Lab Unix

Alumno: Lucas Lavandeira

Padrón: 98042

Parte 1

rm0

```c
#include <unistd.h>


void rm0(const char* file) {
    unlink(file);
}

int main(int argc, char** argv) {
    rm0(argv[1]);
    return 0;
}
```

cat0

```c
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

#define BUF_SIZE 256

void cat0(const char* file) {
    int fd = open(file, O_RDONLY);
    if (fd < 0) {
        perror("Error abriendo archivo de entrada");
        return;
    }

    ssize_t bytes = 0;
    do {
        char buf[BUF_SIZE] = {0};
        bytes = read(fd, buf, BUF_SIZE);

        if (bytes < 0) {
            perror("Error leyendo del archivo de entrada");
            return;
        }

        if (write(STDOUT_FILENO, buf, BUF_SIZE) < 0) {
            perror("Error escribiendo a salida estándar");
            return;
        }

    } while (bytes == BUF_SIZE);

    if (close(fd) < 0) {
        perror("Error cerrando el archivo de entrada");
        return;
    }
}

int main(int argc, char** argv) {
    cat0(argv[1]);
    return 0;
}
```

touch0

```c
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>


void touch0(const char* file) {
    int fd = open(file, O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH);
    if (fd < 0) {
        perror("Error abriendo el archivo de entrada");
    }

    if (close(fd) < 0) {
        perror("Error cerrando el archivo de entrada");
    }
}

int main(int argc, char** argv) {
    touch0(argv[1]);
}
```

Nota: el uso de los flags de permisos en `open` en esta función y en las siguientes fue el encontrado para lograr permisos 0664 por fuera de usar `umask`

stat0

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

void stat0(const char* file) {
    struct stat s;
    if (stat(file, &s) < 0) {
        perror("Error en stat del archivo de entrada");
        return;
    }

    static const char *msg =
            "Size: %d\n"
            "File: %s\n"
            "Type: %s\n";

    const char *type;
    if (S_ISREG(s.st_mode)) {
        type = "regular file";
    } else if (S_ISDIR(s.st_mode)) {
        type = "directory";
    } else {
        return;
    }
    printf(msg, s.st_size, file, type);
}

int main(int argc, char** argv) {
    stat0(argv[1]);
    return 0;
}
```

rm1 (opcional)

```c
#include <unistd.h>
#include <stdio.h>


void rm1(const char* file) {
    if (unlink(file)) {
        char msg[256];
        snprintf(msg, 256, "rm: cannot remove %s", file);
        perror(msg);
    }
}
```

```
int main(int argc, char** argv) {
    rm1(argv[1]);
}
```

Parte 2

ln0

```
#define _POSIX_C_SOURCE 200809L
#include <unistd.h>
#include <stdio.h>

void ln0(const char* from, const char* to) {
    if (symlink(from, to) < -1) {
        perror("Error creando el symlink");
        return;
    }
}

int main(int argc, const char** argv) {
    ln0(argv[1], argv[2]);
}
```

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

Se crea un symlink roto: el file system va a seguir el enlance pero va a ir a parar a un archivo no existente. Por ejemplo, hacer un `cat` de un symlink roto va a devolver el error de archivo inexistente.

mv0

```
#define _POSIX_C_SOURCE 200809L

#include <stdio.h>

void mv0(const char* oldpath, const char* newpath) {
    if (rename(oldpath, newpath) < 0) {
        perror("Error en rename");
        return;
    }
}

int main(int argc, const char** argv) {
    mv0(argv[1], argv[2]);
}
```

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

Sí, usar `mv` es la "forma unix" de renombrar archivos

cp0

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

#define BUF_SIZE 4096

void cp0(const char* src, const char* dst) {
    int src_fd = open(src, O_RDONLY);
    if (src_fd < 0) {
        perror("Error abriendo archivo de entrada");
        return;
    }

    int dst_fd = open(dst, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH |
S_IWOTH);
    if (dst_fd < 0) {
        perror("Error creando archivo destino");
        return;
```

```
    }

    ssize_t bytes = 0;
    do {
        char buf[BUF_SIZE] = {0};
        bytes = read(src_fd, buf, BUF_SIZE);

        if (bytes < 0) {
            perror("Error leyendo del archivo de entrada");
            return;
        }

        if (write(dst_fd, buf, (size_t) bytes) < 0) {
            perror("Error escribiendo al archivo destino");
            return;
        }

    } while(bytes);

    if (close(src_fd) < 0) {
        perror("Error cerrando el archivo de entrada");
        return;
    }

    if (close(dst_fd) < 0) {
        perror("Error cerrando el archivo destino");
        return;
    }
}

int main(int argc, const char** argv) {
    cp0(argv[1], argv[2]);
}
```

touch1 (opcional)

