75.08 Sistemas Operativos

Lab Unix - Parte 2

Nombre y Apellido: Damián Cassinotti
Padrón: 96618
Fecha de Entrega: 06/04/2018

Índice

1. Comandos obligatorios

En esta segunda parte del Lab se implementarán más versiones simplificadas de comandos típicos de Unix. Por cada comando pedido se indicará un resumen de lo pedido y luego el código fuente utilizado para la solución. En caso de haber preguntas en el enunciado, las mismas serán respondidas luego del código.

1.1. ln0

Para el comando ln se pidió una implementación en la cual se puedan crear soft links a archivos.
A continuacón se mostrará el código de los archivos utilizados.

```c
#define _POSIX_C_SOURCE 200809L
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main(int argc, char *argv[]);
void ln0(const char* file, const char* link);
```

ln0.h

```c
#include "ln0.h"

int main(int argc, char *argv[]) {
    if (argc != 3)
        return -1;
    ln0(argv[1], argv[2]);
    return 0;
}

void ln0(const char* file, const char* link) {
    symlink(file, link);
}
```

ln0.c

¿Qué ocurre si se intenta crear un enlace a un archivo que no existe?]
En caso de querer crear un soft link sobre un archivo inexistente se creará un dangling link (enlace colgante). Dicho enlace apuntará "hacia la nada", pues su archivo destino no existe. Si lo intentamos abrir desde la terminal mediante el comando cat, el mismo dará un error indicando que el archivo no existe.

1.2. mv0

En el caso del comando mv, se pidió una implementación que mueva archivos regulares o directorios a otros. Para esto se debe renombrar el enlace a dicho archivo (regular o directorio).
A continuacón se mostrará el código de los archivos utilizados.

```h
#define _POSIX_C_SOURCE 200809L
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main(int argc, char *argv[]);
void mv0(const char* file, const char* link);
int fileExists(const char* file);
```

mv0.h

```c
#include "mv0.h"

int main(int argc, char *argv[]) {
    if (argc != 3)
        return -1;
    if (!fileExists(argv[1])){
        char errMsg[strlen(argv[1]) + 18]; // mensaje + directorio
        sprintf(errMsg, "mv: cannot stat '%s'", argv[1]);
        perror(errMsg);
        return -1;
    }
    mv0(argv[1], argv[2]);
    return 0;
}

void mv0(const char* oldPath, const char* newPath) {
    rename(oldPath, newPath);
}

int fileExists(const char* file) {
    struct stat buf;
    int result = stat(file, &buf);
    if (result < 0)
        return 0;
    return 1;
}
```

mv0.c

¿Se puede usar mv0 para renombrar archivos dentro del mismo directorio?
Sí, se puede usar mv para renombrar archivos. Como se usa el rename, se pueden renombrar el archivo manteniendo el path intacto.

1.3. cp0

Nuestra implementación del comando cp sólo tendrá en cuenta la copia de arhcivos regulares. Si el archivo destino existe se reemplaza; caso contrario, el archivo se crea.
A continuacón se mostrará el código de los archivos utilizados.

```c
#define _POSIX_C_SOURCE 200809L
// open
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
// read
#include <unistd.h>

#define READ_BYTES 10

int main(int argc, char* argv[]);
void cp0(const char* input, const char* output);
// Return read-only file descriptor
int openForReading(const char *file);
// Return write-only file descriptor
int openOrCreateForWriting(const char *file);
// read READ_BYTES bytes from descriptor and stores in buffer
int read0(int descriptor, char *buffer);
// prints on descriptor cantidad from buffer;
void writeInto(int output_descriptor, char *buffer, int cantidad);
```

cp0.h

```c
#include "cp0.h"

int main(int argc, char* argv[]) {
    if (argc != 3)
        return -1;
    cp0(argv[1], argv[2]);
    return 0;
}

void cp0(const char* input, const char* output) {
    int input_descriptor = openForReading(input);
    int output_descriptor = openOrCreateForWriting(output);
    char buffer[READ_BYTES];
    int leidos;
```

```
15      while ((leidos = read0(input_descriptor, buffer)) > 0) {
16          writeInto(output_descriptor, buffer, leidos);
17      }
18      close(input_descriptor);
19      close(output_descriptor);
20  }
21
22  int openForReading(const char *file) {
23      return open(file, O_RDONLY);
24  }
25
26  int openOrCreateForWriting(const char *file) {
27      return creat(file, S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH);
28  }
29
30  int read0(int descriptor, char *buffer) {
31      return read(descriptor, buffer, READ_BYTES);
32  }
33
34  void writeInto(int output_descriptor, char *buffer, int cantidad) {
35      int escritos = 0;
36      while (escritos < cantidad) {
37          escritos += write(output_descriptor, buffer, cantidad);
38      }
39  }
```

