Documentación del código de Shell

shell.c

En este .c están los comandos que se han implementado.

listar: listar_directorios(const char *ruta), es una función para listar directorios creados o el contenido de un directorio en específico.

Se declara dos variables necesarias para interactuar con el directorio, “dir” y “entry”, donde “dir” en un puntero a DIR, que representa el directorio abierto y “entry” es un puntero a struct dirent, que contiene la información de cada entrada del directorio.

```
 DIR *dir;
 struct dirent *entry;
```

se verifica si la ruta del directorio es NULL, por ejemplo, cuando solo escribe “listar”, si es asi, se asigna la ruta “.” que representa el directorio actual, entonces lista el directorio actual si no se le especifica.

```
// Si no se especifica una ruta, usamos el directorio actual
    if (ruta == NULL) {
        ruta = ".";
    }
```

Si se especifico la ruta se intentará abrir con “opendir(ruta)”, si no, devuelve null y registramos el error.

```
dir = opendir(ruta);
    if (dir == NULL) {
        printf("Error al abrir el directorio '%s', verifique su
existencia\n", ruta);

        // esto es para ir agregando los errores que se le presentan al
usuario e ir guardando en sistema_error.log
        char mensaje[256];
        snprintf(mensaje, sizeof(mensaje), "Error al abrir el directorio
'%s', verifique su existencia\n", ruta);
        registrar_error(mensaje);  // Registrar en el log

        return;
    }
```

Luego se uso un while con readdir para leer las entradas del directorio una por una, readdir(dir) devuelve un puntero a una estructura dirent que contiene los detalles del directorio, en el “if” se revisa si la entrada no es “.” o sea, cuando no se le especifica o si no es “..”, esto es para volver al directorio padre, y si no son ninguno de esos dos, se imprime el contenido del directorio.

```
//printf("Contenido del directorio '%s':\n", ruta);
    while ((entry = readdir(dir)) != NULL) {
```

```
        // Ignoramos las entradas especiales "." y ".."
        if (strcmp(entry->d_name, ".") != 0 && strcmp(entry->d_name,
"..") != 0) {
            printf("- %s\n", entry->d_name);
        }
    }
```

ir: cambiar_directorio(const char *nombre_directorio), es una función para cambiar de directorio.

Con chdir se intenta cambiar el directorio en donde nombre_directorio es el nombre del directorio en donde quiere acceder el usuario, luego con getcwd se obtiene la ruta completa del nuevo directorio actual y lo imprime.

```
if (chdir(nombre_directorio) == 0) {
        printf("Cambiado al directorio '%s'.\n", nombre_directorio);

        // Obtener y mostrar el nuevo directorio actual
        char ruta_actual[256];
        if (getcwd(ruta_actual, sizeof(ruta_actual)) != NULL) {
            printf("Directorio actual: %s\n", ruta_actual);
        } else {
            printf("Error al obtener el directorio actual: %s\n",
strerror(errno));
        }
    }
```

Sino, registramos el error.

```
else {
        // Mostrar un mensaje si ocurre un error al cambiar de directorio
        printf("Error al cambiar al directorio '%s', verifique su
existencia\n", nombre_directorio);

        // esto es para ir agregando los errores que se le presentan al
usuario e ir guardando en sistema_error.log
        char mensaje[256];
        snprintf(mensaje, sizeof(mensaje), "Error al cambiar al
directorio '%s', verifique su existencia\n", nombre_directorio);
        registrar_error(mensaje);  // Registrar en el log
    }
```

renombrar: renombrar_archivo(const char *nombre_actual, const char *nuevo_nombre), esta función es para renombrar archivos o directorios.

Se utiliza la función rename para cambiar el nombre de cualquier archivo o directorio.

```
if (rename(nombre_actual, nuevo_nombre) == 0) {
        printf("El archivo o directorio '%s' ha sido renombrado a
'%s'.\n", nombre_actual, nuevo_nombre);
    } else {
        printf("Error al renombrar '%s' a '%s', verifique su
existencia\n", nombre_actual, nuevo_nombre);

        // esto es para ir agregando los errores que se le presentan al
usuario e ir guardando en sistema_error.log
        char mensaje[256];
        snprintf(mensaje, sizeof(mensaje), "Error al renombrar '%s' a
'%s', verifique su existencia\n", nombre_actual, nuevo_nombre);
        registrar_error(mensaje);  // Registrar en el log
    }
```

mover: mover_archivo_o_directorio(const char *origen, const char *destino), esta función para mover tanto archivos o directorios en un directorio especificado por el usuario.

Info_destino se utiliza para almacenar la información del destino (usando stat), ruta_final en una cadena en donde se construye la ruta completa del archivo o directorio en el nuevo destino, luego, en el “if” se revisa si el destino es un directorio, y si lo es, se construye la ruta final concatenando el destino con el nombre del archivo o directorio origen, utilizando snprintf y strrchr.

```
if (stat(destino, &info_destino) == 0 && S_ISDIR(info_destino.st_mode)) {
        // Construir la ruta completa (directorio destino + nombre del
archivo o directorio origen)
        snprintf(ruta_final, sizeof(ruta_final), "%s/%s", destino,
strrchr(origen, '/') ? strrchr(origen, '/') + 1 : origen);
    }
```

Luego, se utiliza rename para mover el archivo o directorio, si no se puede, se genera el error y lo registramos.

```
// Intentar mover el archivo o directorio
    if (rename(origen, ruta_final) == 0) {
        printf("Archivo o directorio '%s' movido a '%s'.\n", origen,
ruta_final);
    } else {
        printf("Error al mover '%s' a '%s', verifique si el contenido a
mover o el destino exista\n", origen, ruta_final);

        // esto es para ir agregando los errores que se le presentan al
usuario e ir guardando en sistema_error.log
        char mensaje[256];
```

```
        snprintf(mensaje, sizeof(mensaje), "Error al mover '%s' a '%s',
verifique si el contenido a mover o el destino exista\n", origen,
ruta_final);
        registrar_error(mensaje);  // Registrar en el log
    }
```

Copiar: copiar_archivo(const char *origen, const char *destino), copiar_directorio(const char *origen, const char *destino) y copiar(const char *origen, const char *destino), funciones que sirven para copiar tanto archivos o directorios.

Se verifica si el destino es un directorio, con stat(destino, &dest_stat) se verifica si la ruta de destino es válida, y si es un directorio se construye una nueva ruta de destino añadiendo el nombre del archivo de origen al destino, con basename((char *)origen), obtenemos solo el nombre del archivo sin el cambio completo, para que se agrege al directorio de destino.

```
// Verificar si el destino es un directorio
    if (stat(destino, &dest_stat) == 0 && S_ISDIR(dest_stat.st_mode)) {
        // Si es un directorio, concatenar el nombre del archivo origen
al destino
        snprintf(final_destino, sizeof(final_destino), "%s/%s", destino,
basename((char *)origen));
    } else {
        // Si no es un directorio, usar el destino tal cual
        strncpy(final_destino, destino, sizeof(final_destino) - 1);
        final_destino[sizeof(final_destino) - 1] = '\0';
    }
```

open(origen, O_RDONLY), lo que hace es abrir el archivo de origen en modo lectura.

```
src_fd = open(origen, O_RDONLY);
    if (src_fd < 0) {
        return;
    }
```

open(final_destino, O_WRONLY | O_CREAT | O_TRUNC, 0644), esto abre o crea el archivo de destino en modo escritura.

```
dest_fd = open(final_destino, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (dest_fd < 0) {
        close(src_fd);
        return;
    }
```

read(src_fd, buffer, BUFFER_SIZE), lee hasta BUFFER_SIZE bytes del archivo de origen, y write(dest_fd, buffer, bytes_read), escribe esos bytes en el archivo de destino.

```
// Leer del archivo origen y escribir en el archivo de destino en bloques
de tamaño BUFFER_SIZE
    while ((bytes_read = read(src_fd, buffer, BUFFER_SIZE)) > 0) {
        bytes_written = write(dest_fd, buffer, bytes_read);
        if (bytes_written != bytes_read) {
            close(src_fd);
            close(dest_fd);
            return;
        }
    }
```