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <utime.h>
#include <stdio.h>

void touch1(const char* file) {
    if (utime(file, NULL) < 0) {
        int fd = open(file, O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH);
        if (fd < 0) {
            perror("Error abriendo el archivo de entrada");
        }

        if (close(fd) < 0) {
            perror("Error cerrando el archivo de entrada");
        }
    };

}

int main(int argc, const char** argv) {
    touch1(argv[1]);
}
```

ln1 (opcional)

```
#define _POSIX_C_SOURCE 200809L
#include <unistd.h>
#include <stdio.h>

void ln0(const char* from, const char* to) {
    if (link(from, to) < -1) {
        perror("Error creando el link");
        return;
    }
}

int main(int argc, const char** argv) {
```

```
    ln0(argv[1], argv[2]);
}
```

Preguntas:

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

Un hard link crea una "copia indistinguible" al usuario del archivo. No hay manera de saber cuál de los archivos es el original una vez hecho el link. A nivel implementación del filesystem, un hard link implica crear un archivo nuevo que apunte al mismo inode que el original.

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

El enlace es mantenido, porque es una relación archivo - inode, independiente al archivo original. Por la misma razón, no hay pérdida de datos: el inode que representa a los datos sigue siendo accesible.

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

Los soft links, en cambio, son un archivo "puntero" a otro archivo. Al borrar el original, el enlace termina apuntando a un archivo no existente, y cualquier acceso termina resultando en un error de archivo inválido. Si el archivo original era el único hard link al inodo de los datos, los datos son perdidos.

Parte 3

tee0

```
#define _POSIX_C_SOURCE 200809L
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

#define BUF_SIZE 256


void tee0(const char* file) {
    int fd = open(file, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH);
    if (fd < 0) {
        return;
    }

    ssize_t bytes = 0;
    do {
        char buf[BUF_SIZE] = {0};
        bytes = read(STDIN_FILENO, buf, BUF_SIZE);

        if (bytes < 0) {
            perror("Error leyendo de entrada estándar");
            return;
        }

        if (write(STDOUT_FILENO, buf, bytes) < 0) {
            perror("Error escribiendo a salida estándar");
            return;
        }

        if (write(fd, buf, bytes) < 0) {
            perror("Error escribiendo al archivo de salida");
            return;
        }

    } while (bytes);

    if (close(fd) < 0) {
        perror("Error cerrando el archivo de salida");
    }
}

int main(int argc, char** argv) {
    tee0(argv[1]);
}
```

ls0

```
#include <dirent.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

void ls0() {
    DIR* dir = opendir(".");

    if (!dir) {
        perror("Error abriendo el directorio");
        return;
    }

    errno = 0;
    struct dirent *dirent = readdir(dir);
    while(dirent) {

        // homemade strlen
        size_t bytes = 0;
        char c = dirent->d_name[bytes];
        while (c) {
            bytes++;
            c = dirent->d_name[bytes];
        }

        if (write(STDOUT_FILENO, dirent->d_name, bytes) < 0) {
            perror("Error escribiendo el nombre de archivo");
            return;
        }
        write(1, "\n", 1);
        dirent = readdir(dir);
    }

    if (errno) {
        perror("Error leyendo el directorio");
        return;
    }

    if (closedir(dir) < 0) {
        perror("Error cerrando el directorio");
        return;
    }
}

int main() {
    ls0();
}
```

cp1

```
#define _POSIX_C_SOURCE 200112L
#include <sys/stat.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

void cp1(const char* src, const char* dst) {

    size_t page_size = (size_t) sysconf(_SC_PAGESIZE);
    if (page_size == -1 && errno == EINVAL) {
        perror("Error obteniendo el tamaño de página del sistema");
        return;
    }

    int src_fd = open(src, O_RDWR);
    if (src_fd < 0) {
        perror("Error abriendo archivo de entrada");
        return;
    }
```

