PRÁCTICAS S0

ÍNDICE

P0:

Sistemas Operativos Grado en Informática. Curso 2024-2025 Asignación de laboratorio 0: Introducción al lenguaje de programación C Para familiarizarnos con el lenguaje de programación C, comenzaremos a codificar un shell. La codificación de este shell continuará en las próximas asignaciones de laboratorio. En este shell, debemos mantener dos listas: una para los comandos ingresados al shell y otra para los archivos abiertos del shell.

Comenzaremos con un shell casi vacío, que básicamente es un bucle que:

- Imprime un mensaje de bienvenida
- Lee desde la entrada estándar una línea de texto que incluye un comando (con sus argumentos).
- Almacena este comando en una lista de comandos que se le han dado, cada uno con su número de orden en una lista que llamaremos "historial de comandos".
- Separa el comando y sus argumentos
- Procesa el comando con sus argumentos

En este momento, este shell solo debe entender los comandos descritos a continuación. En las próximas asignaciones de laboratorio, completaremos este shell: construiremos la asignación de laboratorio 1 sobre el código de la asignación de laboratorio 0, la asignación de laboratorio 2 sobre el código de la asignación de laboratorio 1, y así sucesivamente.

Este shell también debe tener la capacidad de abrir y cerrar archivos con las llamadas al sistema open y close. También puede duplicar descriptores de archivo (con la llamada al sistema dup). El shell debe mantener una lista de sus archivos abiertos (simplemente agregando un elemento a esta lista cada vez que se abre un archivo y eliminando una entrada cuando se cierra un archivo), que podemos examinar con el comando apropiado (open), y debe ser coherente con la lista que tiene el sistema (disponible con lsof -p shell pid en Linux).

El archivo ayudaP0.c proporciona algún código que se puede utilizar para esta asignación de laboratorio.

Comandos:

- **authors:** Imprime los nombres y logins de los autores del programa. autores -l imprime solo los logins y autores -n imprime solo los nombres.
- **pid:** Imprime el ID del proceso que ejecuta el shell.
- **ppid:** Imprime el ID del proceso padre del shell.
- **cd [dir]:** Cambia el directorio de trabajo actual del shell a dir (utilizando la llamada al sistema chdir). Cuando se invoca sin argumentos, imprime el directorio de trabajo actual (utilizando la llamada al sistema getcwd).
- **date [-t|-d]:** Imprime la fecha actual en el formato DD/MM/AAAA y la hora actual en el formato hh:mm:ss.
- **date -d:** Imprime la fecha actual en el formato DD/MM/AAAA.
- **date -t:** Imprime la hora actual en el formato hh:mm:ss.
- **historic [N|-N]:** Muestra el historial de comandos ejecutados por este shell. Para hacer esto, debemos implementar una lista para almacenar todos los comandos ingresados al shell.

    - historic: Imprime todos los comandos que se han ingresado con su número de orden.
    - historic N: Repite el comando número N (del historial de comandos).
    - historic -N: Imprime solo los últimos N comandos.
- Los estudiantes pueden decidir si el historial comienza a numerar los comandos en 0 o en 1. Hipotéticamente, hay un escenario en el que intentar repetir un comando histórico podría generar un bucle infinito o un desbordamiento de pila (dependiendo de cómo se codifique), por lo que los estudiantes pueden elegir no almacenar llamadas a histórico N en la lista de historial si lo desean (ver NOTAS SOBRE LA IMPLEMENTACIÓN DE LISTAS al final de este documento).
- open [file] mode: Abre un archivo y lo agrega (junto con el descriptor de archivo y el modo de apertura) a la lista de archivos abiertos del shell. Para el modo, utilizaremos cr para O_CREAT, ap para O_APPEND, ex para O_EXCL, ro para O_RDONLY, rw para O_RDWR, wo para O_WRONLY y tr para O_TRUNC.
- Abrir sin argumentos lista los archivos abiertos del shell. Para cada archivo, lista su descriptor, el nombre del archivo y el modo de apertura. El shell heredará de su proceso padre los descriptores de archivo 0, 1 y 2 (stdin, stdout y stderr). Para obtener el modo de apertura de un descriptor (df), podemos utilizar fcntl(fd, F_GETFL).
- close[df]: Cierra el descriptor de archivo df y elimina la entrada correspondiente de la lista.
- dup[df]: Duplica el descriptor de archivo df (utilizando la llamada al sistema dup, creando la entrada correspondiente en la lista de archivos.
- infosys: Imprime información sobre la máquina que ejecuta el shell (obtenida mediante la llamada al sistema uname).
- help[cmd]: ayuda muestra una lista de comandos disponibles. ayuda cmd proporciona una breve ayuda sobre el uso del comando cmd.
- bye

IMPORTANTE

- Este programa debe compilar sin errores (no debe producir advertencias, incluso al compilar con `gcc -Wall`).
- NO SE PERMITEN ERRORES DE TIEMPO DE EJECUCIÓN (errores de segmentación, errores de bus, etc.), a menos que se especifique explícitamente. Los programas con errores de tiempo de ejecución no recibirán puntuación.
- Este programa no debe tener fugas de memoria (por favor, utilice `valgrind` para verificar).
- Cuando el programa no pueda realizar su tarea (por cualquier razón, por ejemplo, intentar cambiar el directorio de trabajo actual a un directorio que no existe o que el shell no tiene suficientes privilegios), debe informar al usuario, proporcionando una descripción apropiada del error, como la proporcionada por `strerror()`, `perror()`, `sys_errlist[errno]`, etc.
- Toda la entrada y salida se realizará a través de la entrada y salida estándar.
- Se proporcionarán archivos ejecutables de una implementación de este shell. Por favor, revise ellos para cualquier duda.
- Los estudiantes deben utilizar UNA DE LAS IMPLEMENTACIONES DE LISTA comentadas a continuación.

Información sobre las llamadas al sistema y las funciones de la biblioteca necesarias para codificar este programa está disponible a través de `man`: (`printf`, `gets`, `read`, `write`, `exit`, `getpid`, `getppid`, `getcwd`, `chdir`, `time`, `open`, `close`, `dup`, etc.).

PRESENTACIÓN DE TRABAJO

- El trabajo debe realizarse en parejas.
- El nombre del archivo principal del programa será `p0.c` en una carpeta llamada `P0`. El programa debe poder compilarse con `gcc p0.c`. Alternativamente, se puede proporcionar un archivo `Makefile` en la carpeta `P0` para que el programa se pueda compilar con solo `make`.
- Solo uno de los miembros del grupo de trabajo debe presentar el código fuente. Los nombres y logins de todos los miembros del grupo deben estar en el código fuente del programa principal (al principio del archivo, como comentarios).
- **FECHA LÍMITE**: Viernes 27 de septiembre. Este trabajo de laboratorio no recibirá puntuación, ni será evaluado. Sin embargo, todo el código para este trabajo debe reutilizarse para los siguientes trabajos de laboratorio. Este trabajo también ayudará a familiarizarse con el procedimiento de presentación de todos los siguientes trabajos de laboratorio (a partir del próximo trabajo de laboratorio, el trabajo presentado incorrectamente no será evaluado).
- El procedimiento de presentación se anunciará en una fecha posterior.

INDICIOS

- Un shell es básicamente un bucle:

```
while (!terminado) {
    imprimirPrompt();
    leerEntrada();
    procesarEntrada();
}
```

- `imprimirPrompt()` y `leerEntrada()` pueden ser tan simples como llamadas a `printf` y `gets` (hay una razón por la que se debe utilizar `fgets()` en lugar de `gets()`).
- El primer paso al procesar la cadena de entrada es dividirla en palabras. Para esto, la función de biblioteca `strtok` es útil. Tenga en cuenta que `strtok` no asigna memoria ni copia cadenas, solo rompe la cadena de entrada insertando caracteres de fin de cadena (`'\0'`). La siguiente función divide la cadena apuntada por `cadena` (supuestamente no nula) en un array nulo-terminado de punteros (`trozos`). La función devuelve el número de palabras que estaban en `cadena`:

```
c

int TrocearCadena(char *cadena, char *trozos[]) {
    int i = 1;
    if ((trozos[0] = strtok(cadena, " \n\t")) == NULL)
        return 0;
    while ((trozos[i] = strtok(NULL, " \n\t")) != NULL)
        i++;
    return i;
}
```

NOTAS SOBRE LA IMPLEMENTACIÓN DE LISTA

- Las implementaciones de lista deben consistir en los tipos de datos y las funciones de acceso. Todo acceso a la lista debe hacerse utilizando las funciones de acceso mencionadas anteriormente.
- Los estudiantes pueden elegir una de las tres implementaciones de lista siguientes:

    0. Lista enlazada: La lista se compone de nodos dinámicamente asignados. Cada nodo tiene algún elemento de información y un puntero al nodo siguiente. La lista en sí es un puntero al primer nodo, cuando la lista está vacía, este puntero es `NULL`, por lo que las funciones `CreateList`, `InsertElement` y `RemoveElement` deben recibir la lista por referencia, ya que pueden (en el caso de insertar o eliminar el primer elemento) modificar la lista. También se puede utilizar una versión doblemente enlazada de esta lista (cada nodo tiene dos punteros).
    1. Lista enlazada con nodo cabeza: Similar a la lista enlazada, excepto que la lista en sí es un puntero a un nodo vacío (con información nula) primer nodo. Crear la lista es asignar este primer elemento (nodo cabeza). `CreateList` debe recibir la lista por referencia, mientras que `InsertElement` y `RemoveElement` pueden recibir la lista por valor. También se puede utilizar una versión doblemente enlazada de esta lista (cada nodo tiene dos punteros).
    2. Matriz de punteros: La lista es una matriz (ya sea estática o dinámicamente asignada) de punteros. Cada puntero apunta a un elemento en la lista que se asigna dinámicamente. Para implementar la lista con esta matriz, podemos utilizar un array nulo-terminado o podemos utilizar enteros adicionales. También podemos hacer que la lista sea completamente dinámica utilizando un puntero dinámicamente asignado en lugar de un array de tamaño fijo.

P1:

Sistemas Operativos

Grado en Informática 2024/2025

Trabajo de Laboratorio 1: Sistemas de archivos

Continuamos con la programación del shell que comenzamos en el trabajo de laboratorio 0. Añadiremos los siguientes comandos. Revisa el shell proporcionado para verificar el funcionamiento y la sintaxis de los comandos. Puedes usar el comando **help** o **"command -?"** para obtener ayuda:

- **makefile**: crea un archivo
- **makedir**: crea un directorio
- **listfile**: proporciona información sobre archivos o directorios
- **cwd**: imprime el directorio de trabajo actual
- **listdir**: lista el contenido de los directorios
- **reclist**: lista los directorios de manera recursiva (subdirectorios después)
- **revlist**: lista los directorios de manera recursiva (subdirectorios antes)
- **erase**: elimina archivos y/o directorios vacíos
- **delrec**: elimina archivos y/o directorios no vacíos de manera recursiva

Aunque el trabajo de laboratorio 0 no tuvo puntuación, estos comandos podrán ser evaluados al calificar el trabajo de laboratorio 1.

- **authors**: imprime los nombres y logins de los autores del programa
- **cd**: cambia el directorio de trabajo actual del shell
- **historic**: muestra el historial de comandos ejecutados por este shell
- **open**: abre un archivo y lo añade (junto con el descriptor de archivo y el modo de apertura) a la lista de archivos abiertos del shell
- **close**: cierra un descriptor de archivo y elimina el ítem correspondiente de la lista
- **dup**: duplica el descriptor de archivo, creando la nueva entrada correspondiente en la lista de archivos
- **help**: muestra ayuda sobre los comandos
- **exit**: termina el shell

IMPORTANTE:

- La información sobre las llamadas al sistema y funciones de biblioteca necesarias para programar estos comandos está disponible a través de **man**: (**open, opendir, readdir, lstat, unlink, mkdir, rmdir, realpath, readlink**...) (si decides usar **ftw** o **nftw**, ¡prepárate para que te pregunten cómo funcionan!)
- Se proporciona un shell de referencia (para varias plataformas) para que los estudiantes puedan verificar cómo debe funcionar el shell. Este programa debe ser revisado para encontrar la sintaxis de los diversos comandos.
- Se proporciona un archivo adicional en C (**ayudaP1.c**) con algunas funciones útiles.
- Para verificar qué tipo de objeto de sistema de archivos es un nombre, se debe usar una de las llamadas al sistema **stat**.

NO USES EL CAMPO *d_type* EN LA ENTRADA DE DIRECTORIO.

- El programa debe compilarse sin errores (sin generar advertencias, incluso al compilar con **"gcc -Wall"**).
- **NO SE PERMITIRÁN ERRORES DE EJECUCIÓN** (segmentación, error de bus...). Los programas con errores de ejecución no recibirán puntuación.
- Estos programas no deben tener fugas de memoria (puedes usar **valgrind** para verificarlo).
- Cuando el programa no pueda realizar su tarea (por cualquier razón, por ejemplo, falta de privilegios), debe informar al usuario (ver la sección de errores).
- Toda la entrada y salida se debe hacer a través de la entrada y salida estándar.

ERRORES:

Cuando una llamada al sistema no puede realizar la tarea que se le pidió, devuelve un valor especial (generalmente -1) y establece una variable externa entera (**errno**) en un código de error. La página de **man** de la llamada al sistema explica la razón por la cual produjo dicho código de error.

Un mensaje genérico que explique el error se puede obtener de cualquiera de estos métodos:

- La función **perror()** imprime ese mensaje en el error estándar (la pantalla, si no se ha redirigido el error estándar).
- La función **strerror()** devuelve una cadena con la descripción del error si le proporcionamos el código de error.
- El array externo de punteros, **extern char * sys_errlist[]**, contiene las descripciones de los errores indexadas por número de error, de modo que **sys_errlist[errno]** tiene una descripción del error asociado a **errno**.

ENTREGA DEL TRABAJO

- El trabajo debe realizarse en parejas.
- Se utilizará **campusvirtual.udc.gal** para enviar el código fuente: un archivo comprimido **zip** que contenga SOLO un directorio llamado **P1** (en mayúsculas), donde residirán todos los archivos fuente del trabajo de laboratorio. El nombre del archivo zip debe ser **p1.zip** (en minúsculas).
- El nombre del programa principal será **p1.c** (en minúsculas). El programa debe poder compilarse con **gcc p1.c**. Opcionalmente, se puede suministrar un **Makefile** para que todo el código fuente se pueda compilar con solo usar **make**. En ese caso, el programa compilado debe llamarse **p1** y estar ubicado en el mismo directorio **P1** (sin subdirectorios de compilación o similares).
- **SOLO UNO DE LOS MIEMBROS DEL GRUPO** enviará el código fuente. Los nombres y logins de todos los miembros del grupo deben aparecer en el código fuente del programa principal (en la parte superior del archivo).
- Los trabajos que no cumplan con estas reglas serán descartados.
- **FECHA LÍMITE**: 23:00, viernes 25 de octubre de 2024.

P2

Sistemas Operativos

Grado en Informática 2024/2025
Tarea de Laboratorio 2: Memoria

Continuamos codificando el shell que comenzamos en la primera tarea de laboratorio. Ahora el shell tendrá la capacidad de asignar o desasignar bloques de memoria a través de `malloc`, `mmap` o memoria compartida. El shell mantendrá una lista de los bloques de memoria asignados (solo aquellos asignados con el comando `allocate`, no los que necesita asignar para su funcionamiento normal).

Comandos y Parámetros

Asignación de Memoria

1. **`allocate -malloc n`**
   Asigna un bloque de memoria de `malloc` de tamaño `n` bytes. Actualiza la lista de bloques de memoria.
2. **`allocate -mmap file perm`**
   Mapea un archivo a memoria con permisos `perm`. Actualiza la lista de bloques de memoria.
3. **`allocate -createshared cl n`**
   Crea un bloque de memoria compartida con clave `cl` y tamaño `n`, y lo adjunta al espacio de direcciones del proceso. Actualiza la lista de bloques de memoria.
4. **`allocate -shared cl`**
   Adjunta un bloque de memoria compartida al espacio de direcciones del proceso (el bloque debe estar ya creado pero no necesariamente adjunto al espacio del proceso). Actualiza la lista de bloques de memoria.

Desasignación de Memoria

5. **`deallocate -malloc n`**
   Desasigna un bloque de memoria de `malloc` de tamaño `n` (siempre que haya sido previamente asignado con `allocate`). Actualiza la lista de bloques de memoria.
6. **`deallocate -mmap file`**
   Desmapea un archivo de la memoria (siempre que haya sido previamente mapeado). Actualiza la lista de bloques de memoria.
7. **`deallocate -shared cl`**
   Desacopla un bloque de memoria compartida con clave `cl` (siempre que haya sido previamente asignado). Actualiza la lista de bloques de memoria.
8. **`deallocate -delkey cl`**
   Elimina el bloque de memoria con clave `cl` del sistema. **NO DESACOPLA LA MEMORIA COMPARTIDA CON ESA CLAVE EN CASO DE QUE ESTÉ ADJUNTA.**

9. **`deallocate addr`**
   Desasigna el bloque con dirección `addr`. (Si es un bloque de `malloc`, lo libera; si es un bloque de memoria compartida, lo desacopla…). Actualiza la lista de bloques de memoria.