Luego, cierra los archivos origen y destino e imprime que se ha podido compiar el archivo.

```
// Cerrar los archivos de origen y destino
    close(src_fd);
    close(dest_fd);

    // Imprimir un mensaje indicando que el archivo ha sido copiado
    printf("Archivo copiado de '%s' a '%s'.\n", origen, final_destino);
```

En **“copiar_directorio”**, opendir(origen) intentara abrir el directorio origen (el que queremos copiar), lo cual abrirá sin problemas si es que especificamos bien lo que se quiere copiar.

```
// Abre el directorio de origen
    dir = opendir(origen);
    if (dir == NULL) {
        //registrar_error("Error al abrir el directorio de origen");
        return;
    }
```

Luego, se construyen las rutas completas de origen y destino para cada entrada del directorio.

```
// Construye las rutas completas de origen y destino
        snprintf(src_path, sizeof(src_path), "%s/%s", origen, entry-
>d_name);
        snprintf(dest_path, sizeof(dest_path), "%s/%s", destino, entry-
>d_name);
```

Por ultimo, copiamos el contenido.

```
// Obtiene información sobre la entrada actual
        if (stat(src_path, &statbuf) == 0) {
```

```
            if (S_ISDIR(statbuf.st_mode)) {
                // Si es un directorio, copiar recursivamente
                copiar_directorio(src_path, dest_path);
            } else if (S_ISREG(statbuf.st_mode)) {
                // Si es un archivo, copiar
                copiar_archivo(src_path, dest_path);
            }
        }
```

Se cierra el directorio origen y se muestra un mensaje de que se pudo realizar la copia.

```
// Cierra el directorio de origen
    closedir(dir);

    // Imprime un mensaje indicando que el directorio ha sido copiado
    printf("Directorio copiado de '%s' a '%s'.\n", origen, destino);
```

La función principal “**copiar**”, es donde revisa si se quiere copiar un archivo o un directorio.

Primeramente, obtiene información del archivo o directorio origen, si existe o no.

```
// Obtiene información sobre el origen
    if (stat(origen, &statbuf) < 0) {

        printf("Error al obtener información del origen, verifica si
existe archivo o directorio.\n");

         // esto es para ir agregando los errores que se le presentan al
usuario e ir guardando en sistema_error.log
        char mensaje[256];
        snprintf(mensaje, sizeof(mensaje), "Error al obtener información
del origen, verifica si existe archivo o directorio.\n");
        registrar_error(mensaje);  // Registrar en el log
        return;
    }
```

Luego, se revisa si el contenido a copiar es un directorio o un archivo, y de acuerdo al caso se llama a la función correspondiente.

```
// Verifica si el origen es un directorio
    if (S_ISDIR(statbuf.st_mode)) {
        // Es un directorio: copiar recursivamente
        copiar_directorio(origen, destino);
    } else if (S_ISREG(statbuf.st_mode)) {
        // Es un archivo: copiar
        copiar_archivo(origen, destino);
    }
```

creardir: crear_directorio(const char *nombre_directorio), función para crear directorios.

Se utilizo la función mkdir en donde se le dio permisos de un directorio “755”.

```
if (mkdir(nombre_directorio, 0755) == 0) {
        printf("Directorio '%s' creado con éxito.\n", nombre_directorio);
    } else {
        // Mostrar un mensaje si ocurre un error al crear el directorio
        printf("Error al crear el directorio '%s': %s\n",
nombre_directorio, strerror(errno));
    }
```

permisos: cambiar_permisos(const char *modo, char **archivos, int num_archivos) y cambiar_permisos_recursivo(const char *ruta, mode_t permisos), con estas funciones se pueden cambiar permisos de un archivo o varios archivos a la vez.

En la función “**cambiar_permisos”** se uso strtol para convertir la cadena modo a un numero octal para representar los permisos.

```
char *endptr;
    mode_t permisos = strtol(modo, &endptr, 8);

    if (*endptr != '\0' || permisos > 0777 || permisos < 0) {
        printf("Error: El modo '%s' no es valido..\n", modo);

        // esto es para ir agregando los errores que se le presentan al
usuario e ir guardando en sistema_error.log
        char mensaje[256];
        snprintf(mensaje, sizeof(mensaje), "Error: El modo '%s' no es
valido.\n", modo);
        registrar_error(mensaje);  // Registrar en el log

        return;
    }
```

Se itera sobre las rutas en caso de querer modificar varios archivos, y si es solo un archivo, no tendrá problemas, solo entrara una vez a la función “**cambiar_permisos_recursivo**”, num_archivos es la cantidad de archivos que colocamos, puede ser uno o varios.

```
for (int i = 0; i < num_archivos; i++) {
        cambiar_permisos_recursivo(archivos[i], permisos);
    }
```

Ahora, estando dentro de la función “**cambiar_permisos_recursivo**”, con stat se obtiene la información sobre la ruta y la guarda en st.

```
// Obtener información sobre la ruta
    if (stat(ruta, &st) == -1) {
        printf("Error al acceder a '%s': %s\n", ruta, strerror(errno));
        return;
    }
```

Se revisa si es un archivo el que va a recibir el cambio de permisos (con S_ISREG(st.st_mode)), si es asi, usamos la función chmod para cambiar los permisos, y si no es un archivo, por ejemplo, es un directorio, se generara un error y se registrara ese error.

```
if (S_ISREG(st.st_mode)) {
        if (chmod(ruta, permisos) == 0) {
            printf("Permisos de '%s' cambiados a '%o'.\n", ruta,
permisos);
        } else {
            printf("Error al cambiar permisos de '%s': %s\n", ruta,
strerror(errno));
        }
    } else {
        printf("Advertencia: '%s' no es un archivo regular. Se
omitirá.\n", ruta);

         // esto es para ir agregando los errores que se le presentan al
usuario e ir guardando en sistema_error.log
        char mensaje[256];
        snprintf(mensaje, sizeof(mensaje), "Advertencia: '%s' no es un
archivo regular. Se omitirá.\n", ruta);
        registrar_error(mensaje);  // Registrar en el log
    }
```

propietario: obtener_uid(const char *nombre_usuario), obtener_gid(const char *nombre_grupo), cambiar_propietario_y_grupo (const char *nombre_usuario, const char *nombre_grupo, char **archivos, int num_archivos), estas funciones se usan para cambiar tanto el usuario como el grupo de un archivo o varios archivos.

Función para obtener el UID de un usuario por su nombre.

```
uid_t obtener_uid(const char *nombre_usuario) {
    struct passwd *pwd = getpwnam(nombre_usuario);
    if (pwd == NULL) {
        return -1; // Usuario no encontrado
    }
    return pwd->pw_uid;
}
```

- **getpwnam(nombre_usuario)**: Busca información del usuario en el sistema. Devuelve un puntero a una estructura passwd.
- **if (pwd == NULL)**: Verifica si el usuario no existe. Retorna -1 como indicador de error.
- **return pwd->pw_uid**: Devuelve el UID del usuario si es encontrado.

Función para obtener el GID dado el nombre del grupo.

```
gid_t obtener_gid(const char *nombre_grupo) {
    struct group *grp = getgrnam(nombre_grupo);
    if (grp == NULL) {
        return -1; // Grupo no encontrado
    }
    return grp->gr_gid;
}
```

- **getgrnam(nombre_grupo)**: Busca información del grupo en el sistema
- **if (grp == NULL)**: Verifica si el grupo no existe. Retorna -1 en caso de error.
- **return grp->gr_gid**: Devuelve el GID del grupo si es encontrado.

