
  while (debugg != NIL_HOLE)
  {
    printf("%d ",debugg->h_len);
    debugg = debugg->h_next;
  }
  printf("\n");
  /************************************************************/

#endif
```

```
  while (actual->h_next != NIL_HOLE)
  {
    if (actual->h_next->h_len > max->h_len)
    {
      max = actual->h_next;
      anterior = actual;
    }
    actual = actual->h_next;
  }

  if (max->h_len < clicks)
  }
    return(NO_MEM);
  }
  else
  {
    old_base = max->h_base;
    max->h_base += clicks;
    max->h_len -= clicks;
  }

  if (max->h_len == 0)
  {
    del_slot(anterior, max);
  }

#ifdef DEBUGG

  /*DEBUG: para ver como queda la memoria una vez hecha la alicacion*/
  debugg = hole_head;
  printf("Los segmentos de memoria quedaron de la siguiente manera \n");
  while (debugg != NIL_HOLE)
  {
    printf("%d ",debugg->h_len);
    debugg = debugg->h_next;
  }
  printf("\n");
  /*****************************************************************/

#endif

  return(old_base);
}

#else

PUBLIC phys_clicks alloc_mem(clicks)
phys_clicks clicks;             /* amount of memory requested */
```

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

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

  hp = hole_head;
  while (hp != NIL_HOLE) {
        if (hp->h_len >= clicks) {
            /* We found a hole that is big enough.  Use it. */
            old_base = hp->h_base;  /* remember where it started */
            hp->h_base += clicks;   /* bite a piece off */
            hp->h_len -= clicks;    /* ditto */

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

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

        prev_ptr = hp;
        hp = hp->h_next;
  }
  return(NO_MEM);
}

#endif
[...]
```

3. Referencias

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