Operaciones de Memoria

10. **`memfill addr cont ch`**
    Llena la memoria con el carácter `ch`, comenzando en la dirección `addr`, durante `cont` bytes.

11. **`memdump addr cont`**
    Vuelca el contenido de `cont` bytes de memoria en la dirección `addr` a la pantalla. Vuelca códigos hexadecimales, y para los caracteres imprimibles, el carácter asociado.

12. **`memory -funcs`**
    Imprime las direcciones de 3 funciones del programa y 3 funciones de biblioteca.

13. **`memory -vars`**
    Imprime las direcciones de 3 variables externas, 3 externas inicializadas, 3 estáticas, 3 estáticas inicializadas y 3 variables automáticas.

14. **`memory -blocks`**
    Imprime la lista de bloques asignados.

15. **`memory -all`**
    Imprime todo lo anterior (`-funcs`, `-vars` y `-blocks`).

16. **`memory -pmap`**
    Muestra la salida del comando `pmap` para el proceso del shell (equivalente a `vmmap` en macOS).

Operaciones de Archivos

17. **`readfile file addr cont`**
    Lee `cont` bytes de un archivo en la dirección de memoria `addr`.

18. **`writefile file addr cont`**
    Escribe en un archivo `cont` bytes comenzando en la dirección de memoria `addr`.

19. **`read df addr cont`**
    Lo mismo que `readfile` pero usando un descriptor de archivo (ya abierto).

20. **`write df addr cont`**
    Lo mismo que `writefile` pero usando un descriptor de archivo (ya abierto).

21. **`recurse n`**
    Ejecuta la función recursiva `n` veces. Esta función tiene un arreglo automático de tamaño 2048 y un arreglo estático de tamaño 2048. Imprime las direcciones de ambos arreglos y su parámetro (así como el número de recursiones) antes de llamarse a sí misma.

Información Adicional

IMPORTANTE:
Debemos implementar (implementación de lista libre) una lista de bloques de memoria. Para cada bloque debemos almacenar:

- Su dirección de memoria.
- Su tamaño.
- Hora de asignación.
- Tipo de asignación (memoria `malloc`, memoria compartida, archivo mapeado).
- Otra información: clave para bloques de memoria compartida, nombre de archivo y descriptor de archivo para archivos mapeados.

Los comandos del shell `allocate` y `deallocate` asignan y desasignan bloques de memoria y los añaden (o eliminan) de la lista. Cada elemento de la lista tiene información sobre un bloque de memoria que creamos con el comando `allocate`. La información sobre ese bloque es la previamente mencionada. Trataremos con tres tipos de bloques de memoria:

1. **Memoria `malloc`:** Esta es la memoria más común que usamos, se asigna con la función de biblioteca `malloc` y se desasigna con la función de biblioteca `free`.
2. **Memoria compartida:** Esta es memoria que puede ser compartida entre varios procesos. La memoria se identifica por un número (llamado clave) para que dos procesos que usan la misma clave accedan al mismo bloque de memoria. Usamos la llamada al sistema `shmget` para obtener la memoria y `shmat` y `shmdt` para colocarla en (o desacoplarla de) el espacio de direcciones del proceso. `shmget` necesita la clave, el tamaño y los flags. Usaremos `flags=IPC_CREAT | IPC_EXCL | permisos` para crear uno nuevo (da error si ya existe) y `flags=permisos` para usar uno ya creado. Para eliminar una clave, usaremos el comando `deallocate -delkey` (este comando no desasigna nada, solo elimina la clave). El estado de la memoria compartida en el sistema se puede verificar a través de la línea de comandos con el comando `ipcs`. Se proporciona un archivo C adicional (`ayudaP2.txt`) con algunas funciones útiles.
3. **Archivos mapeados:** También podemos mapear archivos en memoria para que aparezcan en el espacio de direcciones de un proceso. Las llamadas al sistema `mmap` y `munmap` hacen el trabajo. Nuevamente, el archivo C adicional (`ayudaP2.txt`) se proporciona con algunas funciones útiles.

El contenido de nuestra lista debe ser compatible con lo que el sistema muestra con el comando `pmap` (o `procstat vm`, `vmmap`, dependiendo de la plataforma). La función recursiva tiene un arreglo estático y un arreglo dinámico del mismo tamaño (2048 bytes) y muestra sus direcciones junto con la dirección y el valor del parámetro.

Errores de Tiempo de Ejecución Legítimos

Aunque **NO SE PERMITIRÁN ERRORES DE TIEMPO DE EJECUCIÓN** (error de segmentación, error de bus, etc.) y los programas con errores de tiempo de ejecución no obtendrán puntuación, este programa puede legítimamente producir errores de segmentación en escenarios como los siguientes:

- `memdump` o `memfill` intentan acceder a una dirección no válida proporcionada a través de la línea de comandos.
- `memfill`, `readfile` o `read` corrompen la pila de usuario o el heap.

Recordatorios

- La información sobre las llamadas al sistema y funciones de biblioteca necesarias para codificar estos programas está disponible a través de `man`: (`shmget`, `shmat`, `malloc`, `free`, `mmap`, `munmap`, `shmctl`, `open`, `read`, `write`, `close`, etc.).
- Se proporciona un shell de referencia (para varias plataformas) para que los estudiantes verifiquen cómo debería funcionar el shell. Este programa debe ser revisado para encontrar la sintaxis y el comportamiento de los diversos comandos. **POR FAVOR, DESCARGA LA ÚLTIMA VERSIÓN** (el comando de versión, en el shell de referencia más nuevo, muestra qué versión estás usando).
- El programa debe compilar sin errores (no producir advertencias incluso al compilar con `gcc -Wall`).
- Estos programas no pueden tener fugas de memoria (puedes usar `valgrind` para comprobarlo).
- Cuando el programa no pueda realizar su tarea (por cualquier motivo, por ejemplo, falta de privilegios), debe informar al usuario (ver sección de errores).
- Todas las entradas y salidas se realizan a través de la entrada y salida estándar.
- Los errores deben ser tratados como en la tarea de laboratorio anterior.

Entrega del Trabajo

- El trabajo debe hacerse en parejas.
- Se utilizará Moodle para enviar el código fuente: un archivo zip que contenga un directorio llamado `P2` donde residan todos los archivos fuente de la tarea de laboratorio.
- El nombre del programa principal será `p2.c`. El programa debe poder compilarse con `gcc p2.c`. Opcionalmente, se puede proporcionar un `Makefile` para que todo el código fuente pueda compilarse con solo hacer `make`. Si ese es el caso, el programa compilado debe llamarse `p2` y colocarse en el mismo directorio que las fuentes (sin directorios de construcción ni similares).
- **SOLO UNO DE LOS MIEMBROS DEL GRUPO enviará el código fuente.** Los nombres y logins de todos los miembros del grupo deben aparecer en el código fuente del programa principal (en la parte superior del archivo).
- Los trabajos enviados que no cumplan con estas reglas serán desestimados.
- **FECHA LÍMITE:** 23:00, viernes 22 de noviembre de 2023.

P3

Sistemas Operativos

Grado en Informática 2024/2025

Tarea de laboratorio 3: Procesos

Continuamos codificando el shell que comenzamos en la primera tarea de laboratorio. Añadiremos los siguientes comandos. Consulta el shell de referencia suministrado para entender cómo funcionan y su sintaxis exacta. (Puedes usar el comando de ayuda, "comando -?" o "comando -help" para obtener ayuda):

1. **`getuid`**: Muestra las credenciales del proceso (real y efectiva).
2. **`setuid [-l] id`:** Cambia la credencial efectiva del proceso (`-l` para login).
3. **`showvar v1 v2 ...`**: Muestra el valor y la dirección de las variables de entorno `v1`, `v2`, ...
    - El acceso debe realizarse usando el tercer argumento de `main()`, la variable global `environ` y la función de biblioteca `getenv`.
4. **`changevar [-a|-e|-p] var val`**: Cambia el valor de una variable de entorno. Solo se puede crear una nueva variable si se accede con `putenv` (`-p`).
5. **`subsvar [-a|-e] v1 v2 val`**: Sustituye una variable de entorno `v1` por otra `v2` con valor `val`.
6. **`environ [-environ|-addr]`:** Muestra el entorno del proceso.
7. **`fork`:** El shell realiza la llamada al sistema `fork` y espera a que termine su proceso hijo.
8. **`search`:** muestra o modifica la lista de búsqueda (la lista de directorios donde se encuentra el shell busca ejecutables)
    - `-add dir`: Añade un directorio a la lista de búsqueda.
    - `-del dir`: Elimina un directorio de la lista de búsqueda.
    - `-clear`: Limpia la lista de búsqueda.
    - `-path`: Importa los directorios en la variable PATH a la lista de búsqueda.
9. **`exec progspec`**: Ejecuta, sin crear un nuevo proceso, el programa descrito por `progspec` (ver explicación más abajo).
10. **`execpri prio progspec`**: Ejecuta un programa con una prioridad específica (`prio`), sin crear un nuevo proceso.
11. **`fg progspec`**: Crea un proceso que ejecuta en primer plano el programa descrito por `progspec`.
12. **`fgpri prio progspec`**: Igual que `fg`, pero cambia la prioridad del proceso a `prio`.

13. **`back progspec`**: Crea un proceso que ejecuta en segundo plano el programa descrito por `progspec`.
14. **`backpri prio progspec`:** Igual que `back`, pero cambia la prioridad del proceso a `prio`.
15. **`listjobs`:** Lista los procesos en segundo plano (ejecutados con `back` o `backpri`).
16. **`deljobs -term|-sig`:** Elimina procesos en segundo plano de la lista.
17. **`****`-** Para cualquier cosa que no sea un comando de shell, el shell asumirá que es un programa externo (en el formato de ejecución que se describe a continuación). Esto es equivalente a fg progspec siendo **** progspec

---

TÓPICOS IMPORTANTES

I - LISTAS

Debemos implementar (sin restricciones en la implementación de la lista) dos listas: una lista de procesos que se ejecutan en segundo plano y una lista de directorios donde el shell busca archivos ejecutables: la lista de búsqueda.

1. **Lista de procesos que se ejecutan en segundo plano** (procesos creados con los comandos del shell `back` o `backpri`). Para cada proceso debemos mostrar:
    - **Su PID**
    - **Fecha y hora de lanzamiento**
    - **Estado** (**TERMINADO**, **DETENIDO**, **SEÑALIZADO** o **ACTIVO**) (con el valor de retorno/señal cuando sea relevante)
    - **Su línea de comandos**
    - **Su prioridad**

Los comandos del shell `listjobs` y `deljobs` muestran y manipulan esta lista. Solo los procesos lanzados desde el shell para ejecutarse en segundo plano (con los comandos `back` o `backpri`) se añadirán a la lista.

Insertamos los procesos en la lista como **ACTIVOS**. Debemos actualizar, utilizando la llamada al sistema `waitpid`, el estado de los procesos antes de imprimir. Ten en cuenta que la llamada `waitpid` informa sobre los cambios de estado, no sobre el estado en sí, de hecho, solo informa UNA VEZ que un proceso ha finalizado.

pid_t waitpid (pid_t pid, int *wstatus, int options);
El parámetro `wstatus` solo tiene un valor significativo si `waitpid` devuelve el `pid`.

La **prioridad** (ya que puede cambiar) debe obtenerse en el momento de la impresión, por lo que no es necesario almacenarla en la lista.

La **ejecución en primer plano** significa que el proceso principal espera a que el hijo termine antes de continuar.

La **ejecución en segundo plano** significa que el proceso principal continúa ejecutándose en paralelo con el hijo; no espera a que el hijo termine.

2. **Lista de directorios donde el shell busca archivos ejecutables**.

Esta es una lista simple de directorios donde el shell buscará los archivos ejecutables para ejecutar. Es análoga al `PATH` del shell del sistema, excepto que la implementamos con una lista y no como una variable de entorno como el shell del sistema. La llamada al sistema `execvp()` busca ejecutables en la lista de directorios contenida en la variable de entorno `PATH`, pero no utilizamos esa llamada al sistema, sino `execve()`, que no busca en los directorios del `PATH` y nos permite pasar un entorno alternativo.

La función `Ejecutable` que se muestra a continuación (también disponible en el archivo `ayudaP3-25.c.txt`) se encarga de encontrar un ejecutable en la lista de búsqueda (siempre que hayamos implementado las funciones `SearchListFirst()` y `SearchListNext()` para obtener los directorios de nuestra lista de búsqueda).

```
char * Ejecutable (char *s)
{
   static char path[MAXNAME];
   struct stat st;
   char *p;

   if (s == NULL || (p = SearchListFirst()) == NULL)
      return s;
   if (s[0] == '/' || !strncmp(s, "./", 2) || !strncmp(s, "../", 3))
      return s;       /* s es una ruta absoluta */

   strncpy(path, p, MAXNAME - 1);
   strncat(path, "/", MAXNAME - 1);
   strncat(path, s, MAXNAME - 1);
   if (lstat(path, &st) != -1)
      return path;

   while ((p = SearchListNext()) != NULL) {
      strncpy(path, p, MAXNAME - 1);
      strncat(path, "/", MAXNAME - 1);
      strncat(path, s, MAXNAME - 1);
      if (lstat(path, &st) != -1)
         return path;
   }
   return s;
}
```

Con esto en mente, podemos fácilmente construir una función que maneje la ejecución de un programa, cambiando (o no) el entorno y cambiando (o no) la prioridad.

```
int Execpve(char *tr[], char **NewEnv, int *pprio)
{
   char *p; /* NewEnv contiene la dirección del nuevo entorno */
            /* pprio la dirección de la nueva prioridad */
            /* NULL indica que no se cambiarán el entorno y/o la
prioridad */

   if (tr[0] == NULL || (p = Ejecutable(tr[0])) == NULL) {
      errno = EFAULT;
      return -1;
   }

   if (pprio != NULL && setpriority(PRIO_PROCESS, getpid(),
*pprio) == -1 && errno) {
      printf("Imposible cambiar prioridad: %s\n", strerror(errno));
      return -1;
   }

   if (NewEnv == NULL)
      return execv(p, tr);
   else
      return execve(p, tr, NewEnv);
}
```

II - `progspec`: formato para ejecución

El formato para crear un proceso que ejecute un programa es:

[VAR1 VAR2 VAR3 ....] executablefile [arg1 arg2.....]
Los elementos dentro de corchetes [ ] son opcionales.

- `executablefile` es el nombre del archivo ejecutable a ejecutar.
- `arg1 arg2 ...` son los argumentos pasados al ejecutable. (El número de estos es indefinido).
- `VAR1 VAR2 ...` (si están presentes) significa que la ejecución será con un entorno que consista únicamente en las variables `VAR1`, `VAR2`, `VAR3` (sus valores serán tomados de `environ`).

La regla es simple: el primer nombre que no sea una variable de entorno (encontrado en `environ`) es el archivo ejecutable.

Ejemplos:

->ls
Ejecuta, creando un proceso en primer plano, el programa `ls`.

-> fg ls -l -a /home
Ejecuta, creando un proceso en primer plano, `'ls -l -a /home'`.

-> fgpri 12 xterm -fg yellow -e /bin/bash
Ejecuta, creando un proceso en primer plano, `'xterm -fg yellow -e /bin/bash'` con su prioridad establecida a 12.

-> back xterm -fg green -bg black -e /usr/local/bin/ksh
Ejecuta creando un proceso en segundo plano `'xterm -fg green -bg black -e /usr/local/bin/ksh'`

-> execpri 9 xterm -fg white
Ejecuta sin crear un nuevo proceso `'xterm -fg white'` con su prioridad establecida a 9

-> fgpri 12 TERM HOME DISPLAY xterm -fg yellow -e /bin/bash
Ejecuta, creando un proceso en primer plano, `'xterm -fg yellow -e /bin/bash'` con su prioridad establecida a 12 en un entorno que solo contiene las variables de entorno `TERM`, `HOME` y `DISPLAY`. Sus valores se toman de `environ`.

-> execpri 9 TERM HOME DISPLAY xterm -fg white
Ejecuta sin crear un nuevo proceso `'xterm -fg white'` con su prioridad establecida a 9 en un entorno que solo contiene las variables de entorno `TERM`, `HOME` y `DISPLAY`. Sus valores se toman de `environ`

-> TERM HOME DISPLAY xterm -fg yellow -e /bin/bash
Ejecuta, creando un proceso en primer plano, `'xterm -fg yellow -e /bin/bash'` en un entorno que solo contiene las variables de entorno `TERM`, `HOME` y `DISPLAY`. Sus valores se toman de `environ`.

Aunque el shell de referencia también permite especificar & para ejecución en segundo plano, @ para cambios de prioridad e implementa redirección, **NO NECESITAS añadir esa funcionalidad a esta tarea de laboratorio**.