Ahora, estando dentro de la función "**cambiar_propietario_y_grupo**", se inicializan los identificadores de usuario y grupo con -1, esto indica que no habrá cambios si no se especifican los valores validos.

```
 uid_t uid = -1; // Inicializamos con -1 (sin cambio)
 gid_t gid = -1; // Inicializamos con -1 (sin cambio)
```

En el "if" revisamos si el nombre de usuario que le pasamos existe y si no le pasamos "-", el guion indica vacio, o sea, que no quieres cambios, solo mantener.

Llamamos a la función para obtener el UID, y en caso de no encontrar al usuario, genera el error.

```
if (nombre_usuario != NULL && strcmp(nombre_usuario, "-") != 0) {
        uid = obtener_uid(nombre_usuario);
        if (uid == (uid_t)-1) {
            printf("Error: El usuario '%s' no existe.\n",
nombre_usuario);

            // esto es para ir agregando los errores que se le presentan
al usuario e ir guardando en sistema_error.log
            char mensaje[256];
            snprintf(mensaje, sizeof(mensaje), "Error: El usuario '%s' no
existe.\n", nombre_usuario);
            registrar_error(mensaje);  // Registrar en el log

            return;
        }
```

```
    }
```

Mismo caso cuando queremos cambiar el “grupo” de un archivo, solo que aca llama a la función obtener_gid, para obtener el GID correspondiente.

```
if (nombre_grupo != NULL && strcmp(nombre_grupo, "-") != 0) {
        gid = obtener_gid(nombre_grupo);
        if (gid == (gid_t)-1) {
            printf("Error: El grupo '%s' no existe.\n", nombre_grupo);

            // esto es para ir agregando los errores que se le presentan
al usuario e ir guardando en sistema_error.log
            char mensaje[256];
            snprintf(mensaje, sizeof(mensaje), "Error: El grupo '%s' no
existe.\n", nombre_grupo);
            registrar_error(mensaje);  // Registrar en el log

            return;
        }
    }
```

En caso de que, si existan, se procede a cambiar los propietarios del archivos o archivos.

Se itera sobre cada archivo recibido para el cambio utilizando chown para hacer los cambios de propietario, también si se genera el error, lo mostramos en pantalla y lo registramos.

```
// Cambiar propietario y grupo para cada archivo/directorio
    for (int i = 0; i < num_archivos; i++) {
        if (chown(archivos[i], uid, gid) == 0) {
            printf("Propietario de '%s' cambiado a '%s' y grupo a
'%s'.\n",
                   archivos[i],
                   (nombre_usuario && strcmp(nombre_usuario, "-") != 0) ?
nombre_usuario : "(sin cambio)",
                   (nombre_grupo && strcmp(nombre_grupo, "-") != 0) ?
nombre_grupo : "(sin cambio)");
        } else {
            printf("Error al cambiar propietario o grupo de '%s': %s\n",
archivos[i], strerror(errno));

            // esto es para ir agregando los errores que se le presentan
al usuario e ir guardando en sistema_error.log
            char mensaje[256];
            snprintf(mensaje, sizeof(mensaje), "Error al cambiar
propietario o grupo de '%s': %s\n", archivos[i], strerror(errno));
            registrar_error(mensaje);  // Registrar en el log
        }
```

```
    }
```

usuario: usuario_existe(const char *nombre_usuario) y agregar_usuario(const char *nombre_usuario, const char *contrasena, const char *horario, const char *lugares_conexion), estas funciones se usan para agregar nuevos usuarios en la Shell.

La función usuario_existe, verifica si el usuario que se quierer agregar ya existe, si existe será imposible agregarlo otra vez, esta función es llamado dentro de la función "**agregar_usuario**".

- **getpwnam(nombre_usuario)**: Busca información del usuario en el sistema, si el usuario no existe, devuelve NULL.
- **return (pwd != NULL)**: Devuelve 1 si el usuario existe, de lo contrario 0.

```
// Función para verificar si un usuario ya existe
int usuario_existe(const char *nombre_usuario) {
    struct passwd *pwd = getpwnam(nombre_usuario);
    return (pwd != NULL); // Retorna 1 si el usuario existe, 0 si no
}
```

Ahora, dentro de la función "**agregar_usuario**", verificamos la existencia del usuario, si genera el error, significa que el usuario ya fue agregado.

```
if (usuario_existe(nombre_usuario)) {
        printf("Error: El usuario '%s' ya existe.\n", nombre_usuario);
        return;
    }
```

Separamos la hora de entrada y salida con una coma entre ellos, esto lo hacemos para poder hacer mejor la comparativa de si el usuario entro con su horario permitido o no.

```
// Separar el horario en hora de entrada y hora de salida
    char hora_entrada[6], hora_salida[6];
    sscanf(horario, "%5[^,],%5s", hora_entrada, hora_salida);
```

Se uso useradd para crear al usuario, se ejecuta el comando con system(comando).

```
char comando[256];
    snprintf(comando, sizeof(comando), "useradd -m %s", nombre_usuario);
    if (system(comando) != 0) {
        printf("Error al agregar el usuario '%s'. Verifica si tienes
permisos de root.\n", nombre_usuario);
        return;
    }
```

Y para establecer la contraseña se uso chpasswd, que también se ejecuto con system.

```
// Establecer la contraseña
    snprintf(comando, sizeof(comando), "echo '%s:%s' | chpasswd",
nombre_usuario, contrasena);
    if (system(comando) != 0) {
        printf("Error al establecer la contrasena para '%s'.\n",
nombre_usuario);
        return;
    }
```

Todos los usuarios agregados se guardan en /usr/local/bin/usuarios_agregados.txt, se guardo en esa ruta para que sea accesible para todos y que no se generen problemas de permisos a la hora de usar la información.

Se abre el archivo de modo anexado para que no sobrescriba las otras informaciones ya presentes, luego se escribe los datos del usuario nuevo con el formato que se ve en el codigo.

```
// Guardar los datos del usuario en el archivo
    FILE *archivo = fopen("/usr/local/bin/usuarios_agregados.txt", "a");
    if (archivo == NULL) {
        printf("Error al abrir el archivo para registrar datos
adicionales.\n");
        return;
    }

    // Guardar los datos en el formato:
nombre_usuario|hora_entrada,hora_salida|IPs_permitidas
    fprintf(archivo, "%s|%s|%s\n", nombre_usuario, horario,
lugares_conexion);
    fclose(archivo);
```

contrasena: cambiar_contrasena(const char *nombre_usuario, const char *nueva_contrasena), esta función se usa para cambiar la contraseña de cualquier usuario estando como root y estando como no root, solo podras cambiar tu contraseña.

Primeramente, obtenemos el nombre del usuario, para verificar si ingresamos como root o como no root, utilizando getenv(“USER”).

```
// Obtener el usuario actual
    const char *usuario_actual = getenv("USER");
    if (usuario_actual == NULL) {
        printf("Error: No se pudo determinar el usuario actual.\n");
        return;
    }
```

verificamos si es que el usuario que esta intentando cambiar la contraseña de algún usuario, es el usuario “root” o es un usuario “normal”, si es un usuario “normal” y si coincide el nombre del usuario_actual con nombre_usuario (aca “nombre_usuario” es el nombre del usuario al que se le quiere modificar la contraseña) entonces esto significa que el usuario esta tratando de cambiar su propia contrasena, lo cual es valido.

El cambio de contrasena se hace ejecutando el comando passwd con system.

