Lab Unix

El objetivo de este lab es tener un primer contacto con la interfaz del kernel mediante syscalls tradicionales de Unix, en particular las relacionadas al manejo de archivos.

El lab consiste en la implementación de versiones simplificadas de las herramientas Unix más comunes provistas en cualquier distribución: *ls*, *cat*, *rm*, *cp*, *mv*, etc.

Syscalls UNIX

Se debe implementar cada ejercicio usando las syscalls Unix apropiadas, evitando el uso de las "funciones de alto nivel" que proporciona la biblioteca estándar de C. Así, por ejemplo, para la apertura de archivos se debe usar la syscall `open(2)`, y no la función `fopen(3)`.

Sí se permite el uso de funciones de la biblioteca estándar para trabajar con strings y para mostrar información por pantalla. Así, por ejemplo, para escribir en la consola se puede usar `printf(3)` en lugar de `write(2)`.

Tanto en el caso de syscalls, como funciones, se puede consultar su documentación mediante el comando `man`. Esto es particularmente recomendable en el caso de syscalls como `stat(2)`, que son complejas y tienen muchos flags: `man 2 stat`. En las páginas de manual también se indican los includes necesarios para cada syscall.

En cada ejercicio se indica la lista de syscalls recomendadas. Como cada ejercicio emula una herramienta estándar de Unix, se puede obtener una descripción de la funcionalidad completa también en las páginas del man (e.g. `man 1 cat`).

Bibliografía

Una buena referencia sobre sistemas Unix/POSIX es **KERR**. En particular para este lab:

- **cap. 4**: *File I/O: The Universal I/O Model*
- **cap. 15**: *File Attributes*
- **cap. 18**: *Directories and Links*
- **cap. 49**: *Memory Mappings*

Opcionalmente se puede leer el capítulo 3 a modo de introducción.

Esqueleto y flags de compilación

El siguiente esqueleto de un comando que acepta un único parámetro puede usarse a modo de ejemplo para cualquiera de las implementaciones:

```
#define _POSIX_C_SOURCE 200809L

#include <...>

void rm0(const char *file) {
    // ...
}

int main(int argc, char *argv[]) {
    rm0(argv[1]);
}
```

Se recomienda compilar utilizando los flags `-std=c11 -Wall -Wextra -g`.

Parte 1

rm0 ☆

rm (*remove*) es la herramienta unix que permite eliminar archivos y directorios.

El uso estándar `rm <file>` permite borrar solo archivos regulares, y arrojará error si se intenta eliminar un directorio.

Para la implementación de *rm0* solo se considerará el caso de archivos regulares.

```
$ ls
archivo1   archivo2   directorio1   rm0
$ ./rm0 archivo1
$ ls
archivo2   directorio1   rm0
```

Se pide: implementar *rm0* que elimina un archivo regular.

Pre-condición: el archivo existe y es regular.

Syscalls recomendadas: unlink.

cat0 ☆

cat (*concatenate*) es una herramienta unix que permite concatenar archivos y mostrarlos por salida estándar. En este lab se implementará una versión simplificada de cat, que muestra en pantalla los contenidos de un único archivo.

```
$ cat ejemplo.txt
Sistemas Operativos, 1er cuatrimestre 2018
```

Se pide: Implementar *cat0* que toma un archivo regular y muestra su contenido por salida estándar.

Pre-condición: solo se pasa un archivo, este archivo existe y se tienen permisos de lectura.

Syscalls recomendadas: open, read, write, close.

touch0 ☆

touch toma como parámetro un archivo (de cualquier tipo) y permite actualizar su metadata, especialmente las fechas de último acceso (*atime*) y última modificación (*mtime*); ambos atributos pueden verse mediante el comando *stat*. Una llamada a touch sobre un archivo actualiza ambas fechas al tiempo actual.

No obstante, el uso más común del comando touch es la creación de archivos regulares: si el parámetro referencia a un archivo que no existe, se lo crea. La primera versión *touch0* sólo implementará esta funcionalidad de touch.

Se pide: Implementar *touch0* que toma como parámetro un archivo y lo crea en caso de que no exista (el archivo creado debe estar en blanco). Si el archivo ya existía, no se hace nada.

Ejemplo:

```
$ ls
touch0
$ ./touch0 un_archivo
$ ls
touch0 un_archivo
$ stat un_archivo
  File: un_archivo
  Size: 0     Blocks: 0    IO Block: 4096   regular empty file
[...]
```

Notar que el tamaño del archivo creado es 0, y stat también nos lo indica enunciando *regular empty file*.

Pre-condición: si el archivo existe, es un archivo regular.