III - CREDENCIALES

Utilizaremos la llamada al sistema `setuid` para cambiar las credenciales del usuario. Bajo circunstancias normales, las credenciales reales y efectivas serán las mismas, por lo que no se permitirán cambios en las credenciales del proceso.

Para verificar esta parte de la tarea del laboratorio, debemos:

- Dar al ejecutable el bit de ejecución `setuid`, es decir, `rwsr-xr-x` (4755).
- Debemos ejecutarlo desde otro usuario para que la credencial real sea la del usuario que ejecuta el archivo y las credenciales guardadas y efectivas sean las del propietario del archivo. (Es una buena idea colocar el archivo ejecutable en un directorio escribible para ambos usuarios, como `/tmp`).
- Las llamadas al sistema `setuid` se comportan de manera diferente para el usuario root, por lo que no debemos usar esa cuenta para verificar esta parte del laboratorio.

RECUERDA:

- La información sobre las llamadas al sistema y las funciones de la biblioteca necesarias para codificar estos programas está disponible a través de `man`: `fork`, `exec`, `waitpid`, `getenv`, `putenv`, `getpriority`, ...
- Se proporciona un shell de referencia (para varias plataformas) para que los estudiantes verifiquen cómo debe funcionar el shell. Este programa debe ser revisado para conocer la sintaxis y el comportamiento de los diversos comandos. **DESCARGA LA VERSIÓN MÁS RECIENTE**.
- El programa debe compilarse sin errores (no debe producir advertencias incluso cuando se compila con `gcc -Wall`).
- Estos programas no deben tener fugas de memoria (puedes usar `valgrind` para comprobarlo).
- Cuando el programa no pueda realizar su tarea (por cualquier motivo, por ejemplo, falta de privilegios), debe informar al usuario.
- Toda la entrada y salida se realiza a través de la entrada y salida estándar.
- Los errores deben ser tratados como en los laboratorios anteriores.
- Se proporciona un archivo adicional en C con algunas funciones útiles.

ENTREGA DEL TRABAJO

- El trabajo debe hacerse en parejas.
- Moodle se utilizará para entregar el código fuente: un archivo zip que contenga un directorio llamado `P3` donde residen todos los archivos fuente de la tarea del laboratorio.
- El nombre del programa principal será `p3.c`. El programa debe poder ser compilado con `gcc p3.c`. Opcionalmente, se puede proporcionar un archivo `Makefile` para que todo el código fuente se compile con solo usar `make`. Si es así, el programa compilado debe llamarse `p3`.
- **SOLO UNO DE LOS MIEMBROS DEL GRUPO** entregará el código fuente. Los nombres y logins de todos los miembros del grupo deben aparecer en el código fuente de los programas principales (en la parte superior del archivo).
- **FECHA LÍMITE**: 23:00, sábado 14 de diciembre de 2024.

FUNCIONAMIENTO DE LOS COMANDOS

FUNCIONES P0 -> básicas

1-authors:

authors [-n|-l] Muestra los nombres y/o logins de los autores

-> authors
Ismael Brea Arias: ismael.brea
Borja Casteleiro Goti: borja.casteleiro

-> authors -n
Ismael Brea Arias
Borja Casteleiro Goti

->authors -l
ismael.brea
borja.casteleiro

->authors -j

(No sale nada si pones mal el argumento)

2-pid:

pid [-p] Muestra el pid del shell o de su proceso padre

-> pid
Pid del shell: xxxx

-> pid -p
Pid padre del shell: xxxx

-> pid -v
Invalid argument, use pid (solo en mi shell, en el de referencia funciona, pero no tiene sentido)

3-ppid:

pid [-p] Muestra el pid del shell o de su proceso padre

-> pid
Pid padre del shell: xxxx

-> pid -v
Invalid argument, use ppid (solo en mi shell, en el de referencia funciona, pero no tiene sentido)

4-cd:

Indicandole una ruta o un directorio que está dentro del que estamos, accede a él. Por ejemplo, imaginate que tenemos una carpeta que se llama hola y queremos acceder a ella con comandos
-> cd hola
Directorio actual: /home/ismael/Escritorio/SO/PRÁCTICAS/P1

Nuevo directorio actual: /home/ismael/Escritorio/SO/PRÁCTICAS/P1/hola
Si no existiese el directorio al que se intenta acceder:
-> cd adios
Error al cambiar de directorio: No such file or directory

5-date:
date [-d|-t] Muestra la fecha y/o la hora actual

-> date
19:08:34
14/10/2024

-> date - d (devuelve solo la fecha)
14/10/2024

-> date -t (devuelve solo la hora)
19:08:34

-> date -r (argumento no existente)
Use: date[-t | -d]

6-historic:
historic [-c|-N|N] Muestra (o borra)el historico de comandos
  -N: muestra los N primeros
  -c: borra el historico
  N: repite el comando N
El historial EMPIEZA EN 0

historic: imaginemos que usamos cd y luego pid
-> historic
--- COMMAND HISTORY: ---
0: cd
1: pid
2: historic

-> historic 1
Ejecutando el comando: pid
Pid del shell: 7873

-> historic -x
*

-> historic -c
Historial eliminado

```
-> historic
--- COMMAND HISTORY: ---
0: historic

-> cwd
/home/ismael/Escritorio/SO/PRÁCTICAS/P1

-> reclist hola
**********hola
      0 texto.txt
**********hola1
   4096  hola1
**********hola/hola1
      0 texto1.txt

-> historic
--- COMMAND HISTORY: ---
0: historic
1: cwd
2: reclist hola
3: historic

-> historic 3
Ejecutando hist (3): historic
--- COMMAND HISTORY: ---
0: historic
1: cwd
2: reclist hola
3: historic
4: historic 3
5: historic

-> historic 2
Ejecutando hist (2): reclist hola
**********hola
      0 texto.txt
**********hola1
   4096  hola1
**********hola/hola1
      0 texto1.txt

-> historic -2
--- ULTIMOS 2 COMANDOS: ---
0: historic
1: cwd
```

7-open:
No abre el archivo para que lo uses directamente en el shell.
Lo que hace es:
a) Abrir el archivo en el sistema operativo.
b) Guardar información sobre ese archivo abierto en una lista del shell.
Cuando abres un archivo, el sistema te da un número (descriptor de archivo).
Ejemplo: `open fich ex` abre "fich" de forma exclusiva y guarda esa información.

Ejemplos de apertura de ficheros: rwx

0= 000: ---
1= 001: --x
2= 010: -w-
3= 011: -wx
4= 100: r--
5= 101: r-x
6= 110: rw-
7= 111: rwx

00 --- Sin permisos.
111 --x Solo ejecución para todos.
222 -w- Solo escritura para todos.
444 r-- Solo lectura para todos.
555 r-x Lectura y ejecución para todos.
666 rw- Lectura y escritura para todos.
777 rwx Todos los permisos para todos.

Imagina que creamos un fichero que se llame fich.
Esto debe hacer open:

-> open
descriptor: 0, offset: ( )-> entrada estandar O_RDWR (stdin)
descriptor: 1, offset: ( )-> salida estandar O_RDWR (stdout)
descriptor: 2, offset: ( )-> error estandar O_RDWR (stderr)
descriptor: 3, offset: ( )-> no usado
descriptor: 4, offset: ( )-> no usado
descriptor: 5, offset: ( )-> no usado
descriptor: 6, offset: ( )-> no usado
descriptor: 7, offset: ( )-> no usado
descriptor: 8, offset: ( )-> no usado
descriptor: 9, offset: ( )-> no usado
descriptor: 10, offset: ( )-> no usado
descriptor: 11, offset: ( )-> no usado
descriptor: 12, offset: ( )-> no usado
descriptor: 13, offset: ( )-> no usado
descriptor: 14, offset: ( )-> no usado
descriptor: 15, offset: ( )-> no usado
descriptor: 16, offset: ( )-> no usado
descriptor: 17, offset: ( )-> no usado
descriptor: 18, offset: ( )-> no usado
descriptor: 19, offset: ( )-> no usado

Cada vez que añadamos abramos descriptor, lo cerremos (close), lo dupliquemos, lo mapeemos etc, el descriptor se establecerá por orden en esta lista.

FUNCIONES P1-> sistemas de ficheros

Resumen básico de sistemas de ficheros unix:
1. **¿Qué es un sistema de ficheros?**
Un **sistema de ficheros** es una forma de organizar y gestionar cómo se almacenan, acceden y manejan los datos (archivos y directorios) en un dispositivo de almacenamiento, como un disco duro o SSD. En Unix, todo está organizado como archivos y directorios, incluso cosas como dispositivos o impresoras.

2. La estructura del sistema de ficheros en Unix

El sistema de archivos de Unix sigue una estructura jerárquica en forma de **árbol invertido**, donde la raíz es un directorio llamado `/`. Todo comienza desde ese directorio raíz.

Ejemplo:

```
/
├── bin
├── home
│   ├── usuario1
│   └── usuario2
└── var
    └── log
```

- `/` es el directorio **raíz**.
- **Dentro del directorio raíz** hay otros directorios como `/bin` (que guarda programas ejecutables), `/home` (donde están las carpetas de los usuarios), y `/var` (que almacena archivos de registros o logs).

3. Archivos y directorios

- **Archivos**: En Unix, todo es un archivo. Un archivo puede ser un documento, una imagen, un programa, etc. Pero también puede ser un dispositivo o incluso un enlace a otro archivo.
- **Directorios**: Son como carpetas que pueden contener otros archivos o directorios. Sirven para organizar los archivos.

4. Inodos (i-nodes)

Aquí entra uno de los conceptos más importantes en Unix: **los inodos**.

- Un **inodo** es como la "tarjeta de información" de un archivo o un directorio. No contiene el contenido del archivo, pero almacena **información sobre él**.

¿Qué tipo de información contiene un inodo?

1. **Tamaño** del archivo.
2. **Permisos** (quién puede leer, escribir o ejecutar el archivo).
3. **Propietario** del archivo.
4. **Fecha de creación/modificación** del archivo.
5. **Punteros a los bloques de datos** (donde está el contenido real del archivo en el disco).

Cuando buscas un archivo, Unix primero accede a su **inodo** para saber dónde están guardados los datos reales en el disco.

Ejemplo:

Supongamos que tienes un archivo llamado `documento.txt`. El sistema de archivos buscará el **inodo** de ese archivo, que le dice dónde están almacenados los datos reales del archivo en el disco.

5. Bloques de datos

El **bloque de datos** es la parte del disco donde se almacena realmente el contenido de un archivo. Cuando un archivo es grande, puede ocupar varios bloques de datos. El inodo tiene punteros que dicen en qué bloques están los datos del archivo.

Relación entre inodo y bloques de datos:

- El inodo no contiene el contenido del archivo, solo información sobre él.
- El contenido del archivo se almacena en los **bloques de datos** del disco, y el inodo tiene punteros que apuntan a esos bloques.

6. Tipos de archivos en Unix

En Unix, hay varios tipos de archivos:

1. **Archivos regulares**: Son los archivos comunes (texto, programas, imágenes, etc.).
2. **Directorios**: Son carpetas que contienen otros archivos o directorios.
3. **Enlaces simbólicos**: Son como accesos directos que apuntan a otro archivo o directorio.
4. **Archivos de dispositivo**: Representan hardware como impresoras o discos duros, pero se tratan como archivos.
5. **FIFOs y sockets**: Son archivos especiales utilizados para la comunicación entre procesos.

7. Permisos y Propietarios

Unix es un sistema multiusuario, y cada archivo tiene un conjunto de **permisos** que controlan quién puede hacer qué con él:

- **Lectura (r)**: Permite leer el contenido del archivo.
- **Escritura (w)**: Permite modificar el contenido del archivo.
- **Ejecución (x)**: Permite ejecutar el archivo (si es un programa).

Cada archivo tiene un **propietario** y un **grupo**, y los permisos se definen para:

- **El usuario propietario**.
- **El grupo** al que pertenece el archivo.
- **Otros** (todos los demás).

Ejemplo de permisos:

Supongamos que un archivo tiene los permisos `rwxr-xr--`:

- El **propietario** puede leer, escribir y ejecutar el archivo.
- El **grupo** puede leer y ejecutar el archivo, pero no modificarlo.
- **Otros** solo pueden leer el archivo.

8. Operaciones comunes en el sistema de archivos Unix

Algunas de las operaciones más comunes que puedes realizar en el sistema de archivos Unix son:

- **Crear archivos y directorios**:
    - Para crear un archivo, puedes usar comandos como `touch`.
    - Para crear un directorio, usas `mkdir`.
- **Mover y renombrar archivos**:
    - Usas el comando `mv` para mover o renombrar un archivo.
- **Eliminar archivos y directorios**:
    - El comando `rm` elimina archivos, y `rmdir` elimina directorios vacíos.
- **Listar archivos**:
    - El comando `ls` te permite ver el contenido de un directorio.

9. Montaje de sistemas de archivos

En Unix, un **sistema de archivos** (o partición) no está accesible automáticamente. Para poder usarlo, se tiene que "montar". El **montaje** es el proceso de hacer que un sistema de archivos sea accesible en un directorio específico del sistema.

Ejemplo:

Si tienes una partición en otro disco, podrías montarla en `/mnt` para acceder a ella a través de ese directorio.

Resumen:

1. **Sistema de archivos lógico**: Es la estructura jerárquica que organiza archivos y directorios.
2. **Inodos**: Almacenan información sobre los archivos (permisos, propietario, tamaño, ubicación de los datos, etc.).
3. **Bloques de datos**: Contienen el contenido real de los archivos.
4. **Permisos**: Controlan quién puede acceder o modificar archivos.
5. **Operaciones**: Comandos como `ls`, `mv`, `rm` te permiten interactuar con el sistema de archivos.
6. **Montaje**: Es necesario montar sistemas de archivos para poder usarlos.

1. `open`

- **Descripción**: Abre un archivo o crea uno nuevo.
- **Uso**: Se utiliza para obtener un descriptor de archivo que se puede usar para leer, escribir o manipular el archivo.
- **Ejemplo**:

```
int fd = open("archivo.txt", O_RDONLY); // Abre el archivo para lectura.
```

2. `opendir`

- **Descripción**: Abre un directorio para poder leer su contenido.
- **Uso**: Devuelve un puntero a una estructura que representa el directorio.
- **Ejemplo**:

```
DIR *dir = opendir("mi_directorio"); // Abre el directorio.
```

3. `readdir`

- **Descripción**: Lee la entrada siguiente de un directorio abierto.
- **Uso**: Se utiliza junto con `opendir` para obtener información sobre los archivos y subdirectorios dentro de un directorio.
- **Ejemplo**:

```
struct dirent *entry;

while ((entry = readdir(dir)) != NULL) {

    printf("%s\n", entry->d_name); // Imprime el nombre de cada archivo/subdirectorio.

}
```

4. `lstat`

- **Descripción**: Obtiene información sobre un archivo o enlace simbólico sin seguir el enlace.
- **Uso**: Similar a `stat`, pero no sigue enlaces simbólicos. Devuelve información como el tamaño, permisos y tipo de archivo.
- **Ejemplo**:

```
struct stat info;

lstat("mi_enlace_simbolico", &info); // Obtiene información del enlace simbólico
```

5. `unlink`

- **Descripción**: Elimina un archivo del sistema de archivos.
- **Uso**: Se utiliza para borrar un archivo. Si es el último enlace al archivo y no hay procesos que lo estén usando, se libera la memoria.
- **Ejemplo**:

```
unlink("archivo_a_borrar.txt"); // Elimina el archivo especificado.
```

6. `mkdir`

- **Descripción**: Crea un nuevo directorio.
- **Uso**: Se utiliza para crear un directorio en el sistema de archivos.
- **Ejemplo**:

mkdir("nuevo_directorio", 0755); // Crea un nuevo directorio con permisos 755.

7. `rmdir`

- **Descripción**: Elimina un directorio vacío.
- **Uso**: Se utiliza para borrar un directorio, pero solo si está vacío.
- **Ejemplo**

rmdir("directorio_vacio"); // Elimina el directorio especificado si está vacío.

8. `realpath`

- **Descripción**: Obtiene la ruta absoluta de un archivo o directorio.
- **Uso**: Convierte una ruta relativa en una ruta absoluta y resuelve cualquier enlace simbólico.
- **Ejemplo**:

char resolved_path[PATH_MAX];
realpath("ruta_relativa", resolved_path); // Resuelve la ruta a su forma absoluta.

9. `readlink`

- **Descripción**: Lee el contenido de un enlace simbólico.
- **Uso**: Devuelve la ruta a la que apunta un enlace simbólico.
- **Ejemplo**:

char buffer[PATH_MAX];
ssize_t length = readlink("mi_enlace_simbolico", buffer, sizeof(buffer)-1);
buffer[length] = '\0'; // Añade un terminador nulo.