Y en caso de que sea usuario “root”, hace el mismo análisis.

```
// Verificar si el usuario actual es root o el mismo usuario
    if (strcmp(usuario_actual, "root") == 0 || strcmp(usuario_actual,
nombre_usuario) == 0) {
        if (strcmp(usuario_actual, nombre_usuario) == 0 &&
strcmp(usuario_actual, "root") != 0) {
            // El usuario no root cambiará su propia contraseña
interactivamente
            printf("Cambiando la contrasena del usuario '%s'. Por favor,
introduce la nueva contrasena.\n", usuario_actual);

            char comando[256];
            snprintf(comando, sizeof(comando), "passwd");

            // Ejecutar el comando passwd sin el nombre del usuario
            int resultado = system(comando);
            if (resultado == 0) {
                printf("Contrasena para el usuario '%s' cambiada con
exito.\n", usuario_actual);

                // Registrar en el log
                char mensaje[256];
                snprintf(mensaje, sizeof(mensaje), "Contrasena para el
usuario '%s' cambiada con exito.\n", usuario_actual);
                registrar_error(mensaje);
            } else {
                printf("Error al cambiar la contrasena para el usuario
'%s'.\n", usuario_actual);

                // Registrar en el log
                char mensaje[256];
                snprintf(mensaje, sizeof(mensaje), "Error al cambiar la
contrasena para el usuario '%s'.\n", usuario_actual);
                registrar_error(mensaje);
            }
        } else {
            // El usuario root cambia contraseñas sin restricciones
            if (nueva_contrasena == NULL || strlen(nueva_contrasena) ==
0) {
```

```
                printf("Error: Debes proporcionar la nueva contrasena
para el usuario '%s'.\n", nombre_usuario);
                return;
            }

            char comando[256];
            snprintf(comando, sizeof(comando), "echo '%s:%s' | chpasswd",
nombre_usuario, nueva_contrasena);

            // Ejecutar el comando
            int resultado = system(comando);
            if (resultado == 0) {
                printf("Contrasena para el usuario '%s' cambiada con
exito.\n", nombre_usuario);

                // Registrar en el log
                char mensaje[256];
                snprintf(mensaje, sizeof(mensaje), "Contrasena para el
usuario '%s' cambiada con exito.\n", nombre_usuario);
                registrar_error(mensaje);
            } else {
                printf("Error al cambiar la contrasena para el usuario
'%s'.\n", nombre_usuario);

                // Registrar en el log
                char mensaje[256];
                snprintf(mensaje, sizeof(mensaje), "Error al cambiar la
contrasena para el usuario '%s'.\n", nombre_usuario);
                registrar_error(mensaje);
            }
        }
    }
```

Y en caso de querer cambiar la contrasena de otro usuario estando como ususario “normal” (no root), se generará un error y se registrara ese error.

```
else {
        // Si el usuario no es root y no coincide con la cuenta actual
        printf("Error: No tienes permisos para cambiar la contrasena de
otros usuarios.\n");

        // Registrar en el log
        char mensaje[256];
        snprintf(mensaje, sizeof(mensaje), "Error: Usuario '%s' intento
cambiar la contrasena de '%s' sin permisos.\n", usuario_actual,
nombre_usuario);
        registrar_error(mensaje);
    }
```

Ahora, estaremos viendo también todo lo que se hizo en el **main** para hacer uso de todas las funciones.

Al principio se declaran varias variables que serán de ayuda para poder manipular la Shell.

```
 char comando[256];
    char *accion, *argumentos[10];  // Permitir hasta 10 argumentos
(puedes ajustar este límite)
    int num_argumentos;
    // se usa para hacer el registro de inicio y cierre de sesion
    char ip_actual[50];
    char horario_actual[50];
    char *usuario = getenv("USER");  // con esto obtenemos el 'nombre'
del usuario.
```

Se hacen ya llamadas a funciones para registrar el inicio de sesión por el usuario.

El “0” que se le pasa como ultimo para metro a “**registrar_sesion**”, es como una bandera que se usa para que se haga de forma correcta las comparaciones de que si el usuario ingreso en un gorario permitido o con la ip permitida, o sea, esto es útil en la función “**validar_inicio_sesion**” que se encuentra en **log.c**.

```
obtener_ip_actual(ip_actual, sizeof(ip_actual)); // Para obtener su Ip
actual
    obtener_timestamp(horario_actual, sizeof(horario_actual)); // Usar
esta función para obtener la hora actual

    // registra el inicio de sesion
    registrar_sesion(usuario, "inicio", ip_actual, horario_actual, 0);
```

luego de todo esto, se le da la bienvenida al usuario.

```
//Bienvenido a la terminal personalizada. Escribe 'salir' para
terminar.\n
    printf("Bienvenido a la shell, si quiere cerrarlo escriba el comando
'salir'.\n");
```

El while (1) mantiene la Shell activa hasta que el usuario escriba “salir”

Dentro del while primeramente se imprime el indicador, que es lo que se vera en todo momento estando en la Shell, seguidamente tenemos a fgets, que leera el comando del usuario.

```
printf("> ");
fgets(comando, sizeof(comando), stdin);

        // Limpieza del comando ingresado
```

```
        comando[strcspn(comando, "\n")] = 0;  // Eliminar salto de línea
        while (strlen(comando) > 0 && comando[strlen(comando) - 1] == '
') {
            comando[strlen(comando) - 1] = '\0';  // Eliminar espacios al
final
        }
```

Llamamos la función “**registrar_movimientos**”, para registrar cada comando que ha ejecutado el usuario (funciona como el ‘history’), esta función se encuentra en **log.c**.

```
// Registrar el comando ingresado por el usuario
        registrar_movimientos(comando);
```

Los comandos se procesan al dividir la entrada en una acción principal y argumentos.

Strtok, lo que hace es dividir la cadena en palabras separadas por espacios, en ‘accion’ se guarda la primera palabra del comando (ejemplo: ‘listar dir1’, entonces en ‘accion’ se guarda ‘listar’) y el resto de las palabras se guardan en los argumentos.

```
// Dividimos el comando en acción y argumentos
        accion = strtok(comando, " ");
        num_argumentos = 0;

        // Extraer todos los argumentos restantes
        char *arg = strtok(NULL, " ");
        while (arg != NULL && num_argumentos < 10) {  // Limitar a 10
argumentos
            argumentos[num_argumentos++] = arg;
            arg = strtok(NULL, " ");
        }
```

Luego, accedemos a “if (accion)” en donde estan todos los comandos que serán ejecutados con las llamadas de sus respectivas funciones.

creardir.

Primeramente tenemos a “creardir”, en donde se hace la comparación de si la acción es “creardir”, y si es asi, llamamos a la función que se encargara de crear el directorio pasandole su respectivo argumento como parámetro (los argumentos eran las palabras ingresadas después de la acción, ejemplo, “creardir Directorio_prueba”, entonces aca el argumentos[0] = Directorio_prueba).

```
f (strcmp(accion, "creardir") == 0) {
                if (num_argumentos >= 1) {
                    crear_directorio(argumentos[0]);
                } else {
                    printf("Error: No se proporciono un nombre para el
directorio.\n");
                }
            }
```

listar.

En donde podias especificarle el directorio a listar o no.

```
else if (strcmp(accion, "listar") == 0) {
                listar_directorios(num_argumentos >= 1 ? argumentos[0] :
NULL);
            }
```

ir.

Donde “argumentos[0]” es el directorio que se especificó.

```
else if (strcmp(accion, "ir") == 0) {
                if (num_argumentos >= 1) {
                    cambiar_directorio(argumentos[0]);
                } else {
                    printf("Error: No se proporciono un nombre para el
directorio.\n");
                }
            }
```

renombrar.

“argumentos[0]” es el archivo o directorio al que le quiere cambiar el nombre y “argumentos[1]” es el nuevo nombre.

```
else if (strcmp(accion, "renombrar") == 0) {
                if (num_argumentos >= 2) {
                    renombrar_archivo(argumentos[0], argumentos[1]);
                } else {
                    printf("Error: Debes proporcionar el nombre actual y
el nuevo nombre.\n");
                }
            }
```

mover.

“argumentos[0]” es el archivo o directorio a mover y “argumentos[1]” es el lugar a donde quiere moverlo.

```
else if (strcmp(accion, "mover") == 0) {
                if (num_argumentos >= 2) {
                    mover_archivo_o_directorio(argumentos[0],
argumentos[1]);
                } else {
                    printf("Error: Debes proporcionar el archivo o
directorio origen y el destino.\n");
                }
```

```
            }
```

copiar.