cp0.c

2. Comandos opcionales

2.1. touch1

En este comando opcional se agrega una extensión al comando touch0 desarrollado en la parte 1 de este lab. En esta nueva implementación, además de crear archivos en caso de no existir, se deberán modificar las fechas de acceso y modificiación de un archivo existente, alterando su metadata.
A continuacón se mostrará el código de los archivos utilizados.

```c
#define _POSIX_C_SOURCE 200809L
// Open
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
// Close
#include <unistd.h>
// utime
#include <sys/types.h>
#include <utime.h>


int main(int argc, char* argv[]);
void touch1(const char *file);
int fileExists(const char* file);
```

touch1.h

```c
#include "touch1.h"

int main(int argc, char* argv[]) {
    if (argc != 2)
        return -1;
    touch1(argv[1]);
    return 0;
}

void touch1(const char *file) {
    if (!fileExists(file)) {
        int fd = open(file, O_CREAT|O_RDWR, \
            S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH);
        close(fd);
    } else {
        utime(file, NULL);
    }
}

int fileExists(const char* file) {
    struct stat buf;
```

```
22      int result = stat(file, &buf);
23      if (result < 0)
24          return 0;
25      return 1;
26  }
```

touch1.c

2.2. ln1

Esta versión de ln contemplará el caso de la creación de hard links (recordar que ln0 solamente creaba soft links).
A continuacón se mostrará el código de los archivos utilizados.

```
1  //#define _POSIX_C_SOURCE 200809L
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <stdio.h>
6  #include <errno.h>
7
8  int main(int argc, char *argv[]);
9  void ln1(const char* file, const char* link);
10 int fileExists(const char* file);
```

ln1.h

```
1  #include "ln1.h"
2
3  int main(int argc, char *argv[]) {
4      if (argc != 3)
5          return -1;
6      if (!fileExists(argv[1])){
7          printf ("ln: failed to access '%s': No such file or directory\n", argv[1]);
8          return -1;
9      }
10     ln1(argv[1], argv[2]);
11     return 0;
12 }
13
14 void ln1(const char* file, const char* new_file) {
15     link(file, new_file);
16 }
17
18 int fileExists(const char* file) {
19     struct stat buf;
20     int result = stat(file, &buf);
21     if (result < 0)
```

```
22          return 0;
23      return 1;
24  }
```

ln1.c

¿Cuál es la diferencia entre un hard link y un soft link? La diferencia entre un soft link y un hard link radica principalmente en la forma de acceder a los contenidos.
En la caso del soft link, éstos se interpretan en tiempo de ejecución como si el contenido del enlace fuese sustituido por el contenido del path al cual apunta. Además el soft link, tal como se vio anteriormente, puede referir a un archivo inexistente.
Por otror lado, el hard link se usa exactamente igual que el archivo original para cualquier operación. Ambos refieren al mismo archivo (físico) y son indistinguibles, es decir que no se puede saber cuál es el archivo original.

Crear un hard link a un archivo, luego eliminar el archivo original ¿Qué pasa con el enlace? ¿Se perdieron los datos del archivo? Al tratarse de un hard link, ambos archivos apuntan directamente al mismo espacio físico de memoria. Por esto, al eliminar el archivo original lo único que se hace es eliminar una de las referencias a dicha memoria.
Ya que los archivos son indistinguibles, podremos borrar uno o el otro sin perder los datos. Se podría decir que, en el fondo, todo archivo es un hard link.

Repetir lo mismo, pero con un soft link. ¿Qué pasa ahora con el enlace? ¿Se perdieron los datos esta vez? En este caso, al eliminar el archivo origen sí se pierden los datos, creandose un dangling link.
Esto se debe a que, a diferencia de los hard links, los soft links apuntan hacia un path. Si dicho path no existe, no habrá forma de volver a recorrer el camino necesario en memoria para acceder a los datos.