Resumen

- **`open`** y **`opendir`** son para abrir archivos y directorios, respectivamente.
- **`readdir`** se usa para leer las entradas de un directorio.
- **`lstat`** y **`readlink`** obtienen información sobre archivos y enlaces simbólicos.
- **`unlink`**, **`mkdir`**, y **`rmdir`** son para eliminar archivos o directorios y crear nuevos directorios.
- **`realpath`** convierte rutas relativas a absolutas.

Campos Comunes de `struct stat`

1. **`st_dev`**:
   - Tipo: `dev_t`
   - Describe el número de dispositivo en el que reside el archivo.
2. **`st_ino`**:
   - Tipo: `ino_t`
   - El número de inodo del archivo, que es un identificador único en el sistema de archivos.
3. **`st_mode`**:
   - Tipo: `mode_t`
   - Contiene los permisos y tipo del archivo (por ejemplo, si es un archivo regular, directorio, enlace simbólico, etc.).
4. **`st_nlink`**:
   - Tipo: `nlink_t`
   - Número de enlaces duros que apuntan al inodo del archivo.
5. **`st_uid`**:
   - Tipo: `uid_t`
   - Identificador de usuario del propietario del archivo.
6. **`st_gid`**:
   - Tipo: `gid_t`
   - Identificador de grupo del archivo.
7. **`st_size`**:
   - Tipo: `off_t`
   - Tamaño del archivo en bytes.
8. **`st_atime`**:
   - Tipo: `time_t`
   - Tiempo de último acceso al archivo.
9. **`st_mtime`**:
   - Tipo: `time_t`
   - Tiempo de la última modificación del contenido del archivo.
10. **`st_ctime`**:
    - Tipo: `time_t`
    - Tiempo del último cambio de metadatos del archivo (como permisos o propietario).
11. **`st_blksize`**:
    - Tipo: `blksize_t`
    - Tamaño del bloque de entrada/salida óptimo para el sistema de archivos.
12. **`st_blocks`**:
    - Tipo: `blkcnt_t`
    - Número de bloques asignados al archivo.

Conceptos Clave

1. **`DIR`**:
    - `DIR` es una estructura definida en `<dirent.h>` que contiene la información necesaria para acceder a un directorio. No está diseñada para ser manipulada directamente por el programador; en su lugar, se usa junto con funciones específicas para navegar por los contenidos del directorio.
2. **Uso de `DIR *dir`**:
    - Cuando se declara un puntero `DIR *dir`, se está reservando espacio para manejar un directorio. Este puntero se utilizará en funciones que permiten abrir, leer y cerrar directorios.

Funciones Comunes Relacionadas con `DIR *dir`

1. **`opendir`**:
    - Abre un directorio y devuelve un puntero `DIR *`.
    - Prototipo:

      c

1-makefile
Crea un fichero cualquiera con el nombre que le pasemos (también se le puede añadir extensión)
->makefile
/home/ismael/Escritorio/SO/PRÁCTICAS/P1 //imprime el dir de trabajo actual

-> makefile hola.txt
Archivo hola.txt creado correctamente

-> makefile hola.txt
Imposible crear hola.txt. FIle exists->

-> makefile
Error: Se requiere un nombre de directorio.

En esta función se usa **open** y **close**:
-open:
`open()` devuelve un **file descriptor** (`fd`) que te permitirá interactuar con el archivo. A través de este `fd`, puedes realizar operaciones como escribir datos en el archivo o cerrarlo cuando ya no lo necesites.
-Localización del archivo: El sistema busca el archivo en el sistema de ficheros.
-**Acceso al inodo del archivo**: El sistema localiza el inodo asociado con el archivo.
-Carga del inodo en memoria: El inodo se carga en la caché de inodos si no está ya en memoria.
-**Asociación con el descriptor de archivo**: El sistema asigna un file descriptor (`fd`) que se asocia con el inodo cargado.

-close:
Al usar `close(fd)`, el sistema libera el descriptor de archivo y asegura que el inodo del archivo y sus datos estén correctamente sincronizados y almacenados en el disco.
-Liberación del descriptor de archivo: El sistema libera el file descriptor (`fd`)
-Escritura de datos pendientes: Si hay datos en caché, se escriben en el disco,

-**Actualización del inodo**: Se actualizan los metadatos del inodo, si es necesario (como las marcas de tiempo)

-Liberación de recursos asociados: Se liberan los recursos del sistema relacionados con el archivo.

2-makedir:
Crea un directorio cualquiera con el nombre que le pasemos
->makedir
/home/ismael/Escritorio/SO/PRÁCTICAS/P1 //imprime el dir de trabajo actual

-> makedir hola
Directorio hola creado correctamente

->makedir hola
Imposible crear el archivo hola: File exists

-> makedir
Error: Se rquiere un nombre de directorio

3-listfile:
Proporciona información sobre uno o varios archivos o directorios en un sistema unix.
-> listfile
/home/ismael/Escritorio/SO/PRÁCTICAS/P1

- **Comprobación de un archivo:** Imaginemos que ya está creado el archivo 'archivo1'

-> listfile archivo1
0 archivo

(Tamaño del archivo) (Nombre del archivo)

->listfile -long archivo1
2024/10/17-17:40 1 (5912197) ismael ismael -rw-r--r-- 0 archivo1

->listfile -acc archivo1
2024/10/17-17:40 0 archivo1

-> listfile -link archivo1 (-link solo funciona si es un enlace simbólico, si no devuelve listfile normal)
0 archivo

->listfile -hid archivo1 (comando no existente para ver que no funciona bien)
Opción no reonocida: -hid

->listfile -long -acc archivo1 (mezcla ambos comandos)
2024/10/17-17:40 1 (5912197) ismael ismael -rw-r--r-- 0 archivo1
2024/10/17-17:40 0 archivo1

- **Comprobación de un directorio** Imaginemos que ya está creado el directorio 'hola'

-> listfile hola
4096 hola

-> listfile -long hola
2024/10/14-18:43 2 (5912266) ismael ismael drwxrwxr-x 4096 hola

(1) (2) (3) (4) (5) (6) (7) (8)

Todo viene del struct stat, que lee el bloque del inodo:
1-fecha y hora de la última modificación del archivo. Se obtiene con el struct stat, especificamente con st_mtime (tiempo de modificación) .
***2- el num 2** indica el número de **hard links** que apuntan al archivo o directorio. Se obtiene del campo st_nlink del struct stat
3-Número del inodo
4-Nombre del propietario del archivo. Se obtiene con getpwuid(file_info.st_uid)
5-Nombre del grupo al que pertence el archivo. Se obtiene usando getgrgid(file_info.st_gid)
6-Permisos del archivo: se dividen en3 grupos de 3 caracteres para cada uno
d: indica que es un directorio (si fuese - indicaría que es un archivo normal)
rwx: permisos del propietario: lectura (r), escritura (w) y ejecución (w).
rwx: permisos del grupo: los mismos que los del propietario.
r-x: permisos para otros usuarios. lectura (r), sin escritura(-) y ejecución

Se obtiene a través del campo st_mode del struct stat y se convierte a un formato legible mediante la función ConvierteModo.

7-Tamaño del archivo en bytes. Se obtiene del campo st_size del struct stat.
8-Nombre del archivo que se está listando. Es el argumento que se le pasó a la función.

*hard links:
Los enlaces duros son una forma de crear múltiples nombres para el mismo archivo en el sistema de archivos, lo que permite a los usuarios acceder a los mismos datos a través de diferentes nombres sin duplicar el espacio en disco. Esto proporciona flexibilidad y eficiencia en la gestión de archivos en sistemas Unix y Linux.

-> listfile -long
Error: Se requiere al menos un nombre de archivo

-listfile -acc hola
-> listfile -acc hola
2024/10/15-21:12 4096 hola

(1) (2) (3)

1-fecha y hora del último acceso al archivo. Se obtiene con el struct stat, especificamente con st_atime (tiempo acceso)
2- Tamaño del archivo
3-Nombre del archivo

-> listfile -acc
Error: Se requiere al menos un nombre de archivo

-> listfile -link hola/texto.txt
6 hola/texto.txt

Muestra el tamaño del enlace simbólico (la cantidad de bytes ocupados por la referencia).

-> listfile -link
Error: Se requiere al menos un nombre de archivo

4-cwd:
Imprime el directorio de trabajo actual

Obtenemos el trabajo de directorio actual usando getcwd() (get current working directory)de la libería <unistd.h>

-> cwd
/home/ismael/Escritorio/SO/PRÁCTICAS/P1

-> cwd -acc
Argumento invalido, usa cwd

5-listdir:
Lista el contenido de los directorios, es decir, muestra todos los archivos y subdirectorios que hay dentro de un directorio específico en el sistema de archivos.
SOLO PARA DIRECTORIOS, YA QUE UN ARCHIVO NO PUEDE CONTENER A OTROS

-long: devuelve un listado largo
-hid: incluye al listdir normal los ficheros ocultos
-acc: evuelve el accestime
-link: si el enlace es simbólico, devuelve el path contenido

Imaginemos que tenemos un directorio hola con un archivo texto.txt dentro

->listdir hola
**********hola
0 texto.txt

(1) (2)

La primera línea indica que se está mostrando el contenido del directorio hola
1- indica los bytes que tiene el archivo
2-es el nombre del archivo

-> listdir
home/ismael/Escritorio/SO/PRÁCTICAS/P1

Para los archivos hace lo mismo que listfile
Vamos a añadir un directorio hola1 al directorio hola y dentro de hola un archivo texto1.txt

-> listdir -hid hola
**********hola
6 texto.txt (1)
4096 .. (2)
4096 . (3)
4096 hola1 (4)

La primera línea indica que se está mostrando el contenido del directorio hola:
1-tamaño del archivo + nombre del archivo
2-tamaño del archivo + enlace simbólico que representa el propio directorio padre del directorio actual
3-tamaño del archivo + enlace simbólico que representa el propio directorio hola
4- tamaño del archivo + otro directorio dentro de hola, en este caso hola1

-> listdir -long hola
**********hola
2024/10/15-21:05 1 (5914662) ismael ismael -rw-r--r-- 6 texto.txt
2024/10/15-21:12 2 (5912768) ismael ismael drwxrwxr-x 4096 hola1

Lo mismo que listfile -long pero con todos los archivos que hay dentro del directorio

-> listdir -acc hola
**********hola
2024/10/15-21:05 6 texto.txt
2024/10/15-21:11 4096 hola1

(1) (2) (3)

La primera línea indica que se está mostrando el contenido del directorio hola:
1-Indica el accesstime. Última vez que se accedió al archivo
2- Tamaño del archivo
3-Nombre del archivo

6-reclist:

Lista los directorios de manera recursiva (subdirectorios después). En elcaso de los archivos, como no se pueden listar, devolvemos solo tamaño y nombre, excepto en -long y -acc que añadimos cosas.

-> reclist
/home/ismael/Escritorio/SO/PRÁCTICAS/P1

Para archivos hace lo mismo que listfile y listdir.
Para directorios:

Imaginemos que tenemos un directorio carpeta1 con un subidirectorio carpeta2 que contiene otro subdirectorio carpeta3 y un archivo2 en el mismo directorio que carpeta2

-> listdir carepta1
****error al acceder a carepta1: No such file or directory

-> listdir carpeta1
**********carpeta1
4096 carpeta2
0 archivo1

-> reclist carpeta1
**********carpeta1
0 archivo1
4096 carpeta2
**********carpeta1/carpeta2
4096 carpeta3
**********carpeta1/carpeta2/carpeta3

-> reclist -long carpeta1
**********carpeta1
2024/10/23-13:25 1 (5912285) ismael ismael -rw-r--r-- 0 archivo1
2024/10/23-13:23 3 (5912283) ismael ismael drwxrwxr-x 4096 carpeta2
**********carpeta1/carpeta2
2024/10/23-13:23 2 (5912284) ismael ismael drwxrwxr-x 4096 carpeta3
**********carpeta1/carpeta2/carpeta3

-> reclist -acc carpeta1

**********carpeta1

2024/10/23-13:23 4096 carpeta2

2024/10/23-13:25 0 archivo1

**********carpeta1/carpeta2

2024/10/23-13:23 4096 carpeta3

**********carpeta1/carpeta2/carpeta3

-> reclist -hid carpeta1

**********carpeta1

4096 ..

4096 .

4096 carpeta2

0 archivo1

**********carpeta1/carpeta2

4096 ..

4096 carpeta3

4096 .

**********carpeta1/carpeta2/carpeta3

4096 ..

4096 .

-> reclist -link carpeta1

**********carpeta1

4096 carpeta2

0 archivo1

**********carpeta1/carpeta2

4096 carpeta3

**********carpeta1/carpeta2/carpeta3

-> reclist -link hola_simbolico
hola_simbolico -> hola
4 hola_simbolico

7-revlist:

Lista los directorios de manera recursiva (subdirectorios antes). En elcaso de los archivos, como no se pueden listar, devolvemos solo tamaño y nombre, excepto en -long y -acc que añadimos cosas.

8-erase:

Elimina archivos y/o directorios VACÍOS

Es decir, ELIMINA cualquier tipo de archivo, y elimina solo directorios que están vacíos. Si un directorio contiene un archivo vacío no se eliminará ya que no esta vacío. Si un directorio contiene unicamente otro subdirectorio tampoco se borrará, ya ue no está vacío. En el caso de que haya un archivo dentro de un directorio tiene que poder borrarse si accedemos al directorio y lo borramos desde donde encuentra. Es decir, si tenemos un directorio 1 y un archivo 1.1, ara borrar el archivo se puede hacer erase 1/1.1.

-> makefile prueba //creamos un archivo prueba de 0 bytes
Archivo 'prueba' creado correctamente

-> erase prueba
Archivo prueba eliminado correctamente

-> makefile prueba //creamos un archivo prueba, pero esta vez escribimos contenido una vez creado
Archivo 'prueba' creado correctamente

-> erase prueba
Archivo prueba eliminado correctamente

-> makedir h
Directorio 'h' creado correctamente

-> erase h
Directorio h eliminado correctamente

Si creo un directorio y le añado un fichero NO se borra, ya que el directorio NO está vacío.
-> makedir 1
Directorio '1' creado correctamente
-> makefile 1/"
Archivo 1/" creado correctamente
-> erase 1
Imposible borrar 1: Directorio no vacío

Puedo borrar " solo haciendo esto:
-> erase 1/"
Archivo 1/" eliminado correctamente
-> erase 1
Directorio eliminado correctamente

Si creo un directorio y le añado un directorio vacío NO se borra, ya que el directorio NO está vacío.
-> makedir 1
Directorio '1' creado correctamente
-> makedir 1/"
Directorio 1/" creado correctamente

->erase 1/"
Directorio 1/" eliminado correctamente
->erase 1
Directorio 1 eliminado correctamente

9-delrec: (recursivamente) **DIFERENTE AL SHELL DE REFEFENCIA (LA HACE MAL)**
Elimina archivos/o directorios NO VACÍOS recursivamente

Es decir, elimina cualquier archivo (esté vacío o no, es indiferente), pero para eliminar directorios no puede estar vacío.

-> makefile prueba //creamos un archivo prueba de 0 bytes
Archivo 'prueba' creado correctamente

-> delrec prueba
Archivo prueba eliminado correctamente

-> makefile prueba //creamos un archivo prueba, pero esta vez escribimos contenido una vez creado
Archivo 'prueba' creado correctamente

-> delrec prueba
Archivo prueba eliminado correctamente

-> makedir 1
Directorio 1 creado correctamente

->delrec 1
Imposible borrar 1: Directorio vacío

->makefile 1/1.1
Archivo 1/1.1 creado correctamente

->delrec 1
Directorio 1 eliminado correctamente
//1 ya no está vacío, por tanto eliminar el directorio

-> makedir 1
Directorio 1 creado correctamente

->makefile 1/1.1
Archivo 1/1.1 creado correctamente

->delrec 1/1.1
Archivo 1/1.1 eliminado correctamente

->delrec 1
Imposible borrar 1: Directorio vacío

->makedir 1/1.1
Directorio 1/1.1 eliminado correctamente

->delrec 1
Directorio 1 eliminado correctamente
//1 ya no está vacío, por tanto eliminar el directorio

FUNCIONES P2-> memoria
***-comentarios (no es la salida)**

Resumen básico de la memoria en Unix:

En esta práctica estamos utilizando direcciones de 64 bits. Estas direcciones son direcciones virtuales, es decir, NO se trabaja directamente con la memoria física (RAM), si no que se usa el concepto de memoria virtual. Es una capa de abstracción entre mi programa y la memoria real.

Tipos de Memoria en Unix

1. **Memoria Dinámica (`malloc`/`free`)**:
    - Se gestiona con funciones estándar de C (`malloc`, `calloc`, `realloc`, `free`).
    - Asigna bloques de memoria en el **heap** del proceso.
    - Debe liberarse manualmente con `free`.
2. **Memoria Compartida**:
    - Permite compartir datos entre procesos.
    - Gestionada por:
        - `shmget`: Reserva o crea un bloque de memoria compartida identificado por una **clave**.
        - `shmat`: Adjunta el bloque al espacio de direcciones del proceso.
        - `shmdt`: Desacopla el bloque del proceso.
        - `shmctl`: Elimina la clave o modifica atributos.
    - Uso común: Comunicación entre procesos.
3. **Archivos Mapeados en Memoria (`mmap`/`munmap`)**:
    - Permiten trabajar con archivos como si fueran bloques de memoria:
        - Los datos del archivo se reflejan directamente en el espacio de direcciones del proceso.
    - Funciones clave:
        - `mmap`: Crea el mapeo.
        - `munmap`: Elimina el mapeo.
    - Permite control de acceso con permisos (`PROT_READ`, `PROT_WRITE`, etc.).