“argumentos[0]” archivo o directorio que se copiara, y “argumentos[1]” es el lugar en donde se copiara.

```
else if (strcmp(accion, "copiar") == 0) {
                if (num_argumentos >= 2) {
                    copiar(argumentos[0], argumentos[1]);
                } else {
                    printf("Error: Debes proporcionar el archivo o
directorio origen y el destino.\n");
                }
            }
```

Permisos.

“argumentos[0]” contiene el ‘modo’, “argumentos[1]” contiene el archivo o la ruta del archivo que se modificara y num_argumentos -1, indica la cantidad de archivos que se modificaran.

```
else if (strcmp(accion, "permisos") == 0) {
                if (num_argumentos >= 2) {
                    // Argumento 1: modo; Resto: archivos
                    cambiar_permisos(argumentos[0], &argumentos[1],
num_argumentos - 1);
                } else {
                    printf("Error: Debes proporcionar el modo y al menos
un archivo.\n");
                }
            }
```

propietario.

“argumentos[0]” contiene el nombre de ‘usuario’, “argumentos[1]” contiene el nombre del ‘grupo’, “argumentos[2]” el archivo o los archivos a modificar y num_argumentos – 2, indica la cantidad de archivos que se modificaran.

```
else if (strcmp(accion, "propietario") == 0) {

                if (num_argumentos >= 3) {
                    // Si hay un usuario y un grupo, cambiar ambos
                    cambiar_propietario_y_grupo(argumentos[0],
argumentos[1], &argumentos[2], num_argumentos - 2);
                }
```

usuario.

“argumentos[0]” es el nombre del usuario a agregar, “argumentos[1]” la contrasena, “argumentos[2]” el horario de entrada y salida y “argumentos[3]” es al IP permitida.

```
else if (strcmp(accion, "usuario") == 0) {
                if (num_argumentos >= 4) {  // Ahora esperamos 4
argumentos: nombre_usuario, contrasena, horario, lugares_conexion
                    agregar_usuario(argumentos[0], argumentos[1],
argumentos[2], argumentos[3]);
                } else {
                    printf("Error: Debes proporcionar el nombre del
usuario, contrasena, horario y lugares de conexion.\n");
                }
            }
```

contrasena.

“argumentos[0]” nombre del usuario al que se le cambiara la contraseña y “argumentos[1]” la nueva contraseña.

```
else if (strcmp(accion, "contrasena") == 0) {
                if (num_argumentos >= 2) {
                    cambiar_contrasena(argumentos[0], argumentos[1]); //
Usuario y nueva contraseña
                } else {
                    printf("Error: Debes proporcionar el nombre del
usuario y la nueva contraseña.\n");
                }
            }
```

ejecutar.

En “argumentos[i]” todo lo que escribió el usuario, y el comando_final se van guardando las cadenas que van siendo concatenados (unidos) para luego llamar a la función “ejecutar_comando” y ejecutar el comando del sistema que se ha especificado.

La función “**ejecutar_comando**” se encuentra en **comando_sistemas.c**

```
else if (strcmp(accion, "ejecutar") == 0) {
                if (num_argumentos >= 1) {
                    // Unir los argumentos en un solo comando
                    char comando_final[256] = "";
                    for (int i = 0; i < num_argumentos; i++) {
                        strcat(comando_final, argumentos[i]);
                        if (i < num_argumentos - 1) {
                            strcat(comando_final, " ");
                        }
                    }
                    // Ejecutar el comando
                    ejecutar_comando(comando_final);
                } else {
                    printf("Error: Debes proporcionar un comando del
sistema para ejecutar.\n");
                }
```

```
            }
```

demonio.

“argumentos[0]” contiene ‘start’ o ‘stop’, dependiendo de lo que quiera hacer el usuario, y “argumentos[1]” el nombre del demonio.

```
else if (strcmp(accion, "demonio") == 0) {
                if (num_argumentos >= 2) {
                    gestionar_demonio(argumentos[0], argumentos[1]);
                } else {
                    printf("Error: Debes proporcionar una acción ('start'
o 'stop') y un servicio.\n");
                }
            }
```

transferir.

“argumentos[0]” este contiene el protocolo que se especifico (scp), “argumentos[1]” el archivo o directorio a transferir y “argumentos[2]” sobre el usuario que va a recibir, su IP y la ruta remota en donde se guardara el contenido a transferir.

```
else if (strcmp(accion, "transferir") == 0) {
                if (num_argumentos == 3) { // Protocolo, archivo_local,
servidor:ruta_remota

                    if (strcmp(argumentos[0], "scp") == 0) {
                        ejecutar_transferencia_scp(argumentos[1],
argumentos[2]);
                    } else {
                        printf("Error: Protocolo '%s' no soportado, solo
se admite 'scp'.\n", argumentos[0]);
                    }
                } else {
                    printf("Error: Uso correcto: transferir scp
<archivo_local> <usuario>@<servidor>:<ruta_remota>\n");
                }
            }
```

Luego, tenemos ‘salir’ que para el while para salir de la Shell.

```
else if (strcmp(accion, "salir") == 0) {
      break;
}
```

y esto se imprime si es que se trato de ejecutar un comando que no existe.

```
else {
      printf("Comando no reconocido: '%s'.\n", accion);
}
```

Por ultimo, despues de que el usuario haya salido de la Shell, se registra el cierre de sesión, en donde se volverá a compara si es que salió en un horario permitido.

```
// Registrar cierre de sesión
    obtener_timestamp(horario_actual, sizeof(horario_actual));
    // el usuario cerro sesion
    registrar_sesion(usuario, "cerro", ip_actual, horario_actual, 1);
```

prototipos.h

En este .h estan los prototipos de funciones usadas en log.c, comando_sistema.c, demonio.c y scp.c, incluye funciones como:

- **int es_comando_prohibido(const char *comando);**
  Esto es para verificar si un comando esta prohibido, por ejemplo, el comando 'cd' esta prohibido o el comando 'ls'.
- **void ejecutar_comando(const char *comando);**
  Para ejecutar comandos del sistema, excepto los comando que se ha implementado.
- **void registrar_error(const char *mensaje);**
  Registra un mensaje de error en un archivo de log.
- **void registrar_movimientos(const char *comando);**
  Registra comandos ejecutados junto con un timestamp en un archivo de log.
- **void obtener_timestamp(char *buffer, size_t buffer_size);**
  Obtiene el timestamp actual en formato de cadena, esto se usa para obtener el horario exacto de los movimientos realizados dentro de la Shell por un usuario.
- **void obtener_ip_actual(char *ip_buffer, size_t buffer_size);**
  Obtiene la dirección IP actual del sistema, se usa para comparar si es que el usuario entro con la IP permitida, por ejemplo.
- **void validar_inicio_sesion(const char *usuario, const char *ip_actual, const char *hora_entrada, const char *hora_salida, FILE *log_file, int es_salida);**
  Para validar el inicio y cierre de sesión de un usuario, trabaja con los datos del usuario y revisa si es permitido.
- **void registrar_sesion(const char *usuario, const char *accion, const char *ip_actual, const char *hora_actual, int es_salida);**
  Registra la información sobre una sesión de usuario, como el nombre del usuario, la acción (si es que inicio o cerro sesion), su horario permitido, ip permitida.
- **void gestionar_demonio(const char *accion, const char *demonio);**
  Esto se usa para acceder a demonio.c y activar el demonio o desactivar, donde en 'accion' recibe "start" o "stop" y también recibe el nombre del demonio en el 2do parámetro.
- **void ejecutar_transferencia_scp(const char *archivo_local, const char *destino);**
  es para ejecutar una transferencia de archivos o directorios utilizando SCP.
- **void registrar_transferencia_log(const char *usuario_origen, const char *archivo_local, const char *destino, int exito);**

Para registrar la transferencia de archivos en un archivo de log, guarda casos de éxito o casos fallidos.

comandos_sistema.c

Este .c contiene funciones que verifican si un comando está prohibido y ejecutan comandos del sistema permitidos, si algún comando esta prohibido, devuelve un mensaje de error.