Syscalls recomendadas: open.

stat0 ☆☆

stat muestra en pantalla los metadatos de un archivo, incluyendo información sobre tipo de archivo, fechas de creación y modificación, permisos, etc.

```
$ stat README.md
  File: README.md
  Size: 1318        Blocks: 8          IO Block: 4096   regular file
Device: 806h/2054d  Inode: 2753812     Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/    juan)   Gid: ( 1000/    juan)
Access: 2018-03-14 17:36:37.497432618 -0300
Modify: 2018-03-08 23:27:15.765147109 -0300
Change: 2018-03-08 23:27:15.765147109 -0300
 Birth: -
```

La implementación de *stat0* mostrará únicamente el nombre, tipo y tamaño del archivo (en bytes).

```
$ ./stat0 README.md
Size: 1318
File: README.md
Type: regular file
```

Se pide: Implementar *stat0* que muestra el nombre, tipo y tamaño en bytes de un archivo regular o directorio.

Pre-condición: el archivo existe, y es un directorio o un archivo regular.

Syscalls recomendadas: stat. Se puede consultar también la página de manual `inode(7)`.

rm1 ★

Mostrar cómo se usaría `errno` y `perror(3)` para obtener el siguiente comportamiento de *rm*:

```
$ ./rm1 directorio1
rm: cannot remove 'directorio1': Is a directory
```

find

Se pide escribir una versión muy simplificada de la utilidad `find(1)`. Esta herramienta, tal y como se la encuentra en sistemas GNU/Linux, acepta una miríada de opciones (ver su página de manual, o un resumen gráfico). No obstante, en este lab se implementará sólo una de ellas.

La sinopsis de nuestra implementación será:

```
$ ./find [-i] <cadena>
```

Invocado como `./find xyz`, el programa buscará y mostrará por pantalla todos los archivos del directorio actual (y subdirectorios) cuyo nombre contenga (o sea igual a) `xyz`. Si se invoca como `./find -i xyz`, se realizará la misma búsqueda, pero sin distinguir entre mayúsculas y minúsculas.

Por ejemplo, si en el directorio actual se tiene:

```
.
├── Makefile
├── find.c
├── xargs.c
├── antiguo
│   ├── find.c
│   ├── xargs.c
│   ├── pingpong.c
│   ├── basurarghh
│   │   ├── find0.c
│   │   ├── find1.c
│   │   ├── pongg.c
│   │   └── findddddddd.c
│   ├── planes.txt
│   └── pingpong2.c
├── antinoo.jpg
└── GNUmakefile
```

el resultado de las distintas invocaciones debe ser como sigue (**no importa el orden** en que se impriman los archivos de un mismo directorio):

```
$ ./find akefile
Makefile
GNUmakefile

$ ./find Makefile
Makefile

$ ./find -i Makefile
Makefile
GNUmakefile

$ ./find arg
xargs.c
antiguo/xargs.c
antiguo/basurarghh

$ ./find pong
antiguo/pingpong.c
antiguo/basurarghh/pongg.c
antiguo/pingpong2.c

$ ./find an
antiguo
antiguo/planes.txt
antinoo.jpg

$ ./find d.c
find.c
antiguo/find.c
antiguo/basurarghh/findddddddd.c
```

Ayuda:

- Usar recursividad para descender a los distintos directorios.
- Nunca descender los directorios especiales `.` y `..` (ambos son un “alias”; el primero al directorio actual, el segundo a su directorio inmediatamente superior).
- No es necesario preocuparse por ciclos en enlaces simbólicos.
- En el resultado de `readdir(3)`, asumir que el campo `d_type` siempre está presente, y es válido.
- La implementación *case-sensitive* vs. *case-insensitive* (opción `-i`) se puede resolver limpiamente usando un puntero a función como abstracción. (Ver `strstr(3)`.)

Requisitos:

- Llamar a la función `opendir(3)` **una sola vez**, al principio del programa (con argumento `"."`; no es necesario conseguir el *nombre* del directorio actual, si tenemos su alias).
- Para abrir sub-directorios, usar exclusivamente la función `openat(2)` (con el flag `O_DIRECTORY` como precaución). De esta manera, no es necesario realizar concatenación de cadenas para abrir subdirectorios.
  - Sí será necesario, no obstante, concatenar cadenas para mostrar por pantalla los resultados. No es necesario usar memoria dinámica; es suficiente un único buffer estático de longitud `PATH_MAX` (ver *header* `limits.h`).
  - Funciones que resultarán útiles como complemento a `openat()`: `dirfd(3)`, `fdopendir(3)`.

Llamadas al sistema: `openat(2)`, `readdir(3)`.

Parte 2

ln0 ☆

ln (*link*) permite la creación de enlaces a archivos, tanto “hard links” como “soft links”. Por defecto el uso de *ln* toma dos parámetros, el objetivo del link y el nombre; y crea un hard link. Puede usarse el flag `-s` (o –*symbolic*) para crear enlaces simbólicos (“soft links”).