Gestión de Memoria y la Lista de Bloques

La práctica requiere implementar una lista que registre los bloques asignados mediante comandos `allocate` y `deallocate`. Para cada bloque, se debe almacenar:

1. **Dirección**: Dónde se encuentra en memoria.
2. **Tamaño**: Cuántos bytes ocupa.
3. **Hora de Asignación**: Marca temporal para fines de depuración.
4. **Tipo**:
    - `MALLOC`: Bloques creados con `malloc`.
    - `SHARED`: Memoria compartida (`shmget`).

    - `MMAP`: Archivos mapeados (`mmap`).
5. **Detalles adicionales**:
    - **Memoria compartida**: Clave asociada.
    - **Archivos mapeados**: Nombre y descriptor del archivo.

COMBINACIONES DE PERMISOS (FLAGS):

Combinación	Significado
`r`	Solo lectura: puedes leer el contenido del archivo pero no modificarlo.
`w`	Solo escritura: permite modificar el archivo, pero no leerlo ni ejecutarlo (raro).
`x`	Solo ejecución: permite ejecutar código en la región mapeada, pero no leer ni escribir (poco común).
`rw`	Lectura y escritura: puedes leer y escribir en el archivo mapeado.
`rx`	Lectura y ejecución: puedes leer el archivo y ejecutar código en la región mapeada.
`wx`	Escritura y ejecución: puedes modificar el contenido y ejecutar código (poco seguro).
`rwx`	Lectura, escritura y ejecución: acceso completo al archivo mapeado.
`t`	Sin acceso explícito: la región se reserva, pero no puedes accederla (útil para ciertos casos técnicos).

Cuando mapeamos por defecto sin permisos, se abre por defecto los permisos de **solo lectura r**

FUNCIONES DEL CÓDIGO DE AYUDA:

- **función Recursiva** -> recursiva
- **LlenarMemoria** -> memfill
- **ObtenerMemoriaShmget** -> allocate -create shared / allocate -shared
- **do_AllocateCreateshared** -> allocate -create shared
- **do_AllocateShared** -> allocate -shared
- **MapearFichero** -> allocate -mmap
- **do_AllocateMmap** -> allocate -mmap
- **do_DeallocateDelkey** -> deallocate -delkey
- **LeerFichero** -> readfile
- **Cmd_ReadFile** -> readfile
- **Do_pmap** -> memory -pmap

Memoria Virtual vs. Memoria Física

- **Memoria Virtual**: En sistemas modernos, los programas no acceden directamente a la memoria física. En lugar de eso, acceden a **direcciones de memoria virtual** que el sistema operativo traduce a direcciones físicas utilizando la **tabla de páginas**. Cuando se accede a una dirección de memoria compartida, el sistema operativo se asegura de que los cambios sean reflejados en la memoria física.
- **Memoria Física**: Si la memoria compartida está respaldada por un archivo o mapeada a través de un espacio de direcciones en la memoria física, el sistema operativo se encarga de asegurar que los cambios hechos por los procesos que usan esa memoria compartida se reflejen correctamente.

FUNCIONES IMPORTANTES:

-strtoul:
unsigned long strtoul(const char *str, char **endptr, int base);

Parámetros

1. **`str`**:
    - La cadena que contiene el número a convertir.
    - Puede incluir espacios iniciales, signos positivos/negativos y caracteres válidos para el sistema numérico especificado (base).
2. **`endptr`**:
    - Es un puntero a un puntero de carácter.
    - Al finalizar la conversión, **`*endptr`** apunta al primer carácter en `str` que no forma parte del número. Si no interesa, se puede pasar `NULL`.
3. **`base`**:
    - Especifica la base del sistema numérico para la conversión (por ejemplo, 10 para decimal, 16 para hexadecimal).
    - Si `base` es 0, **`strtoul`** determina la base automáticamente:
        - Prefijo `0x` o `0X` → base 16 (hexadecimal).
        - Prefijo `0` → base 8 (octal).
        - Sin prefijo → base 10 (decimal).

1-allocate: Asigna un bloque de memoria

Tenemos 4 casos posibles: malloc, shared, createsheared y mmap
Creamos una lista general que contenga todos estos casos de memoria y 3 listas específicas que funcionen por separado dependiendo del argumento:
malloc, shared y mmap

-> allocate
******Lista de bloques asignados para el proceso 9054

Muestra la lista GENERAL de todos los bloques de memoria asignados en el PID

MALLOC: reserva un bloque en el espacio de direcciones virtuales de tu proceso

-> allocate -malloc
******Lista de bloques asignados malloc para el proceso 5868
No hay bloques de memoria malloc asignados.

*Muestra una lista específica de todos los bloques de memoria asignados en el PID

-> allocate -malloc 1024
Asignados 1024 bytes en 0x58c2ad592af0
Bloque de memoria añadido a la lista de bloques malloc

*Se asignan 1024 bytes en una dirección. Malloc asigna un bloque de memoria en el **heap.** Cuando llamas a `malloc(size)`, el sistema operativo busca un bloque de memoria libre del tamaño solicitado. Si lo encuentra, reserva ese espacio y devuelve la dirección inicial del bloque (una dirección de memoria en el *heap*).

-> allocate -malloc
******Lista de bloques asignados malloc para el proceso 10420
0x58c2ad592af0 1024 Nov 08 20:17 malloc
(1) (2) (3) (4)

(1)-dirección donde se asignó la memoria con malloc
(2)- tamaño en bytes
(3)-fecha de asignación. Para obtenerla usamos:
struct tm *tm_info = localtime -> para obtener la hora en que se asignó con el struct en segundos y localtime convierte ese valor en formato time_t
strftime -> formateamos la hora para obtenerla como la queremos. convierte la fecha y hora almacenada en `tm_info` en una cadena de texto con el formato `Mes Día Hora:Minuto`
(4)-tipo de asignación: malloc
*Se muestra la lista exclusiva de malloc que contiene los bloques de datos abiertos

-> allocate
******Lista de bloques asignados para el proceso 10420
  0x58c2ad592af0   1024 Nov 08 20:17 malloc

*Muestra la lista general de todos los bloques de memoria asignados indendientmente del tipo

-> allocate malloc
Uso: allocate [-malloc n]

-> allocate -malloc -3
Error: el tamaño debe ser un número positivo

MMAAP: cuando mapeas un archivo a memoria usando `mmap`, estamos creando una asociación entre el contenido del archivo en disco y una dirección de memoria virtual dentro del espacio de tu proceso. Esto te permite trabajar con el contenido del archivo como si estuviera cargado en memoria, sin necesidad de usar funciones tradicionales como `fread` o `fwrite`.

-> allocate mmap
uso: allocate [-malloc|-shared|-ceateshared|-mmap] ....

-> allocate -mmap
******Lista de bloques asignados mmap para el proceso 6279
No hay bloques de memoria mmap asignados

-> allocate -mmap ar.txt
fichero ar.txt mapeado en 0x777469246000
*POR DEFECTO SI NO SE PASAN PERMISOS, SE ABRE EN MODO LECTURA

-> allocate -mmap ar.txt ex
fichero ar.txt mapeado en 0x777469245000

- MMAP nos puede servir más adelante para hacer los siguiente:

Paso 1: Mapear el archivo.
Comando:
```
allocate -mmap ejemplo.txt rw
```
Salida:
```
fichero ejemplo.txt mapeado en 0x485a000
```

Paso 2: Leer contenido desde memoria.
Comando:
```
read -mmap 0x485a000
```
Salida:
```
Hola, Mundo!
```

Paso 3: Modificar el contenido.
Comando:
```
write -mmap 0x485a000 7 "Universo"
```

Esto reemplaza "Mundo" por "Universo".

Paso 4: Guardar los cambios y liberar la memoria.
Comando:
```
deallocate -mmap ejemplo.txt
```

El archivo `ejemplo.txt` ahora contiene:
```
Hola, Universo!
```

-CREATESHARED: (crear compartido)

-SHARED: (compartido)
Para shared usar dos terminales para comprobar que funciona la memoria compartida.
Cada proceso tiene su propia memoria. En este caso, al ser memoria compartida varios procesos comparten la misma memoria, de manera que si en un proceso escribo una cosa el otro la lee. Puedo probar a hacer en un proceso memfill y ver como en el otro proceso se puede ver lo que se hizo aqui.

Abrir 2 terminales: en una creamos shared, en otra hacemos shared y nos da una dirección
abrimos otra terminal y hacemos otra shared

2-deallocate: Deasigna memoria

MALLOC: elimina el primer bloque de la lista que encuentre con ese tamaño
Imaginemos que he hecho allocate -malloc 1024 y 256

-> deallocate
******Lista de bloques asignados para el proceso 7421
0x61ac7f104c40 1024 Nov 09 11:05 malloc
0x61ac7f104af0 256 Nov 09 11:04 malloc

-> deallocate -malloc 256
Memoria de 256 bytes en 0x61ac7f104af0 liberada y eliminada de la lista de bloques malloc

-> deallocate -malloc 102
Memoria de 102 bytes en 0x61ac7f104c40 liberada y eliminada de la lista de bloques malloc

-> deallocate
******Lista de bloques asignados para el proceso 7421
No hay bloques de memoria asignados

-> deallocate -malloc 256
No hay bloque de ese tamaño asignado con malloc

MMAP: elimina el primer bloque de memoria mapeada que encuentre de un archivo por orden de como se añadió. Si añadimos un descriptor wr a un archivo ar.txt y un descriptor ex después, si hacemos deallocate -mmap ar.txt primero solo se borra la dirección del descriptor wr

Imaginemos ese caso:

-> deallocate
******Lista de bloques asignados para el proceso 6376
0x79e70949c000 5 Nov 18 11:31 ar.txt (descriptor 3)
0x79e70949b000 5 Nov 18 11:31 ar.txt (descriptor 4)

> deallocate -mmap ar.txt
Memoria mapeada en 0x485a000, de tamaño 5, liberada

-> deallocate
******Lista de bloques asignados para el proceso 8868
0x485b000 5 2024-11-18 11:30:33(descriptor 4) mmap

-> deallocate -mmap ar.txt
Memoria mapeada en 0x485b000, de tamaño 5, liberada

-> deallocate
******Lista de bloques asignados para el proceso 8868

SHARED Y DELKEY:

`-deallocate -shared cl`

Desacopla un bloque de memoria compartida con clave `cl` (siempre que haya sido previamente asignado). Actualiza la lista de bloques de memoria.

`-deallocate -delkey cl`

Elimina el bloque de memoria con clave `cl` del sistema. **NO DESACOPLA LA MEMORIA COMPARTIDA CON ESA CLAVE EN CASO DE QUE ESTÉ ADJUNTA.**

1. **`shared`**:
    - Desacopla la memoria compartida de un proceso (es decir, el proceso ya no puede acceder a esa memoria), pero **no elimina la clave** ni la memoria del sistema. Otros procesos pueden seguir usando esa memoria.
2. **`delkey`**:
    - Elimina la **clave de memoria compartida** del sistema, lo que significa que **nadie más podrá acceder a esa memoria** usando esa clave. Pero **no desacopla la memoria** ni actualiza la lista interna del programa. Si algún proceso todavía la usa, no se libera hasta que todos la desacoplen.

Diferencia clave:

- **`shared`** solo **desacopla la memoria** para el proceso, pero la clave y la memoria siguen existiendo en el sistema.
- **`delkey` elimina la clave del sistema**, pero no toca la memoria ni la lista de bloques locales del programa.

ADDR:
Podemos eliminar cualquier dirección que esté asignada (en la lista de bloques general)

-> allocate
******Lista de bloques asignados para el proceso 6535
0x705181b33000 12 Nov 23 12:57 ar.txt (descriptor 3)
0x705181b32000 12 Nov 23 12:58 ar.txt (descriptor 4)
0x705181b31000 12 Nov 23 13:55 ar.txt (descriptor 5)
0x5ed57684f320 32 Nov 23 13:55 malloc

-> deallocate 0x705181b33000
Memoria mapeada en 0x705181b33000, de tamaño 12, liberada y eliminada de la lista de bloques mmap

-> allocate
******Lista de bloques asignados para el proceso 6535
0x705181b32000 12 Nov 23 12:58 ar.txt (descriptor 4)
0x705181b31000 12 Nov 23 13:55 ar.txt (descriptor 5)
0x5ed57684f320 32 Nov 23 13:55 malloc

-> deallocate 0x5ed57684f320
Memoria mapeada en 0x5ed57684f320 , de tamaño 12, liberada y eliminada de la lista de bloques malloc

3-memfill: (mal el de referencia)
Memfill llena x bytes que se le pasan con cierto caracter (en hexadecimal) en una dirección de memoria.

-Para -mmap: los cambios que se apliquen con memfill se aplicarán en la memoria del proceso, pero NO se reflejarán en el archivo
-Para -shared: como varios procesos comparten la misma dirección de memoria, en este caso si se espera que los cambios se vean refeljados en el archivo

-> memfill
Uso: memfill addr cont ch

-> memfill 0x78
Error: La dirección 0x78 no está gestionada por el programa

-> allocate -mmap ar.txt wr
fichero ar.txt mapeado en 0x7645b9aa6000

-> memfill 0x7645b9aaa6001 8 -2
Error: La cantidad de bytes tiene que ser mayor que 0

-> memfill 0x7645b9aaa6001 8 0
Error: La dirección 0x7645b9aaa6001 no está gestionada por el programa

*En este caso al ser una dirección mapeda los cambios no se reflejarán sobre el archivo

-> allocate -mmap ar.txt wr

fichero ar.txt mapeado en 0x485a000

-> memfill 0x485a000 42

Uso: memfill addr cont ch

*idnica el uso correcto del comando

-> memfill 0x485a000 3 42

Llenados 3 bytes en la dirección 0x485a000 con el carácter '*' (0x2a)

*Hemos llenado 3 bytes de la dirección especificada con el caracter * que se corresponde con el numero decimal 42 en ASCII y que es 0x2a en hexadecimal.

Deberia poder mostrar los caracteres imprimibles (del 32 al 126)

4-memdump: (mal el de referencia)

Para ver el contenido de cont bytes de una dirección de memoria en la pantalla. Generalmente lo usamos para ver el contenido en una dirección en la que hemos hecho previamente memfill.

5-memory:

-memory -funcs: Imprime las direcciones de 3 funciones del programa y 3 funciones de biblioteca

-memory -vars: Imprime las direcciones de 3 variables externas, 3 externas inicializadas, 3 estáticas, 3 estáticas inicializadas y 3 variables automáticas.

-memory -blocks: Imprime la lista de bloques asignados.

-memory -all: Imprime todo lo anterior (-funcs, -vars y -blocks).

-memory -pmap: Muestra la salida del comando pmap para el proceso del shell (equivalente a vmmap en macOS).

Hemos hecho un malloc de 32 antes

->memory -a
Opcion -a no reconocida

-> memory -funcs
Funciones programa 0x56b14fa95978, 0x56b14fa95779, 0x56b14fa95881
Funciones libreria 0x7a21088600f0, 0x7a2108885b20, 0x7a21088ad640

-funciones programa: **Segmento de código (text segment)**.
Explicación: Estas son direcciones donde se almacenan las funciones definidas directamente en el código del programa. El segmento de código es de solo lectura y contiene las instrucciones compiladas del programa.

- **Ejemplo**: Si el programa tiene funciones como `main()`, `cmd_allocate()`, o `cmd_memfill()`, sus instrucciones estarán almacenadas en esta región.

-funciones biblioteca: **Segmento de bibliotecas compartidas (shared libraries)**.

- **Explicación**: Estas direcciones corresponden a funciones definidas en bibliotecas dinámicas utilizadas por el programa, como `printf()`, `malloc()`, o `execvp()`. Estas funciones se cargan en regiones específicas del espacio de direcciones reservadas para bibliotecas compartidas.

Razón de la diferencia en las direcciones: Las bibliotecas suelen estar cargadas en regiones de memoria virtual altas, separadas del código del programa, para aprovechar la reutilización de código entre procesos.