Lista de comandos que no están permitidos para su ejecución:

```
const char *comandos_prohibidos[] = {
    "cp", "mv", "rename", "ls", "mkdir", "cd", "chmod", "chown",
"passwd", "adduser", "useradd", "service", "systemctl"
};
```

Esta funcion compara el comando ingresado con los comandos prohibidos definidos en la lista `comandos_prohibidos`, si el comando coincide con alguno de los comandos prohibidos, se considera no valido. Retorna ‘1’ si el comando es prohibido, si no, retorna ‘0’ si el comando no está prohibido.

```
int es_comando_prohibido(const char *comando) {
    for (int i = 0; i < sizeof(comandos_prohibidos) /
sizeof(comandos_prohibidos[0]); i++) {
        // Comparar si el comando comienza con un comando prohibido
        if (strncmp(comando, comandos_prohibidos[i],
strlen(comandos_prohibidos[i])) == 0) {
            return 1;
        }
    }
    return 0;
}
```

Antes de ejecutar el comando, esta función verifica si el comando está en la lista de comandos prohibidos, si el comando está permitido, se utiliza “popen” para abrir un proceso y ejecutar el comando, mostrando su salida al usuario.

```
// Función para ejecutar comandos del sistema
void ejecutar_comando(const char *comando) {
    // Verificar si el comando está prohibido
    if (es_comando_prohibido(comando)) {
        printf("Error: El comando '%s' está prohibido.\n", comando);
        return;
    }

    // Abrir un proceso para ejecutar el comando
    FILE *fp = popen(comando, "r");
    if (fp == NULL) {
```

```
        printf("Error: No se pudo ejecutar el comando '%s'.\n", comando);
        return;
    }

    // Leer y mostrar la salida del comando
    char buffer[1024];
    printf("Salida del comando '%s':\n", comando);
    while (fgets(buffer, sizeof(buffer), fp) != NULL) {
        printf("%s", buffer);
    }

    // Cerrar el proceso
    pclose(fp);
}
```

demonio.c

En esta función se simula un demonio que escribe un mensaje en un archivo .log y se ejecuta de manera indefinida.

En **void fake_daemon ()**, se abre un archivo de log, registra que el demonio ha sido iniciado y entra en un bucle infinito para simular el comportamiento de un demonio, en cada ciclo del bucle escribe en el archivo .log que el demonio esta activado y duerme durante 3 segundos.

```
void fake_daemon() {
    // Abre un archivo de log para registrar la actividad del demonio.
    FILE *log = fopen("/tmp/fake_daemon.log", "a");
    if (!log) {
        perror("Error al abrir el archivo de log");
        exit(1);  // Si no se puede abrir el archivo, termina el programa
con error.
    }

    // Registra que el demonio se ha iniciado.
    fprintf(log, "Fake daemon iniciado.\n");
    fflush(log);  // Asegura que el contenido se escriba inmediatamente
en el archivo.

    // Bucle infinito para simular la actividad del demonio.
    while (1) {
        // Registra que el demonio está en ejecución.
        fprintf(log, "fake_daemon ejecutandose...\n");
        fflush(log);  // Asegura que se escriba el contenido en el
archivo.
        sleep(3);  // El demonio duerme 3 segundos entre cada registro,
simulando actividad.
    }
```

```
    fclose(log);  // Este código nunca se alcanzará debido al bucle
infinito.
}
```

En **void iniciar_demonio(const char *demonio)**, se crea un nuevo proceso utilizando fork(), se llama a fork() para crear un nuevo proceso, luego, si es el proceso 'hijo' (pid == 0), crea una nueva sesión con "setsid()" para desvincular al demonio del terminal, si el demonio se llama "fake_daemon" llama a la función "fake_daemon()" para simular el demonio, y lo que hará el 'padre' es imprimir el PID del demonio y luego lo guarda en un archivo.

```
// Esta función crea un nuevo proceso utilizando fork(), que ejecutará el
demonio.
void iniciar_demonio(const char *demonio) {
    pid_t pid = fork();  // Crea un nuevo proceso.
    if (pid < 0) {
        perror("Error al hacer fork");  // Si ocurre un error al hacer
fork, muestra el mensaje de error.
        exit(1);
    }

    if (pid == 0) {
        // Proceso hijo (el demonio).
        setsid();  // Crea una nueva sesión para que el proceso hijo sea
independiente del terminal.

        // Si el demonio es 'fake_daemon', llama a la función
fake_daemon().
        if (strcmp(demonio, "fake_daemon") == 0) {
            fake_daemon();
        } else {
            // Si el demonio no es 'fake_daemon', intenta ejecutar el
demonio real.
            execlp(demonio, demonio, (char *)NULL);
            perror("Error al ejecutar el demonio");
            exit(1);  // Si execlp falla, termina el programa con error.
        }
    } else {
        // Proceso padre: imprime el PID del demonio y lo guarda en un
archivo.
        printf("Demonio '%s' iniciado con éxito. PID: %d\n", demonio,
pid);
        FILE *pid_file = fopen("/tmp/fake_daemon.pid", "w");
        if (pid_file) {
            fprintf(pid_file, "%d\n", pid);  // Guarda el PID generado
por fork() en el archivo.
            fclose(pid_file);
        } else {
```

```
            perror("Error al guardar el PID");
        }
    }
}
```

En la función “**obtener_pid**” solo se obtiene el PID del demonio para darle uso a la hora de querer detener el demonio

En **void detener_demonio(const char *demonio)**, detiene el demonio enviando una señal SIGTERM, primeranmente llama a la función “obtener_pid” para obtener el PID del demonio, y si el PID es valido utiliza “kill()” para enviar la señal SIGTERM lo que solicita al demonio que termine.

```
void detener_demonio(const char *demonio) {
    pid_t pid = obtener_pid(demonio);  // Obtiene el PID del demonio
desde el archivo.
    if (pid == -1) {
        // Si no se encuentra el PID, muestra un mensaje de error y
registra el error en el log.
        printf("Error al intentar detener el demonio (servicio no
inicializado) '%s'.\n", demonio);

        // Registrar en el log
        char mensaje[256];
        snprintf(mensaje, sizeof(mensaje), "Error al intentar detener el
demonio (servicio no inicializado) '%s'.\n", demonio);
        registrar_error(mensaje);

        return;
    }

    // Intenta detener el demonio enviando SIGTERM.
    if (kill(pid, SIGTERM) == 0) {
        printf("Demonio '%s' detenido con exito.\n", demonio);
    } else {
        printf("Error al intentar detener el demonio (servicio no
inicializado)'%s'.\n", demonio);  // Si no se puede detener el demonio,
muestra un error.

        // Registrar en el log
        char mensaje[256];
        snprintf(mensaje, sizeof(mensaje), "Error al intentar detener el
demonio (servicio no inicializado) '%s'.\n", demonio);
        registrar_error(mensaje);
    }


}
```

En **void gestionar_demonio(const char *accion, const char *demonio)**, es solamente donde se coordina el inicio y la detención del demonio, si la acción es ‘start’ y el demonio es ‘fake_daemon’ llama a “iniciar_demonio”, sino, llama a “detener_demonio”.

```
void gestionar_demonio(const char *accion, const char *demonio) {
    // Si la acción es 'start' y el demonio es 'fake_daemon', se inicia
el demonio.
    if (strcmp(accion, "start") == 0 && strcmp(demonio, "fake_daemon") ==
0) {
        iniciar_demonio(demonio);
    }
    // Si la acción es 'stop' y el demonio es 'fake_daemon', se detiene
el demonio...
    else if (strcmp(accion, "stop") == 0 && strcmp(demonio,
"fake_daemon") == 0) {
        detener_demonio(demonio);
    }
    else {
        // Si la acción no es válida, muestra un mensaje de error.
        printf("Accion no valida. Usa 'start' o 'stop'.\n");

        // Registrar en el log
        char mensaje[256];
        snprintf(mensaje, sizeof(mensaje), "Error, accion no valida. Usa
'start' o 'stop'.\n'%s'.\n", demonio);
        registrar_error(mensaje);
    }
}
```

log.c

Este .c contiene varias funciones relacionadas con el registro como registrar errores del sistema, movimientos del usuario, inicio y cierre de sesión, también las validaciones de sesiones de usuarios.