```
    int dst_fd = open(dst, O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH |
S_IWOTH);
    if (dst_fd < 0) {
        perror("Error creando archivo destino");
        return;
    }

    size_t offset = 0;
    struct stat s;
    if (fstat(src_fd, &s) < 0) {
        perror("Error en stat del archivo de entrada");
        return;
    }

    size_t remaining = s.st_size - offset;
    size_t bytes_to_write =  page_size;
    while (remaining) {
        if (remaining < page_size) {
            bytes_to_write = remaining;
        }

        void *map_src = mmap(NULL, page_size, PROT_READ | PROT_WRITE, MAP_SHARED, src_fd, offset);
        if (map_src == MAP_FAILED) {
            perror("Error mappeando src");
            return;
        }

        if (ftruncate(dst_fd, offset + bytes_to_write)) {
            perror("Error truncando dst");
            return;
        }

        void *mmap_dst = mmap(NULL, bytes_to_write, PROT_READ | PROT_WRITE, MAP_SHARED, dst_fd, offset);
        if (mmap_dst == MAP_FAILED) {
            perror("Error mappeando dst");
            return;
        }

        if (!memcpy(mmap_dst, map_src, bytes_to_write)) {
            perror("Error copiando memoria de src a dst");
            return;
        }

        if (munmap(mmap_dst, bytes_to_write) < 0) {
            perror("Error en el unmap de dst");
            return;
        }

        if (munmap(map_src, bytes_to_write) < 0) {
            perror("Error en el unmap de src");
            return;
        }

        offset += bytes_to_write;
        remaining -= bytes_to_write;
    }

    if (close(src_fd) < 0) {
        perror("Error cerrando el archivo de entrada");
        return;
    }

    if (close(dst_fd) < 0) {
        perror("Error cerrando el archivo destino");
        return;
    }
}

int main(int argc, char** argv) {
    cp1(argv[1], argv[2]);
    return 0;
}
```

ps0 (opcional)

```c
#define _BSD_SOURCE
#include <dirent.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <ctype.h>


#define PROC_PATH_BUF_SIZE 17  // /path/<pid>/stat es a lo sumo 17 bytes
#define PROC_NAME_BUF_SIZE 256

#define STAT_CONTENT_BUF_SIZE 1024

void ps0() {
    const char* proc = "/proc";
    DIR* proc_root = opendir(proc);
    if (!proc_root) {
        perror("Error abriendo el dir /proc");
        return;
    }

    struct dirent* ent = readdir(proc_root);
    while (ent) {
        int is_pid = 1;
        int i = 0;
        char cur = ent->d_name[i++];
        while (cur) {
            if (!isdigit(cur)) {
                is_pid = 0;
                break;
            }
            cur = ent->d_name[i++];
        }

        if (!is_pid || ent->d_type != DT_DIR) {
            ent = readdir(proc_root);
            continue;
        }

        char comm[PROC_PATH_BUF_SIZE] = {0};
        snprintf(comm, PROC_PATH_BUF_SIZE, "/proc/%s/comm", ent->d_name);

        int comm_fd = open(comm, O_RDONLY);
        if (comm_fd < 0) {
            perror("Error abriendo el comm file de un proceso");
            return;
        }

        char proc_name[PROC_NAME_BUF_SIZE] = {0};
        if (read(comm_fd, proc_name, PROC_NAME_BUF_SIZE) < 0) {
            perror("Error leyendo el comm file de un proceso");
        }

        char stat[PROC_PATH_BUF_SIZE] = {0};
        snprintf(stat, PROC_PATH_BUF_SIZE, "/proc/%s/stat", ent->d_name);

        int stat_fd = open(stat, O_RDONLY);
        if (stat_fd < 0) {
            perror("Error abriendo el stat file de un proceso");
            return;
        }

        char stat_content[STAT_CONTENT_BUF_SIZE] = {0};
        if (read(stat_fd, stat_content, STAT_CONTENT_BUF_SIZE) < 0) {
            perror("Error leyendo el stat file de un proceso");
            return;
        }

        char status;
        int matched = sscanf(stat_content, "%*s %*s %c", &status);

        if (matched != 1) {
            perror("Error procesando el stat file con sscanf");
            return;
        }
```

```
            printf("%5s %c %s", ent->d_name, status, proc_name);

            if (close(comm_fd) < 0) {
                perror("Error cerrando comm file de un proceso");
                return;
            }


            if (close(stat_fd) < 0) {
                perror("Error cerrando stat file de un proceso");
                return;
            }

            ent = readdir(proc_root);

        }

        if (errno) {
            perror("Error leyendo entrada del dir /proc");
        }

        if (closedir(proc_root) < 0) {
            perror("Error cerrando /proc");
        }
    }

    int main(int argc, char *argv[]) {
        ps0();
    }
```