-> memory -blocks
******Lista de bloques asignados para el proceso 26746
0x56b14fbb7cd0 32 Nov 21 15:08 malloc

*Llama a la lista GENERAL de bloques de memoria asignados (malloc, mmap y shared)

-> memory -vars

```
Variables locales       0x7ffe7888fedc,    0x7ffe7888fed8,    0x7ffe7888fed4
Variables globales      0x56b14faa52f8,    0x56b14faa52f0,    0x56b14faa52f4
Var (N.I.)globales      0x56b14fab6e68,    0x56b14fab6e70,    0x56b14fab6e78
Variables staticas      0x56b14faa52fc,    0x56b14faa5300,    0x56b14faa5304
Var (N.I.)staticas      0x56b14faba760,    0x56b14faba768,    0x56b14faba770



-> memory -pmap
27971:   ./p2
000064c8090c9000      8K r---- p2
000064c8090cb000     36K r-x-- p2
000064c8090d4000     16K r---- p2
000064c8090d8000      4K r---- p2
000064c8090d9000      4K rw--- p2
000064c80a813000    132K rw---   [ anon ]
00007a8375aff000   1028K rw---   [ anon ]
00007a8375c00000    160K r---- libc.so.6
00007a8375c28000   1568K r-x-- libc.so.6
00007a8375db0000    316K r---- libc.so.6
00007a8375dff000     16K r---- libc.so.6
00007a8375e03000      8K rw--- libc.so.6
00007a8375e05000     52K rw---   [ anon ]
00007a8375e45000   1040K rw---   [ anon ]
00007a8375f5d000      8K rw---   [ anon ]
00007a8375f5f000      4K r---- ld-linux-x86-64.so.2
00007a8375f60000    172K r-x-- ld-linux-x86-64.so.2
00007a8375f8b000     40K r---- ld-linux-x86-64.so.2
00007a8375f95000      8K r---- ld-linux-x86-64.so.2
00007a8375f97000      8K rw--- ld-linux-x86-64.so.2
00007ffdad607000   2064K rw---   [ pila ]
00007ffdad8d3000     16K r----   [ anon ]
00007ffdad8d7000      8K r-x--   [ anon ]
ffffffffff600000      4K --x--   [ anon ]
 total            6720K


-> memory       (lo mismo que poner -all)
Variables locales       0x7ffe7888fedc,    0x7ffe7888fed8,    0x7ffe7888fed4
Variables globales      0x56b14faa52f8,    0x56b14faa52f0,    0x56b14faa52f4
Var (N.I.)globales      0x56b14fab6e68,    0x56b14fab6e70,    0x56b14fab6e78
Variables staticas      0x56b14faa52fc,    0x56b14faa5300,    0x56b14faa5304
Var (N.I.)staticas      0x56b14faba760,    0x56b14faba768,    0x56b14faba770
Funciones programa      0x56b14fa95978,    0x56b14fa95779,    0x56b14fa95881
Funciones libreria      0x7a21088600f0,    0x7a2108885b20,    0x7a21088ad640
******Lista de bloques asignados para el proceso 26746
     0x56b14fbb7cd0              32 Nov 21 15:08 malloc
```

*Muestra todo el contenido de -funcs, -blocks y -vars

- **Funciones (`-funcs`)**:

- Imprime las direcciones de funciones del programa (en la sección de código).
- Imprime direcciones de funciones de biblioteca estándar (en la memoria de las bibliotecas dinámicas cargadas).

- **Variables (`-vars`)**:

- **Locales**: En la **pila** (stack), ya que están definidas dentro de una función.
- **Globales inicializadas**: En la sección de datos inicializados (**data segment**).
- **Globales no inicializadas**: En el segmento **BSS**.
- **Estáticas inicializadas**: También en la sección de datos inicializados (**data segment**).
- **Estáticas no inicializadas**: En el segmento **BSS**.

- **Bloques (`-blocks`)**:

- Lista los bloques de memoria dinámicamente asignados (ej., con `malloc`, `mmap`).

- **Pmap (`-pmap`)**:

- Ejecuta un comando del sistema para mostrar el mapeo de memoria del proceso (dependiendo del sistema operativo).

*BBS: El **segmento BSS** (*Block Started by Symbol*) es una sección de memoria utilizada en los programas para almacenar **variables globales y estáticas no inicializadas** o inicializadas a 0. Es una parte del programa que no ocupa espacio en el archivo ejecutable, ya que se inicializa automáticamente con ceros en tiempo de ejecución.

6-readfile: lee x bytes de un archivo que se le pasa en una determinada dirección de memoria. Si no se pasa el tamaño se lee el archivo completo.

-> readfile
faltan parametros
Uso: readfile fich addr cont

-> readfile ar.txt
faltan parametros

-> allocate -mmap ar.txt
fichero ar.txt mapeado en 0x77a0420a5000

-> readfile ar.txt 0x77a0420a5000 12
Imposible leer fichero: Bad address
*NO SE LEE A SI MISMO

Imaginemos que tenemos un fichero hola.txt sin nada:
-> readfile hola.txt 0x70392d598000 8
Leídos 0 bytes de hola.txt en 0x77a0420a5000

*Le hemos dicho que lea 8 bytes del archivo hola.txt que tiene 0 bytes, en la dirección 0x77a0420a5000 (donde está el mapeo de ar.txt). Por tanto como tiene 0 bytes, se leen 0 bytes en esa dirección.

-> allocate -mmap ar.txt wr
fichero ar.txt mapeado en 0x79e70949c000

-> readfile ar.txt 0x79e8 9
Imposible leer fichero: Bad address

*Esto quiere decir que no existe la dirección. No está mapeada

-> readfile ar.txt 0x79e70949c000 18
leidos 12 bytes de ar.txt en 0x79e70949c000

Si no se especifica tamaño o el tamaño es mayor que el del archivo, se lee entero

Lectura entre ellos:
-> allocate -mmap ar.txt wr
fichero ar.txt mapeado en 0x75b8b94a1000

-> allocate -mmap ejemplo.txt wr
fichero ejemplo.txt mapeado en 0x75b8b94a0000

-> readfile ar.txt 0x75b8b94a0000 12
leidos 12 bytes de ar.txt en 0x75b8b94a0000

-> readfile ar.txt 0x75b8b94a1000 5
leidos 5 bytes de ar.txt en 0x75b8b94a1000

-> readfile ejemplo.txt 0x75b8b94a1000 8
leidos 8 bytes de ejemplo.txt en 0x75b8b94a1000

7-writefile: [-o] fichero addr cont

- Sin `-o`: Crea un archivo nuevo y copia datos de la memoria a él. Falla si el archivo ya existe.
- Con `-o`: Sobrescribe el archivo existente o lo crea si no existe, con los datos proporcionados. (sobreescribe cont del addr en fichero)

Imaginemos que **ar.txt** ya existe y contiene Hola mundo! y **ejemplo.txt** también existe y contiene HOLA MUNDO!

-> writefile

faltan parametros

Uso: writefile [-o] fiche addr cont

-Sin -o:

-> allocate -mmap ar.txt wr
fichero ar.txt mapeado en 0x485a000

-> writefile hola.txt 0x485a000 12
escritos 12 bytes en hola.txt desde 0x485a000
*hola.txt no existía. Se crea con el contenido de ar.txt (Hola mundo!)

-> writefile hola.txt 0x485a000 12
so: writefile [-o] fichero addr cont
*Si no especificamos los bytes no funcionará

-Con -o (otro shell):

-> allocate -mmap ar.txt wr
fichero ar.txt mapeado en 0x7c22d85cd000

-> allocate -mmap ejemplo.txt wr
fichero ejemplo.txt mapeado en 0x7c22d85cc000

-> writefile -o ejemplo.txt 0x7c22d85cd000 10
Escritos 10 bytes en ejemplo.txt desde 0x7c22d85cd000
*El resultado en ejemplo.txt es Hola mundi¡o (falta ! porque no llega a los 10 bytes, ar.txt son12 bytes)

-Otro ejemplo con todo

-> allocate -mmap ar.txt w
fichero ar.txt mapeado en 0x70392d597000

-> writefile 1.txt 0x70392d597000 5
Escritos 5 bytes en 1.txt desde 0x70392d597000

-> writefile -o 0x70392d597000 12
faltan parametros
Uso: writefile [-o] fich addr cont

-> writefile -o 1.txt 0x70392d597000 12
Escritos 12 bytes en 1.txt desde 0x70392d597000

8-read: leemos x bytes de un archivo en una dirección de memoria
read df addr x

-> allocate -mmap ar.txt wr
fichero ar.txt mapeado en 0x74af87610000

-> allocate -mmap ejemplo.txt wr

fichero ejemplo.txt mapeado en 0x74af8760f000

-> read 3 0x74af8760f000 5
Leidos 5 bytes del descriptor 3 en la direccion 0x74af8760f000

-> read 3 0x74af87610000 5
Leidos 5 bytes del descriptor 3 en la direccion 0x74af87610000

9-write:

10-recurse: Ejecuta la función recursiva n veces. Esta función tiene un arreglo automático de tamaño 2048 y un arreglo estático de tamaño 2048. Imprime las direcciones de ambos arreglos y su parámetro (así como el número de recursiones) antes de llamarse a sí misma.

-> recurse
Uso: recurse n

-> recurse 1 (*solo se imprimirán dos líneas, n=1 y n=0)
parametro: 1(0x7fff5d444a0c) array 0x7fff5d444a10, arr estatico 0x64b6b2c4b960
parametro: 0(0x7fff5d4439ec) array 0x7fff5d4439f0, arr estatico 0x64b6b2c4b960

(1) (2) (3) (4) (5)

1-parametro: inidcará el valor del parámetro en cada llamada
2- indica el número de mayor a menor hasta 0 en cada llamada. Si ponemos n=x irá desde x a 0
3- Es la dirección que indica donde está almacenado n en la memoria del programa.
4-**Array automático:** es una variable local creada en cada llamada de la función. Su dirección CAMBIA en cada llamada, porque cada vez que se llama a la función, el programa reserva un nuevo espacio en el stack para las variables globales. Al igual qye con n la dirección de automático DISMINUYE PROGRESIVAMENTE, esto refleja como el stack crece "hacia abajo".
Se crean y de destruyen en cada llamada, con diferentes direcciones.
5-Array estático: esta variable es estática, lo que significa que se guarda siempre en mismo lugar de la memoria, sin importar cuantas veces se llame a la función.

¿Por qué cambian las direcciones de `n` y `array`?

Las direcciones de `n` y `array` cambian porque cada vez que llamas a la función:

1. **El programa reserva espacio en el stack** para las variables locales de esa llamada.
    - En el stack, la memoria se organiza "de arriba hacia abajo". Cada nueva llamada ocupa un espacio más bajo en el stack, por eso las direcciones disminuyen.

-> recurse 2
parametro: 2(0x7fff5d444a0c) array 0x7fff5d444a10, arr estatico 0x64b6b2c4b960
parametro: 1(0x7fff5d4439ec) array 0x7fff5d4439f0, arr estatico 0x64b6b2c4b960
parametro: 0(0x7fff5d4429cc) array 0x7fff5d4429d0, arr estatico 0x64b6b2c4b960

-> recurse 3
parametro: 3(0x7fff5d444a0c) array 0x7fff5d444a10, arr estatico 0x64b6b2c4b960
parametro: 2(0x7fff5d4439ec) array 0x7fff5d4439f0, arr estatico 0x64b6b2c4b960
parametro: 1(0x7fff5d4429cc) array 0x7fff5d4429d0, arr estatico 0x64b6b2c4b960
parametro: 0(0x7fff5d4419ac) array 0x7fff5d4419b0, arr estatico 0x64b6b2c4b960

Organización de la memoria de un proceso

En un sistema de 64 bits, el espacio de direcciones tiene un rango amplio (hasta 264 direcciones posibles). El esquema típico para un proceso es el siguiente:

1. **Texto (Código)**: Contiene las instrucciones del programa.
    - Direcciones más bajas, típicamente comenzando en un rango como `0x00400000`.
    - Ejemplo: La dirección donde se almacena el código de la función `Recursiva`.
2. **Datos estáticos y globales**: Almacena variables globales y estáticas inicializadas o no inicializadas.
    - Rango típico: `0x601000` (puede variar según el sistema y el programa).
    - Ejemplo: La variable estática `estatico` pertenece a esta región.
3. **Heap**: Almacena memoria dinámica asignada con funciones como `malloc` o `new`.
    - Crece hacia direcciones altas.
    - Ejemplo: Memoria reservada explícitamente por el programa.
4. **Stack (Pila)**: Utilizado para almacenar variables locales, parámetros y direcciones de retorno.
    - Rango típico: Comienza en direcciones altas (ej., `0x7ff...`) y decrece con cada llamada.
    - Ejemplo: Las variables locales `automatico` y el parámetro `n` pertenecen al stack.

IMPORTANTE:

Cómo interpretar los rangos?

Ejemplo típico (de un proceso en Linux de 64 bits):

- **Código del programa (texto):** `0x00400000` a `0x00500000`.
- **Datos estáticos:** `0x601000` a `0x602000`.
- **Heap:** Comienza cerca de `0x602000` y crece hacia arriba.
- **Stack:** Comienza cerca de `0x7ffc00000000` y decrece.

FUNCIONES P3-> procesos

Funciones del código de ayuda

- **Cmd_fork** -> **fork** (relacionado con la creación de procesos)
- **BuscarVariable** -> **showvar** (para mostrar variables de entorno)
- **CambiarVariable** -> **changevar** (para cambiar variables de entorno)
- **Ejecutable** -> **exec** (para ejecutar programas)
- **Execpve** -> **exec**, **execpri** (para ejecutar programas con o sin cambio de entorno y prioridad)
- **ValorSenal** -> No tiene una relación directa con los comandos del PDF, pero está relacionado con señales en el sistema operativo.
- **NombreSenal** -> Similar a **ValorSenal**, también relacionado con señales en el sistema operativo.

1. ¿Qué es un proceso en Linux?

Un proceso es una instancia en ejecución de un programa. Representa un programa que ha sido cargado en memoria y está siendo ejecutado por la CPU.

Cada proceso tiene:

- **PID (Process ID):** Un identificador único.
- **PPID (Parent Process ID):** El identificador de su proceso padre.
- **Estado:** Puede ser ejecutable, detenido, terminado, entre otros.
- **Espacio de memoria propio:** Incluye:
    - Código (instrucciones del programa).
    - Datos (variables globales y estáticas).
    - Pila (stack) para llamadas a funciones.
    - Heap para memoria dinámica.
- **Contexto de ejecución:** Incluye los registros de la CPU y otras informaciones necesarias para reanudar el proceso.

---

2. Estados de un proceso

Un proceso en Linux puede encontrarse en uno de varios estados:

1. **Running (Ejecutando):** Está usando la CPU activamente.
2. **Ready (Listo):** Está preparado para ejecutarse, pero la CPU está ocupada con otro proceso.
3. **Blocked (Bloqueado):** Está esperando un recurso o una operación de entrada/salida.
4. **Terminated (Terminado):** Ha finalizado su ejecución, pero su información aún puede ser retenida por el sistema.
5. **Zombie:** El proceso ha terminado, pero su padre no ha leído su estado de salida.

El estado de un proceso se puede observar usando comandos como `ps` o `top`.

---

3. Creación de procesos

En Linux, los procesos se crean a partir de otros procesos, formando una jerarquía. La llamada al sistema clave aquí es `fork()`.

- **`fork()`**: Crea un proceso hijo duplicando al padre. El hijo recibe una copia del espacio de memoria del padre.
    - El proceso hijo tiene un nuevo PID pero comparte recursos como archivos abiertos con el padre.
    - `fork()` devuelve 0 al proceso hijo y el PID del hijo al proceso padre.

Ejemplo de jerarquía:

- Proceso inicial (`PID=1`): Es el proceso *init/systemd* (el primer proceso del sistema).
- Otros procesos son creados por *init* o sus descendientes.

---

4. Ejecución de programas

Cuando un proceso necesita ejecutar un programa diferente, utiliza la llamada al sistema `exec`.

- **`exec()`**: Sustituye la memoria del proceso actual con un nuevo programa.
    - No crea un nuevo proceso.
    - El PID no cambia, pero el contenido del proceso sí.
    - Existen variantes como `execl`, `execp`, y `execve` que permiten especificar argumentos y variables de entorno.

---

5. Terminación de procesos

Cuando un proceso termina, llama a `exit()`, liberando recursos asignados por el sistema operativo. Su estado es reportado al proceso padre mediante:

- **`wait()` o `waitpid()`:** Permiten al proceso padre recoger el estado de salida de un hijo.
- Si el padre no recoge este estado, el proceso hijo queda en estado *zombie*.

---

6. Planificación de procesos

Linux utiliza un planificador para decidir qué procesos se ejecutan en la CPU. Las prioridades determinan qué procesos reciben más tiempo de CPU:

- **Prioridad de usuario:** Rango de -20 (más alta) a 19 (más baja). Se puede ajustar con comandos como `nice` o `renice`.
- **Estados de planificación:** Se basa en factores como interactividad, uso de CPU reciente, y prioridades.

---

7. Procesos en primer y segundo plano

En un entorno multitarea, los procesos pueden ejecutarse de diferentes maneras:

- **Primer plano (Foreground):** El proceso tiene control exclusivo sobre el terminal. El usuario espera a que termine antes de ejecutar otros comandos.
- **Segundo plano (Background):** El proceso se ejecuta en paralelo, liberando el terminal para otras tareas. Esto se logra usando el operador `&`.