Estos se usan para especificar las ubicaciones de los archivos de log donde se registran los eventos.

```
#define USUARIOS_LOG_PATH "/var/log/shell/usuario_horarios_log.log"
#define MOVIMIENTOS_LOG_PATH "/var/log/shell/shell_movimientos.log"
#define ERRORES_LOG_PATH "/var/log/shell/sistema_error.log"
```

void registrar_error(const char *mensaje), registra los errores que ocurren en el sistema en un archivo de log, registra también el timestamp actual.

Verifica si el directorio /var/log/shell exista, si no, lo crea, seguidamente abre el archivo de log para ir registrando los errores cometidos por el usuario.

Se le dio todos los permisos al archivo para que cualquier usuario pueda escribir en ella.

```
void registrar_error(const char *mensaje) {
    struct stat st;

    // Crear el directorio /var/log/shell si no existe
    if (stat("/var/log/shell", &st) == -1) {
        mkdir("/var/log/shell", 0777);  // Permisos 777 para el
directorio
    }

    // Abrir el archivo de log en modo de adición
    FILE *log_file = fopen(ERRORES_LOG_PATH, "a");
    if (log_file == NULL) {
        printf("Error al abrir el archivo de log '%s': %s\n",
ERRORES_LOG_PATH, strerror(errno));
        return;
    }

    // Cambiar los permisos del archivo a 777
    chmod(ERRORES_LOG_PATH, 0777);

    // Obtener el timestamp actual
    time_t t = time(NULL);
    struct tm *tm_info = localtime(&t);
    char timestamp[26];
    strftime(timestamp, 26, "%Y-%m-%d %H:%M:%S", tm_info);

    // Escribir el mensaje en el log
    fprintf(log_file, "%s %s\n", timestamp, mensaje);

    // Cerrar el archivo
    fclose(log_file);
}
```

void registrar_movimientos(const char *comando), este registra los movimientos realizados por los usuarios en la Shell, o sea, los comandos ejecutados (funciona como ‘history’), la función recibe lo que se ingresó en la Shell (los comandos usados).

También crea el directorio /var/log/shell si es que no existe, y va registrando los movimientos realizados por el usuario en ‘shell_movimienots.log”.

```
void registrar_movimientos(const char *comando) {
    FILE *log;
    char timestamp[20];  // Espacio para el timestamp
    char mensaje[512];   // Espacio para el mensaje formateado
```

```
    // Crear el directorio /var/log/shell si no existe
    struct stat st;
    if (stat("/var/log/shell", &st) == -1) {
        if (mkdir("/var/log/shell", 0777) != 0) {  // para que cualquier
usuario pueda usarlo
            printf("Error al crear el directorio '/var/log/shell': %s\n",
strerror(errno));
            return;
        }
    }

    // Cambiar los permisos del archivo a 777
    chmod(MOVIMIENTOS_LOG_PATH, 0777);

    // Obtener el timestamp actual
    time_t t = time(NULL);
    struct tm *tm_info = localtime(&t);
    strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", tm_info);

    // Formatear el mensaje
    snprintf(mensaje, sizeof(mensaje), "%s - %s\n", timestamp, comando);

    // Abrir el archivo de log en modo de añadir (append)
    log = fopen(MOVIMIENTOS_LOG_PATH, "a");
    if (log != NULL) {
        // Escribir el timestamp y el comando en el archivo
        fprintf(log, "%s", mensaje);
        fclose(log);
    } else {
        printf("Error al abrir el archivo de log '%s': %s\n",
MOVIMIENTOS_LOG_PATH, strerror(errno));
    }
}
```

La función “**obtener_timestamp**” es solo para obtener el tiempo actual y poder registrar todos los casos con sus respectivas horas en la que se han hecho.

La función “**obtener_ip_actual**” se usa para obtener la dirección IP local de la maquina donde se está ejecutando el programa, la IP que se obtiene es la de la maquina en la red interna o local, se especificó con awk '{print $1}' para extraer solo la primera dirección IP que aparece.

Con “fgets” leemos la salida del comando que contiene la IP.

Esta función es llamada en el main, en shell.c para obtenerlo antes y poder trabajar bien.

```
void obtener_ip_actual(char *ip_buffer, size_t buffer_size) {
    // se obtiene la direccion de la red local, la segunda direccion
ip...
```

```
    FILE *fp = popen("hostname -i | awk '{print $1}'", "r");  //
trabajamos con 'hostname -i' con esto no tenemos problemas en el LFS

    if (fp == NULL) {
        strncpy(ip_buffer, "desconocido", buffer_size);
        return;
    }

    fgets(ip_buffer, buffer_size, fp);
    ip_buffer[strcspn(ip_buffer, "\n")] = '\0'; // Eliminar salto de
línea
    pclose(fp);
}
```

En la función "**validar_inicio_sesion**" se revisa si un usuario inicio sesión desde una IP permitida y dentro del horario permitido.

Primeramente lee un archivo "usuarios_agregados.txt" para obtener la información correcta del usuario, luego se declaran las variables para poder cargar en estos las infomaciones del usuario, el formato de guardado de la información del usuario es la siguiente: User1|09:00,15:00|123.45.0.13, seguidamente, en el while se lee línea por línea el contenido de "usuarios_agregados.txt", previamente se van cargando ya las informaciones en cada variable (para separar cada dato se tiene en cuenta "|" y "," o sea el formato, se van cargando todos los datos excepto los símbolos con 'sscanf'), después de todo esto, se revisa si el usuario esta registrado para poder hacer las comparaciones de los datos.

```
void validar_inicio_sesion(const char *usuario, const char *ip_actual,
const char *hora_entrada, const char *hora_salida, FILE *log_file, int
es_salida) {
    char linea[256];
    FILE *usuarios_file = fopen("/usr/local/bin/usuarios_agregados.txt",
"r"); // esto se cre recien cuando se agregan nuevos usuarios
    if (usuarios_file == NULL) {
        return;
    }

    char usuario_guardado[50], hora_entrada_guardada[50],
hora_salida_guardada[50], ips_guardadas[256];
    int encontrado = 0;

    /* en este while lo que se hace es recorrer cada linea del contenido
de usuarios_agregados.txt, y va guardando informacion de cada linea y
revisando si existe el usurio
    que inicio sesion, si existe, antes de comparar si coinciden, ya se
guarda la informacion del usuario, luego se compara y si coincide se
procede a comparar el resto de informacion*/
    while (fgets(linea, sizeof(linea), usuarios_file)) {
```

```
        // Limpiar la cadena de los saltos de línea y espacios al final
        linea[strcspn(linea, "\n")] = 0; // Eliminar salto de línea al
final

        if (sscanf(linea, "%[^|]|%[^,],%[^|]|%[^\n]", usuario_guardado,
hora_entrada_guardada, hora_salida_guardada, ips_guardadas) == 4) {
            // Limpiar espacios adicionales al final de las cadenas
            usuario_guardado[strcspn(usuario_guardado, " ")] = 0; //
Eliminar espacios
            hora_entrada_guardada[strcspn(hora_entrada_guardada, " ")] =
0; // Eliminar espacios
            hora_salida_guardada[strcspn(hora_salida_guardada, " ")] = 0;
// Eliminar espacios

            // esto si coinciden, significa que el usuario que inicio
sesion fue agregado antes...
            if (strcmp(usuario, usuario_guardado) == 0) {
                encontrado = 1;


                // Validar la IP
                if (strstr(ips_guardadas, ip_actual) == NULL) {
                    fprintf(log_file, "Usuario '%s' inicio sesion desde
una IP no permitida: %s. IP permitida: %s\n",
                            usuario, ip_actual, ips_guardadas);
                }

                // Validar hora de entrada al ingresar
                if (!es_salida && strcmp(hora_entrada,
hora_entrada_guardada) != 0) {
                    fprintf(log_file, "Usuario '%s' inicio sesion fuera
del horario permitido. Hora actual: %s, Hora de entrada permitida: %s\n",
                            usuario, hora_entrada,
hora_entrada_guardada);
                }

                // Validar hora de salida al salir
                if (es_salida && strcmp(hora_salida,
hora_salida_guardada) != 0) {
                    fprintf(log_file, "Usuario '%s' cerro sesion fuera
del horario permitido. Hora actual: %s, Hora de salida permitida: %s\n",
                            usuario, hora_salida, hora_salida_guardada);
                }


                break;
            }
        }
    }
```

```
    // cuando el usuario que ingreso no ha sido agregado previamente, no
esta en "usuarios_agregados.txt", en el caso del 'root' por ejemplo.
    if (!encontrado) {
        fprintf(log_file, "Usuario '%s' no esta registrado en el
sistema.\n", usuario);
    }

    fclose(usuarios_file);
}
```