La implementación de *ln0* replicará el comportamiento de `ln -s`, permitiendo crear

solamente enlaces simbólicos. Un uso de *ln0* podría ser:

```
$ ls -al
[...]
-rw-r--r-- 1 juan juan    0 Mar 21 18:39 archivo1
-rw-r--r-- 1 juan juan    0 Mar 21 18:39 ln0

$ ./ln0 archivo1 enlace

$ ls -al
-rw-r--r-- 1 juan juan    0 Mar 21 18:39 archivo1
lrwxrwxrwx 1 juan juan    8 Mar 21 18:40 enlace -> archivo1
-rw-r--r-- 1 juan juan    0 Mar 21 18:39 ln0
```

Se pide: Implementar *ln0* que permite crear enlaces simbólicos.

Pre-condición: no existe un archivo con el nombre del enlace.

Syscalls recomendadas: symlink.

Pregunta: ¿Qué ocurre si se intenta crear un enlace a un archivo que no existe?

mv0 ☆

mv (*move*) permite mover un archivo (regular o directorio) de un directorio a otro. El archivo no se mueve físicamente sino que sólo se renombra y se modifica el enlace al mismo en su directorio actual. La implementación de *mv0* tendrá la misma funcionalidad que *mv*.

Un ejemplo del uso de *mv0* podría ser:

```
$ ls
directorio1   archivo1
$ ls directorio1
$ mv archivo1 directorio1/archivo2
$ ls
directorio1
$ ls directorio1
archivo2
```

Se pide: Implementar *mv0*, que permite mover un archivo de un directorio a otro.

Pre-condición: el archivo destino no existe.

Syscalls recomendadas: rename.

Pregunta: ¿se puede usar *mv0* para renombrar archivos dentro del mismo directorio?

cp0 ☆☆

cp (*copy*) es el comando de Unix que permite copiar archivos. La sintaxis toma como parámetros dos archivos regulares, y copia los contenidos del primero al segundo. Si el segundo archivo no existe, es creado; y si ya existía, sus contenidos se sobreescriben.

El uso de *cp* sin flags adicionales sólo permite copiar archivos regulares, aunque puede especificarse el flag `-r` para copiar directorios de manera recursiva. La implementación de *cp0* sólo tendrá en cuenta el caso de copiar archivos regulares.

Ejemplo:

```
$ cat archivo1
Sistemas Operativos, 1er cuatrimestre 2018
$ ls
archivo1 cp0
$ ./cp0 archivo1 archivo2
$ ls
archivo1 archivo2 cp0
$ cat archivo2
Sistemas Operativos, 1er cuatrimestre 2018
```

Se pide: Implementar *cp0* que copia los contenidos de un archivo a otro. Utilizar para esta implementación las syscalls básicas de entrada y salida, esto es: open(2), read(2), write(2) y close(2).

Pre-condición: el archivo de origen existe y es regular. El archivo destino no existe.

touch1 ★

Revisitar la implementación de *touch0* realizada en los puntos anteriores y agregarle la funcionalidad de actualización de las fechas en la metadata de un archivo en caso de que ya exista. Tal metadata puede verse mediante el comando *stat*.

Un ejemplo del uso de *touch1* sería:

```
$ stat archivo
[...]
Access: 2018-03-16 17:31:48.722017895 -0300
Modify: 2018-03-14 17:27:56.438147960 -0300
Change: 2018-03-14 17:27:56.438147960 -0300
[...]
$ ./touch0 archivo
$ stat archivo
[...]
Access: 2018-03-21 00:58:04.671902112 -0300
Modify: 2018-03-21 00:58:04.671902112 -0300
Change: 2018-03-21 00:58:04.671902112 -0300
[...]
```

Se actualizaron todas las fechas asociadas al archivo, pero los contenidos del mismo no se vieron modificados, sólo se alteró la metadata.

Implementar *touch1* que toma un archivo como parámetro. Si no existe, crea un archivo regular vacío. Si el archivo existía previamente, modifica las fechas de acceso y de modificación al tiempo actual.

Syscalls recomendadas: utime(2) y la función futimes(3).

ln1 ★

Implementar una versión *ln1* que cree hard links en lugar de soft links. Hacer uso de la syscall *link(2)*. Luego, teniendo las dos versiones, responder las siguientes preguntas:

¿Cuál es la diferencia entre un hard link y un soft link?

Crear un hard link a un archivo, luego eliminar el archivo original ¿Qué pasa con el enlace? ¿Se perdieron los datos del archivo?

Repetir lo mismo, pero con un soft link. ¿Qué pasa ahora con el enlace? ¿Se perdieron los datos esta vez?

Explicar las diferencias.