---

8. Señales y control de procesos

Linux permite que los procesos se comuniquen o sean controlados mediante señales.

- **Señales comunes:**
  - `SIGKILL`: Mata un proceso inmediatamente.
  - `SIGSTOP`: Detiene un proceso.
  - `SIGCONT`: Reanuda un proceso detenido.
- Las señales pueden ser enviadas con comandos como `kill` o `killall`.

---

9. Variables de entorno

Son pares clave-valor que un proceso hereda de su padre. Ejemplo:

- `PATH`: Contiene directorios donde buscar ejecutables.
- `HOME`: Directorio del usuario.

Las variables de entorno pueden ser accedidas o modificadas con funciones como `getenv()` y `putenv()`.

---

10. Seguridad y permisos de procesos

Los procesos tienen credenciales que determinan qué acciones pueden realizar:

- **Real UID:** ID del usuario que inició el proceso.
- **Efectivo UID:** ID que determina los permisos actuales.
- **Setuid bit:** Permite a un programa ejecutarse con permisos diferentes a los del usuario que lo inició (útil para herramientas como `sudo`).

---

11. Threads (Hilos) vs. Procesos

- **Procesos:** Tienen memoria y recursos independientes.
- **Hilos:** Son subunidades dentro de un proceso, que comparten memoria pero tienen su propio contexto de ejecución. Linux implementa hilos con la llamada al sistema `clone()`.

- CREDENCIALES:

1-getuid:
Obtiene y muestra dos tipos de **credenciales** de usuario del proceso que ejecuta el comando:

->getuid
Credencial real: 1000, (ismael)
Credencial efectiva: 1000, (ismael)

1-**Credencial real**: El ID del usuario con el que el proceso fue lanzado (realmente está ejecutando el proceso). -> getuid y getpwuid
2-Credencial efectiva: El ID de usuario con el que el proceso actúa actualmente, lo que puede ser diferente al real (por ejemplo, si se usó un `setuid` para cambiar temporalmente el usuario). -> geteuid y getpwuid

2-setuid: setuid [-l] id

Resumen de los cambios:

Acción	Antes del `setuid`	Después del `setuid`	Propietario del archivo
`setuid -l Ismael`	CR = 1002, CE = 1002	CR = 1002, CE = 1001	1001 (Ismael)
`setuid 1001`	CR = 1002, CE = 1002	CR = 1002, CE = 1001	1001 (Ismael)
`setuid 1002`	CR = 1002, CE = 1002	CR = 1002, CE = 1002	1001 (Ismael)
`setuid 1003` (no existe)	CR = 1002, CE = 1002	CR = 1002, CE = 1002	1001 (Ismael)

- VARIABLES DE ENTORNO:

Variables de entorno:
Son datos que el sistema guarda para que los programas sepan cómo comportarse. Ejemplo:

- `HOME`: Tu carpeta principal.
- `PATH`: Dónde buscar programas cuando los ejecutas.
- **¿Qué es el PATH?**
  Es una lista de carpetas donde el sistema busca los programas que usas en la terminal. Si un programa no está en el `PATH`, no lo puedes ejecutar sin escribir su ruta completa.
- Ver tu PATH: `echo $PATH`.
- **Buscar variables de entorno en C**:
  Tu programa `showvar` hace dos cosas:
- Si no pasas nada, muestra **todas las variables de entorno**.
- Si pasas nombres específicos (como `HOME` o `PATH`), busca su valor y lo muestra.

IMPORTANTE: **`extern char **environ;`**

Es una declaración que permite acceder a **todas las variables de entorno** del sistema desde un programa en C. `environ` es como una "lista global" de todas las variables de entorno, y puedes usarla para inspeccionarlas o procesarlas en tu programa.

3-showvar: Muestra el valor y las direcciones de las variables de entorno

-> showvar

0x7ffc3c05edd8->main arg3[0]=(0x7ffc3c060fc4) LESSOPEN=| /usr/bin/lesspipe %s
0x7ffc3c05ede0->main arg3[1]=(0x7ffc3c060fe4) LANGUAGE=es:gl:en_CA:en
0x7ffc3c05ede8->main arg3[2]=(0x7ffc3c060ffb) LC_TIME=gl_ES.UTF-8
0x7ffc3c05edf0->main arg3[3]=(0x7ffc3c06100c) USER=ismael
0x7ffc3c05edf8->main arg3[4]=(0x7ffc3c06101c) no_proxy=localhost,127.0.0.0/8,::1
0x7ffc3c05ee00->main arg3[5]=(0x7ffc3c061047) XDG_SESSION_TYPE=wayland
0x7ffc3c05ee08->main arg3[6]=(0x7ffc3c061055) SHLVL=1
0x7ffc3c05ee10->main arg3[7]=(0x7ffc3c06105c) HOME=/home/ismael
0x7ffc3c05ee18->main arg3[8]=(0x7ffc3c06107e)
CAML_LD_LIBRARY_PATH=/home/ismael/.opam/ocaml-latest/lib/stublibs:/home/
ismael/.opam/ocaml-latest/lib/ocaml/stublibs:/home/ismael/.opam/ocaml-latest/lib/ocaml
0x7ffc3c05ee20->main arg3[9]=(0x7ffc3c061111) NO_PROXY=localhost,127.0.0.0/8,::1
0x7ffc3c05ee28->main arg3[10]=(0x7ffc3c06113b) DESKTOP_SESSION=ubuntu
0x7ffc3c05ee30->main arg3[11]=(0x7ffc3c061156)
OCAML_TOPLEVEL_PATH=/home/ismael/.opam/ocaml-latest/lib/toplevel
0x7ffc3c05ee38->main arg3[12]=(0x7ffc3c06119c) GNOME_SHELL_SESSION_MODE=ubuntu
0x7ffc3c05ee40->main arg3[13]=(0x7ffc3c0611af) GTK_MODULES=gail:atk-bridge
0x7ffc3c05ee48->main arg3[14]=(0x7ffc3c0611cb) LC_MONETARY=gl_ES.UTF-8
0x7ffc3c05ee50->main arg3[15]=(0x7ffc3c0611e1) MAKEFLAGS=

0x7ffc3c05ee58->main arg3[16]=(0x7ffc3c0611fb)
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus,guid=94fdfb362008d6b3353640066752e316
0x7ffc3c05ee60->main arg3[17]=(0x7ffc3c06124f) SYSTEMD_EXEC_PID=2408
0x7ffc3c05ee68->main arg3[18]=(0x7ffc3c06126a) DBUS_STARTER_BUS_TYPE=session
0x7ffc3c05ee70->main arg3[19]=(0x7ffc3c061286) LIBVIRT_DEFAULT_URI=qemu:///system
0x7ffc3c05ee78->main arg3[20]=(0x7ffc3c06129f) COLORTERM=truecolor
0x7ffc3c05ee80->main arg3[21]=(0x7ffc3c0612b9)
DEBUGINFOD_URLS=https://debuginfod.ubuntu.com
0x7ffc3c05ee88->main arg3[22]=(0x7ffc3c0612e5) MAKE_TERMERR=/dev/pts/1
0x7ffc3c05ee90->main arg3[23]=(0x7ffc3c061300) WAYLAND_DISPLAY=wayland-0
0x7ffc3c05ee98->main arg3[24]=(0x7ffc3c06131a) IM_CONFIG_PHASE=1
0x7ffc3c05eea0->main arg3[25]=(0x7ffc3c061324) LOGNAME=ismael
0x7ffc3c05eea8->main arg3[26]=(0x7ffc3c06132d) _=/usr/bin/make
0x7ffc3c05eeb0->main arg3[27]=(0x7ffc3c061351)
MEMORY_PRESSURE_WATCH=/sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/app.slice/app-gnome\x2dsession\x2dmanager.slice/gnome-session-manager@ubuntu.service/memory.pressure
0x7ffc3c05eeb8->main arg3[28]=(0x7ffc3c061404) XDG_SESSION_CLASS=user
0x7ffc3c05eec0->main arg3[29]=(0x7ffc3c061412) USERNAME=ismael
0x7ffc3c05eec8->main arg3[30]=(0x7ffc3c06141e) TERM=xterm-256color
0x7ffc3c05eed0->main arg3[31]=(0x7ffc3c061446) GNOME_DESKTOP_SESSION_ID=this-is-deprecated
0x7ffc3c05eed8->main arg3[32]=(0x7ffc3c06145e)
PATH=/home/ismael/.opam/ocaml-latest/bin:/home/ismael/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin
0x7ffc3c05eee0->main arg3[33]=(0x7ffc3c061517)
SESSION_MANAGER=local/asus:@/tmp/.ICE-unix/2408,unix/asus:/tmp/.ICE-unix/2408
0x7ffc3c05eee8->main arg3[34]=(0x7ffc3c06155f) PAPERSIZE=a4
0x7ffc3c05eef0->main arg3[35]=(0x7ffc3c06156d) LC_ADDRESS=gl_ES.UTF-8
0x7ffc3c05eef8->main arg3[36]=(0x7ffc3c06158f)
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/b9a8122a_affc_4b87_a21d_dc2439cbc820
0x7ffc3c05ef00->main arg3[37]=(0x7ffc3c0615e3) GNOME_SETUP_DISPLAY=:1
0x7ffc3c05ef08->main arg3[38]=(0x7ffc3c0615f6) XDG_RUNTIME_DIR=/run/user/1000
0x7ffc3c05ef10->main arg3[39]=(0x7ffc3c061615) XDG_MENU_PREFIX=gnome-
0x7ffc3c05ef18->main arg3[40]=(0x7ffc3c061626) MAKELEVEL=1
0x7ffc3c05ef20->main arg3[41]=(0x7ffc3c061630) DISPLAY=:0
0x7ffc3c05ef28->main arg3[42]=(0x7ffc3c061638) LANG=es_ES.UTF-8
0x7ffc3c05ef30->main arg3[43]=(0x7ffc3c061651) LC_TELEPHONE=gl_ES.UTF-8
0x7ffc3c05ef38->main arg3[44]=(0x7ffc3c061671)
XDG_CURRENT_DESKTOP=ubuntu:GNOME
0x7ffc3c05ef40->main arg3[45]=(0x7ffc3c061689) XAUTHORITY=/run/user/1000/.mutter-Xwaylandauth.KSGQY2
0x7ffc3c05ef48->main arg3[46]=(0x7ffc3c0616cb) GNOME_TERMINAL_SERVICE=:1.165
0x7ffc3c05ef50->main arg3[47]=(0x7ffc3c0616e6) XDG_SESSION_DESKTOP=ubuntu
0x7ffc3c05ef58->main arg3[48]=(0x7ffc3c0616f8) XMODIFIERS=@im=ibus
0x7ffc3c05ef60->main arg3[49]=(0x7ffc3c06170b)
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.dz=01;31:*.

gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.webp=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
0x7ffc3c05ef68->main arg3[50]=(0x7ffc3c061d03) SSH_AGENT_LAUNCHER=openssh
0x7ffc3c05ef70->main arg3[51]=(0x7ffc3c061d19)
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
0x7ffc3c05ef78->main arg3[52]=(0x7ffc3c061d3c) LC_NAME=gl_ES.UTF-8
0x7ffc3c05ef80->main arg3[53]=(0x7ffc3c061d4e) SHELL=/bin/bash
0x7ffc3c05ef88->main arg3[54]=(0x7ffc3c061d69) QT_ACCESSIBILITY=1
0x7ffc3c05ef90->main arg3[55]=(0x7ffc3c061d78) MAKE_TERMOUT=/dev/pts/1
0x7ffc3c05ef98->main arg3[56]=(0x7ffc3c061d8e) GDMSESSION=ubuntu
0x7ffc3c05efa0->main arg3[57]=(0x7ffc3c061d9f) LESSCLOSE=/usr/bin/lesspipe %s %s
0x7ffc3c05efa8->main arg3[58]=(0x7ffc3c061dc6) LC_MEASUREMENT=gl_ES.UTF-8
0x7ffc3c05efb0->main arg3[59]=(0x7ffc3c061de5)
OPAM_SWITCH_PREFIX=/home/ismael/.opam/ocaml-latest
0x7ffc3c05efb8->main arg3[60]=(0x7ffc3c061e17) LC_IDENTIFICATION=gl_ES.UTF-8
0x7ffc3c05efc0->main arg3[61]=(0x7ffc3c061e30) QT_IM_MODULE=ibus
0x7ffc3c05efc8->main arg3[62]=(0x7ffc3c061e39)
PWD=/home/ismael/Escritorio/SO/PRÁCTICAS/P3
0x7ffc3c05efd0->main arg3[63]=(0x7ffc3c061e70)
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/usr/share/:/var/lib/snapd/desktop
0x7ffc3c05efd8->main arg3[64]=(0x7ffc3c061edd)
DBUS_STARTER_ADDRESS=unix:path=/run/user/1000/bus,guid=94fdfb362008d6b3353640066752e316
0x7ffc3c05efe0->main arg3[65]=(0x7ffc3c061f30)
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
0x7ffc3c05efe8->main arg3[66]=(0x7ffc3c061f58) LC_NUMERIC=gl_ES.UTF-8
0x7ffc3c05eff0->main arg3[67]=(0x7ffc3c061f6b) MFLAGS=
0x7ffc3c05eff8->main arg3[68]=(0x7ffc3c061f75) LC_PAPER=gl_ES.UTF-8
0x7ffc3c05f000->main arg3[69]=(0x7ffc3c061f97)
MEMORY_PRESSURE_WRITE=c29tZSAyMDAwMDAgMjAwMDAwMAA=
0x7ffc3c05f008->main arg3[70]=(0x7ffc3c061fbc) MANPATH=:/home/ismael/.opam/ocaml-latest/man
0x7ffc3c05f010->main arg3[71]=(0x7ffc3c061fed) VTE_VERSION=7600

MUESTRAS TODAS LAS VARIABLES DE ENTORNO. Todos los nombres en mayúsculas que aparecen después de una dirección de memoria son las variables de entorno. Aparecen en pares de clave.valor, de manera que para cada variable de entonro muestra a su lado su contenido

1. Por ejemplo:
    - `SHELL=/bin/bash`: El valor de la variable `SHELL` es `/bin/bash`, indicando que el shell predeterminado es Bash.
    - `SESSION_MANAGER=local/asus:@/tmp/.ICE-unix/2408,...`: Indica información sobre el manejador de la sesión gráfica.
    - `PATH=/home/ismael/.opam/ocaml-latest/bin:/usr/bin:/bin,...`: Lista de directorios separados por `:` donde el sistema buscará ejecutables. Esto incluye varios directorios como `/usr/bin`, `/bin`, y otros específicos del entorno de desarrollo (`/home/ismael/.opam/ocaml-latest/bin`).

Ejemplos de variables de entorno comunes en la salida:

- **`SHELL=/bin/bash`**: Define qué shell está utilizando el sistema.
- **`SESSION_MANAGER=local/asus:@/tmp/.ICE-unix/2408,...`**: Información sobre la sesión gráfica.
- **`LANG=es_ES.UTF-8`**: El idioma y la codificación del sistema (en este caso, español de España con codificación UTF-8).
- **`PATH=...`**: Una lista de directorios donde el sistema busca programas ejecutables.
- **`HOME=/home/ismael`**: Directorio home del usuario.
- **`USER=ismael`**: Nombre del usuario en el sistema.
- **`TERM=xterm-256color`**: El tipo de terminal que está utilizando (en este caso, un terminal compatible con 256 colores).

Resumen:

Estas son **variables de entorno** que el sistema y los programas utilizan para almacenar configuraciones o información importante sobre el entorno de ejecución. Puedes acceder a ellas desde programas (por ejemplo, con `getenv()` en C) o desde el shell para modificar el comportamiento de la terminal o las aplicaciones.

Algunas variables comunes que puedes ver aquí son:

- **`SHELL`**: Qué shell se está usando.
- **`PATH`**: Directorios donde buscar programas ejecutables.
- **`HOME`**: El directorio principal del usuario.
- **`USER`**: El nombre del usuario que está ejecutando el proceso.
- **`LANG`**: El idioma y la codificación.

-> showvar SHELL
Con arg3 main SHELL=/bin/bash(0x7ffc3c061d4f) @0x7ffc3c05ef80
  Con environ SHELL=/bin/bash(0x7ffc3c061d4f) @0x7ffc3c05ef80
   Con getenv /bin/bash(0x7ffc3c061d4f)

-> showvar PATH
Con arg3 main PATH=/home/ismael/.opam/ocaml-latest/bin:/home/ismael/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin(0x7ffc3c06145f) @0x7ffc3c05eed8
  Con environ PATH=/home/ismael/.opam/ocaml-latest/bin:/home/ismael/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin(0x7ffc3c06145f) @0x7ffc3c05eed8
   Con getenv /home/ismael/.opam/ocaml-latest/bin:/home/ismael/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin(0x7ffc3c06145f)

La variable **PATH** es crucial, ya que indica al sistema dónde buscar los programas ejecutables. Contiene una lista de directorios separados por dos puntos (`:`), y cuando ejecutas un comando, el sistema buscará ese comando en esos directorios en el orden que están listados.