En “**registrar_sesion**”, esta es la función principal llamada desde el main, esto registra el inicio o cierre de sesión de un usuario y luego llama a la función “validar_inicio_sesion” para revisar si es que entro con IP permitida o horario permitido.

```
void registrar_sesion(const char *usuario, const char *accion, const char
*ip_actual, const char *hora_actual, int es_salida) {
    // Crear el directorio /var/log/shell si no existe
    struct stat st;
    if (stat("/var/log/shell", &st) == -1) {
        if (mkdir("/var/log/shell", 0777) != 0) {
            printf("Error al crear el directorio /var/log/shell: %s\n",
strerror(errno));
            return;
        }
    }

    // Verificar si el archivo de log existe, si no, crearlo con permisos
777
    FILE *log_file = fopen(USUARIOS_LOG_PATH, "a");
    if (log_file == NULL) {
        printf("Error al abrir el archivo de log '%s': %s\n",
USUARIOS_LOG_PATH, strerror(errno));
        return;
    }

    chmod(USUARIOS_LOG_PATH, 0777); // Asegurar que el archivo tenga
permisos 777

    // Obtener el timestamp actual
    char timestamp[64];
    obtener_timestamp(timestamp, sizeof(timestamp));

    // Registrar la acción
    fprintf(log_file, "Usuario '%s' %s sesion desde IP '%s' en hora
'%s'.\n", usuario, accion, ip_actual, hora_actual);

    // Validar horario e IP
    validar_inicio_sesion(usuario, ip_actual, hora_actual, hora_actual,
log_file, es_salida);
```

```
    fclose(log_file);
}
```

scp.c

scp.c se utiliza para manejar y registrar transferencias de archivos utilizando “scp”.

La función “**registrar_transferencia_log**”, se utiliza para registrar las transferencias de archivos en un .log llamado “shell_transferencias” ubicado en /var/log/shell, se registran casos exitosos o casos fallidos.

Se crea el directorio /var/log/shell si es que no existe, luego, se abre el archivo de log en modo ‘adicion’ (esto es para no sobrescribir las anteriores), se obtiene el tiempo actual utilizando time(), luego se registra la transferencia en el log.

Si el entero ‘éxito’ es cero (transferencia fallida), sino, (transferencia exitosa)

```
void registrar_transferencia_log(const char *usuario_origen, const char
*archivo_local, const char *destino, int exito) {
    struct stat st;

    // Crear el directorio /var/log/shell si no existe
    if (stat("/var/log/shell", &st) == -1) {
        if (mkdir("/var/log/shell", 0777) != 0) {   // le doy todos los
permisos para que otros puedan escribir en el, que es lo que se busca
para registrar acciones/movimientos
            printf("Error al crear el directorio /var/log/shell: %s\n",
strerror(errno));
            return;
        }
    }

    FILE *log_file = fopen(LOG_TRANSFERENCIAS, "a");
    if (log_file == NULL) {
        printf("Error: No se pudo abrir el archivo de log '%s'.\n",
LOG_TRANSFERENCIAS);
        return;
    }

    chmod(LOG_TRANSFERENCIAS, 0777); // Asegurar que el archivo tenga
permisos 777

    // Obtener el timestamp actual
    char timestamp[64];
    time_t t = time(NULL);
    struct tm *tm_info = localtime(&t);
```

```
    strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", tm_info);


    // Verificar que `usuario_origen` sea valido, en caso de que no sea
valido, le daremos por nombre "desconocido"
    if (usuario_origen == NULL) {
        usuario_origen = "desconocido";
    }

    // Registrar la transferencia en el log
    fprintf(log_file, "[%s] Transferencia %s: Usuario '%s' utilizo scp
para transferir '%s' -> '%s'\n",
            timestamp,
            exito ? "EXITOSA" : "FALLIDA",
            usuario_origen,
            archivo_local,
            destino);

    fclose(log_file);
}
```

En “**ejecutar_transferencia_scp**”, se maneja la transferencia de archivos o directorios mediante el comando “scp”, utilizamos getenv(“USER”) para obtener el nombre del usuario que ejecuta el programa.

```
/ Obtener el usuario actual, esto es para que aparezca el nombre del
usuario a la hora de registrar en el .log
    const char *usuario_origen = getenv("USER");
    if (usuario_origen == NULL) {
        usuario_origen = "desconocido";
    }
```

Se utiliza stat() para verificar si el archivo o directorio a transferir existe, si no existe se registra el error en el log, se le manda “0” como ultimo parámetro, esto es el caso de una tranferencia fallida.

```
// Verificar si el archivo o directorio existe y obtener su información
    if (stat(archivo_local, &st) != 0) {
        printf("Error: No se puede acceder a '%s', verificar su
existencia.\n", archivo_local);
        registrar_transferencia_log(usuario_origen, archivo_local,
destino, 0); // si no existe, sera una transferencia fallida (exito = 0)

        // Registrar en el log si es que no existe el archivo o el
usuario destino
        char mensaje[256];
        snprintf(mensaje, sizeof(mensaje), "Error: No se puede acceder a
'%s', verificar su existencia.\n", archivo_local);
        registrar_error(mensaje);
```

```
        return;
    }
```

Dependiendo si es un archivo o un directorio, se contruye el comando adecuado para hacer la transferencia.

```
// Construir el comando para scp según sea un archivo o un directorio
    char comando[512];

    if (S_ISDIR(st.st_mode)) {
        snprintf(comando, sizeof(comando), "scp -r %s %s", archivo_local,
destino); // Para directorios
    } else {
        snprintf(comando, sizeof(comando), "scp %s %s", archivo_local,
destino); // Para archivos
    }
```

Utilizamos la función system() para ejecutar el comando scp, y se trabaja con ‘1’ para transferencia exitosa o ‘0’ para transferencia fallida.

```
// Ejecutar el comando SCP
    printf("Ejecutando: %s\n", comando);
    int resultado = system(comando); // "system" es para hacer llamada al
sistema y ejecutar "scp arch.txt user1@192.158.0.16:/home/user1"

    if (resultado == 0) { // si 'resultado' es cero, estonces se uso
correctamente el 'scp' y pudo ejecutar sin problema...
        printf("Transferencia exitosa: %s -> %s\n", archivo_local,
destino);
        registrar_transferencia_log(usuario_origen, archivo_local,
destino, 1); // el '1' significa una transferencia "existosa"
    } else { // Comando fallido
        printf("Error en la transferencia: %s -> %s\n", archivo_local,
destino);
        registrar_transferencia_log(usuario_origen, archivo_local,
destino, 0); // el '0' significa una transferencia "fallida"

        // Registrar en el log si es que no existe el archivo o el
usuario destino
        char mensaje[256];
        snprintf(mensaje, sizeof(mensaje), "Error en la transferencia: %s
-> %s, \n", archivo_local, destino);
        registrar_error(mensaje);
    }
}
```