Syscalls recomendadas: link(2), symlink(2)

Parte 3

tee0 ☆☆

tee (conector T) toma como parámetro un archivo, y escribe todo lo que llega por entrada estándar, tanto en la salida estándar como al archivo. Resulta muy útil cuando se quiere ver el resultado de la ejecución de un programa y a su vez guardar una copia de todo lo que escriba en un archivo.

Un ejemplo del uso de `tee` podría ser:

```
$ echo "Hola" | tee dump.txt
Hola
$ cat dump.txt
Hola
```

Por defecto `tee` crea el archivo si no lo encuentra, y lo sobreescribe (trunca) si ya existía. La implementación estándar de tee tiene muchas más opciones que pueden consultarse en el man (*tee(1)*).

Implementar *tee0* que transcribe la entrada estándar tanto en la salida estándar como en el

archivo especificado.

Pre-condición: el archivo o bien no existe, o bien es un archivo regular.

ls0 ☆☆☆

ls (*list*) lista los contenidos del directorio que se le pase por parámetro. Si no se especifica ningún parámetro, ls muestra el contenido de los archivos en el directorio actual (ver *pwd(1)*).

El comando ls admite una gran variedad de flags para elegir qué información se mostrará de los archivos, con qué formato y orden. La implementación de ls0 se corresponderá con `ls -U1`, o lo que es equivalente `ls --format=single-column --sort=none`, que lista únicamente los nombres de los archivos, sin ningún ordenamiento particular y de a uno por línea.

Por ejemplo:

```
$ ls
archivo1   archivo2   archivo3   ls0
$ ./ls0
archivo2
archivo1
ls0
archivo3
```

Se pide: Implementar *ls0* que lista todos los archivos en el directorio actual, uno en cada línea. No hay que preocuparse por el orden en que se listen los archivos, con que se muestren todos es suficiente.

Funciones recomendadas: stat(2), opendir(3), readdir(3), closedir(3).

cp1 ☆☆☆

La syscall *mmap* permite mapear una región de los contenidos de un archivo a memoria, y acceder a los mismos directamente como si fuera un array de bytes. Si se utilizan los flags apropiados (`MAP_SHARED`, ver mmap(2)) los cambios en la memoria correspondiente al archivo se verán reflejados en el mismo.

Si bien el uso de las syscall de entrada y salida básicas es la implementación más común para *cp*, también es posible utilizar *mmap* para copiar archivos. La idea es crear un archivo nuevo, y mapear tanto el archivo origen como el destino a *regiones de memoria distintas*, luego copiar los datos en memoria de una región a la otra (por ejemplo, utilizando *memcpy*).

Implementar *cp1*, que debe tener la misma funcionalidad que *cp0* pero implementada mediante *mmap* y *memcpy*.

Syscalls recomendadas: mmap(2), memcpy(2), open(2)

ps0 ★★

ps (*proccess status*) es un comando unix que permite obtener todo tipo de información acerca de los procesos que están corriendo actualmente, disponiendo de muchos flags que alteran la cantidad de información a mostrar. Ver *ps(1)* para la lista completa de flags.

Toda esta información se obtiene del pseudo-filesystem `/proc`, que mantiene acceso de sólo lectura a muchas estructuras de control del kernel relacionadas con procesos. En particular, los datos de cada proceso se encuentran en el subdirectorio `/proc/[pid]`, siendo *pid* el proccess ID del proceso.

Dentro de `/proc/[pid]` hay información exhaustiva sobre cada proceso. Para este ejercicio nos interesa en particular `/proc/[pid]/comm`, que guarda el nombre del programa que se usó para lanzar el proceso. Para tener una descripción exacta de qué guarda cada archivo en `/proc` y cómo está codificado, referirse a *proc(5)*.

La implementación de *ps0* (mucho más humilde), sólo listará para cada proceso su *pid* y el nombre del binario ejecutable que está corriendo. Para lograrlo hay que recorrer el directorio `/proc` y recaudar la información importante.

La salida de *ps0* equivale a ejecutar `ps -eo pid,comm`, que lista en dos columnas el process id y el comando de todos los procesos. Un ejemplo de esta salida sería:

```
$ ps -eo pid,comm
    1 systemd
    2 kthreadd
    3 ksoftirqd/0
    5 kworker/0:0H
    7 rcu_sched
    8 rcu_bh
[...]
 7531 bash
 7625 kworker/1:0
 8046 ps
```

Implementar *ps0* que debe mostrar la misma información que `ps -eo pid,comm`.

Syscalls recomendadas: opendir(3), readdir(3)

Ayudas: proc(5), isdigit(3), para corroborar que se esté accediendo al directorio de un proceso y no a algún otro archivo de `/proc`.

Challenge del challenge: dar más información del estado de un proceso a través de `/proc/[pid]/stat`, tomar de *proc(5)* el formato del archivo y ayudarse de *scanf(3)* para realizar el parseo.