Ejemplo de la salida general:
-> showvar

0x7ffd929a52d8->main arg3[0]=(0x7ffd929a601e) SHELL=/bin/bash

(1) (2) (3) (4)

1-Es la dirección de memoria donde se encuentra el puntero que apunta al primer elemento de `arg3[]`, que es el arreglo de argumentos pasados al programa.

2-Hace referencia al primer argumento pasado al programa. `arg3` es el nombre que el código utiliza para referirse al arreglo de argumentos del `main`.

3-Es la dirección de memoria donde se encuentra almacenada la cadena de texto que contiene el valor de `arg3[0]`.

4-NOMBRE = VALOR (nombre de la variable de entorno = valor asignado a esa variable).

Ejemplo salida variable/s específica/s:

-> showvar _

ismael@asus(~/Escritorio/SO/PRÁCTICAS/SHELL_REFERENCIA/0LINUX_P3)-> showvar _

Con arg3 main _=./shell.out(0x7ffd929a6fde) @0x7ffd929a54f0

Con environ _=./shell.out(0x7ffd929a6fde) @0x7ffd929a54f0

Con getenv ./shell.out(0x7ffd929a6fe0)

(1) (2) (3) (4)

1-Tipo que usamos para acceder a la variables

2-NOMBRE = VALOR

3-Corresponden a la dirección del **valor** de la variable en memoria,

4-corresponden a la **dirección de la entrada** de la variable en la tabla de entorno o estructura en memoria

4-changevar: changevar [-a|-e|-p] var valor: cambia el valor de una variable de entorno
Tiene las flags: -a (para cambiar por le tercer argumento del main)
-e (para cambiar mediante environ)
-p (para CREAR una nueva variable mediante la funcion putenv -> stdlib.h)

Si hacemos un showvar general o de una variable podemos ver su contenido. Vamos a cambiar el contenido de la variable de entorno USER

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> showvar USER
Con arg3 main USER=ismael(0x7ffe1b5f6052) @0x7ffe1b5f5b28
Con environ USER=ismael(0x7ffe1b5f6052) @0x7ffe1b5f5b28
Con getenv USER=ismael(0x7ffe1b5f6052)

-> ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> changevar -a USER user1
Variable 'USER' cambiada a 'user1' en envp

-> ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> showvar USER
Con arg3 main USER=user1(0x7ffdb5a52052) @0x7ffdb5a50588
Con environ USER=user1(0x7ffdb5a52052) @0x7ffdb5a50588
Con getenv USER=user1(0x7ffdb5a52052)

-> ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> changevar -e USER ismael1
Variable 'USER' cambiada a 'ismael1' en environ

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> showvar USER
Con arg3 main USER=ismael1(0x5b642134d215) @0x7ffe1b5f5b28
  Con environ USER=ismael1(0x5b642134d215) @0x7ffe1b5f5b28
   Con getenv USER=ismael1(0x5b642134d215)

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> changevar -p USER2 ismael
Variable 'USER2' creada o cambiada a 'ismael' mediante putenv

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> showvar USER2
Con arg3 main USER2=ismael(0x5b642134d236) @0x5b642134d498
  Con environ USER2=ismael(0x5b642134d236) @0x5b642134d498
   Con getenv USER2=ismael(0x5b642134d236)

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> changevar USER user1
Uso: changevar [-a|-e|-p] var valor

FALTA PASAR UN ARGUMENTO (-a, -e, -p)

5-subsvar: subsvar [-a|-e] var1 var2 valor: sustituye la variable de entorno var1 con var2=valor

1-Cambiamos la variable de entorno USER por USER2 con el contenido isma
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> subsvar -a USER USER2 isma
Variable 'USER' sustituida por 'USER2=isma' en el entorno local (envp)

2-Si ahora buscamos la variable de entorno USER no aparecerá ya que la hemos cambiado y por tanto se ha eliminado
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> showvar USER
   Con getenv: Variable no encontrada

3-Si ahora buscamos showvar USER2 nos indica sus direcciones y su contenido, ya que ahora existe esta variable.
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> showvar USER2
Con arg3 main USER2=isma(0x57a6fed12216) @0x7ffdd9581858
  Con environ USER2=isma(0x57a6fed12216) @0x7ffdd9581858
   Con getenv USER2=isma(0x57a6fed12216)

6-environ: environ [-environ | -addr]: Muestra el entorno del proceso
environ solo hace lo mismo que showvar solo. Muestran las variables de entorno y sus direcciones mediante el tercer argumento del main arg3 -> envp[]

-environ: muestra lo mismo que environ y showvar normal pero accediendo desde environ en este caso -> environ []
-addr: muestra el valor y donde se almacenan environ y el 3er arg main (muestra sus direcciones de memoria).

1-Environ sin argumentos:
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> environ

0x0x7ffdeb77cfa8->main arg3[0]=(0x7ffdeb77dfeb) LESSOPEN=| /usr/bin/lesspipe %s

0x0x7ffdeb77cfb0->main arg3[1]=(0x7ffdeb77e00b) MAIL=/var/mail/ismael

0x0x7ffdeb77cfb8->main arg3[2]=(0x7ffdeb77e021) LANGUAGE=es:gl:en_CA:en

0x0x7ffdeb77cfc0->main arg3[3]=(0x7ffdeb77e039) LC_TIME=gl_ES.UTF-8

0x0x7ffdeb77cfc8->main arg3[4]=(0x7ffdeb77e04d) USER=ismael

0x0x7ffdeb77cfd0->main arg3[5]=(0x7ffdeb77e059) no_proxy=localhost,127.0.0.0/8,::1

0x0x7ffdeb77cfd8->main arg3[6]=(0x7ffdeb77e07c) XDG_SESSION_TYPE=wayland

0x0x7ffdeb77cfe0->main arg3[7]=(0x7ffdeb77e095) SHLVL=3

0x0x7ffdeb77cfe8->main arg3[8]=(0x7ffdeb77e09d) HOME=/home/ismael

0x0x7ffdeb77cff0->main arg3[9]=(0x7ffdeb77e0af) CAML_LD_LIBRARY_PATH=/home/ismael/.opam/ocaml-latest/lib/stublibs:/home/ismael/.opam/ocaml-latest/lib/ocaml/stublibs:/home/ismael/.opam/ocaml-latest/lib/ocaml

0x0x7ffdeb77cff8->main arg3[10]=(0x7ffdeb77e14e) NO_PROXY=localhost,127.0.0.0/8,::1

0x0x7ffdeb77d000->main arg3[11]=(0x7ffdeb77e171) DESKTOP_SESSION=ubuntu

0x0x7ffdeb77d008->main arg3[12]=(0x7ffdeb77e188) OCAML_TOPLEVEL_PATH=/home/ismael/.opam/ocaml-latest/lib/toplevel

0x0x7ffdeb77d010->main arg3[13]=(0x7ffdeb77e1c9) GNOME_SHELL_SESSION_MODE=ubuntu

0x0x7ffdeb77d018->main arg3[14]=(0x7ffdeb77e1e9) GTK_MODULES=gail:atk-bridge

0x0x7ffdeb77d020->main arg3[15]=(0x7ffdeb77e205) LC_MONETARY=gl_ES.UTF-8

0x0x7ffdeb77d028->main arg3[16]=(0x7ffdeb77e21d) MAKEFLAGS=

0x0x7ffdeb77d030->main arg3[17]=(0x7ffdeb77e228) DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus,guid=8c45d9b2e512cee3459f36c46759d61b

0x0x7ffdeb77d038->main arg3[18]=(0x7ffdeb77e284) SYSTEMD_EXEC_PID=2354

0x0x7ffdeb77d040->main arg3[19]=(0x7ffdeb77e29a) DBUS_STARTER_BUS_TYPE=session

0x0x7ffdeb77d048->main arg3[20]=(0x7ffdeb77e2b8) LIBVIRT_DEFAULT_URI=qemu:///system

0x0x7ffdeb77d050->main arg3[21]=(0x7ffdeb77e2db) COLORTERM=truecolor

0x0x7ffdeb77d058->main arg3[22]=(0x7ffdeb77e2ef) DEBUGINFOD_URLS=https://debuginfod.ubuntu.com

0x0x7ffdeb77d060->main arg3[23]=(0x7ffdeb77e31e) MAKE_TERMERR=/dev/pts/1

0x0x7ffdeb77d068->main arg3[24]=(0x7ffdeb77e336) WAYLAND_DISPLAY=wayland-0

0x0x7ffdeb77d070->main arg3[25]=(0x7ffdeb77e350) IM_CONFIG_PHASE=1

0x0x7ffdeb77d078->main arg3[26]=(0x7ffdeb77e362) LOGNAME=ismael

0x0x7ffdeb77d080->main arg3[27]=(0x7ffdeb77e371) _=/usr/bin/make

0x0x7ffdeb77d088->main arg3[28]=(0x7ffdeb77e381) MEMORY_PRESSURE_WATCH=/sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/app.slice/app-gnome\x2dsession\x2dmanager.slice/gnome-session-manager@ubuntu.service/memory.pressure

0x0x7ffdeb77d090->main arg3[29]=(0x7ffdeb77e438) XDG_SESSION_CLASS=user

0x0x7ffdeb77d098->main arg3[30]=(0x7ffdeb77e44f) USERNAME=ismael

0x0x7ffdeb77d0a0->main arg3[31]=(0x7ffdeb77e45f) TERM=xterm-256color

0x0x7ffdeb77d0a8->main arg3[32]=(0x7ffdeb77e473) GNOME_DESKTOP_SESSION_ID=this-is-deprecated

0x0x7ffdeb77d0b0->main arg3[33]=(0x7ffdeb77e49f) PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin

0x0x7ffdeb77d0b8->main arg3[34]=(0x7ffdeb77e507) SESSION_MANAGER=local/asus:@/tmp/.ICE-unix/2354,unix/asus:/tmp/.ICE-unix/2354

0x0x7ffdeb77d0c0->main arg3[35]=(0x7ffdeb77e555) PAPERSIZE=a4

0x0x7ffdeb77d0c8->main arg3[36]=(0x7ffdeb77e562) LC_ADDRESS=gl_ES.UTF-8

0x0x7ffdeb77d0d0->main arg3[37]=(0x7ffdeb77e579) GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/62241c1e_6309_4402_912b_571f93346219

0x0x7ffdeb77d0d8->main arg3[38]=(0x7ffdeb77e5cf) GNOME_SETUP_DISPLAY=:1

0x0x7ffdeb77d0e0->main arg3[39]=(0x7ffdeb77e5e6) XDG_RUNTIME_DIR=/run/user/1000

0x0x7ffdeb77d0e8->main arg3[40]=(0x7ffdeb77e605) XDG_MENU_PREFIX=gnome-

0x0x7ffdeb77d0f0->main arg3[41]=(0x7ffdeb77e61c) MAKELEVEL=1

0x0x7ffdeb77d0f8->main arg3[42]=(0x7ffdeb77e628) DISPLAY=:0

0x0x7ffdeb77d100->main arg3[43]=(0x7ffdeb77e633) LANG=gl_ES.UTF-8

0x0x7ffdeb77d108->main arg3[44]=(0x7ffdeb77e644) LC_TELEPHONE=gl_ES.UTF-8

0x0x7ffdeb77d110->main arg3[45]=(0x7ffdeb77e65d) XDG_CURRENT_DESKTOP=ubuntu:GNOME

0x0x7ffdeb77d118->main arg3[46]=(0x7ffdeb77e67e) XAUTHORITY=/run/user/1000/.mutter-Xwaylandauth.B0DKY2

0x0x7ffdeb77d120->main arg3[47]=(0x7ffdeb77e6b4) GNOME_TERMINAL_SERVICE=:1.137

0x0x7ffdeb77d128->main arg3[48]=(0x7ffdeb77e6d2) XDG_SESSION_DESKTOP=ubuntu

0x0x7ffdeb77d130->main arg3[49]=(0x7ffdeb77e6ed) XMODIFIERS=@im=ibus

0x0x7ffdeb77d138->main arg3[50]=(0x7ffdeb77e701) LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.webp=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:

0x0x7ffdeb77d140->main arg3[51]=(0x7ffdeb77ecf0) SSH_AGENT_LAUNCHER=openssh

0x0x7ffdeb77d148->main arg3[52]=(0x7ffdeb77ed0b) SSH_AUTH_SOCK=/run/user/1000/keyring/ssh

0x0x7ffdeb77d150->main arg3[53]=(0x7ffdeb77ed34) LC_NAME=gl_ES.UTF-8

0x0x7ffdeb77d158->main arg3[54]=(0x7ffdeb77ed48) SHELL=/bin/bash

0x0x7ffdeb77d160->main arg3[55]=(0x7ffdeb77ed58) QT_ACCESSIBILITY=1

0x0x7ffdeb77d168->main arg3[56]=(0x7ffdeb77ed6b) MAKE_TERMOUT=/dev/pts/1

0x0x7ffdeb77d170->main arg3[57]=(0x7ffdeb77ed83) GDMSESSION=ubuntu

0x0x7ffdeb77d178->main arg3[58]=(0x7ffdeb77ed95) LESSCLOSE=/usr/bin/lesspipe %s %s

0x0x7ffdeb77d180->main arg3[59]=(0x7ffdeb77edb7) LC_MEASUREMENT=gl_ES.UTF-8

0x0x7ffdeb77d188->main arg3[60]=(0x7ffdeb77edd2) OPAM_SWITCH_PREFIX=/home/ismael/.opam/ocaml-latest

0x0x7ffdeb77d190->main arg3[61]=(0x7ffdeb77ee05) LC_IDENTIFICATION=gl_ES.UTF-8

0x0x7ffdeb77d198->main arg3[62]=(0x7ffdeb77ee23) QT_IM_MODULE=ibus

0x0x7ffdeb77d1a0->main arg3[63]=(0x7ffdeb77ee35) PWD=/home/ismael/Escritorio/SO/PRÁCTICAS/P3

0x0x7ffdeb77d1a8->main arg3[64]=(0x7ffdeb77ee63) XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/usr/share/:/var/lib/snapd/desktop

0x0x7ffdeb77d1b0->main arg3[65]=(0x7ffdeb77eec9) DBUS_STARTER_ADDRESS=unix:path=/run/user/1000/bus,guid=8c45d9b2e512cee3459f36c46759d61b

0x0x7ffdeb77d1b8->main arg3[66]=(0x7ffdeb77ef21) XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg

0x0x7ffdeb77d1c0->main arg3[67]=(0x7ffdeb77ef4e) LC_NUMERIC=gl_ES.UTF-8

0x0x7ffdeb77d1c8->main arg3[68]=(0x7ffdeb77ef65) MFLAGS=

0x0x7ffdeb77d1d0->main arg3[69]=(0x7ffdeb77ef6d) LC_PAPER=gl_ES.UTF-8

0x0x7ffdeb77d1d8->main arg3[70]=(0x7ffdeb77ef82) MEMORY_PRESSURE_WRITE=c29tZSAyMDAwMDAgMjAwMDAwMAA=

0x0x7ffdeb77d1e0->main arg3[71]=(0x7ffdeb77efb5) MANPATH=:/home/ismael/.opam/ocaml-latest/man

0x0x7ffdeb77d1e8->main arg3[72]=(0x7ffdeb77efe2) VTE_VERSION=7600

Muestra el entorno del proceso con envp[], tercer argumento del main, al igual que lo haría showvar sin argumentos.

2-environ -environ

0x0x7ffdeb77cfa8->environ[0]=(0x7ffdeb77dfeb) LESSOPEN=| /usr/bin/lesspipe %s

0x0x7ffdeb77cfb0->environ[1]=(0x7ffdeb77e00b) MAIL=/var/mail/ismael

0x0x7ffdeb77cfb8->environ[2]=(0x7ffdeb77e021) LANGUAGE=es:gl:en_CA:en

0x0x7ffdeb77cfc0->environ[3]=(0x7ffdeb77e039) LC_TIME=gl_ES.UTF-8

0x0x7ffdeb77cfc8->environ[4]=(0x7ffdeb77e04d) USER=ismael

0x0x7ffdeb77cfd0->environ[5]=(0x7ffdeb77e059) no_proxy=localhost,127.0.0.0/8,::1

0x0x7ffdeb77cfd8->environ[6]=(0x7ffdeb77e07c) XDG_SESSION_TYPE=wayland

0x0x7ffdeb77cfe0->environ[7]=(0x7ffdeb77e095) SHLVL=3

0x0x7ffdeb77cfe8->environ[8]=(0x7ffdeb77e09d) HOME=/home/ismael

0x0x7ffdeb77cff0->environ[9]=(0x7ffdeb77e0af) CAML_LD_LIBRARY_PATH=/home/ismael/.opam/ocaml-latest/lib/stublibs:/home/ismael/.opam/ocaml-latest/lib/ocaml/stublibs:/home/ismael/.opam/ocaml-latest/lib/ocaml

0x0x7ffdeb77cff8->environ[10]=(0x7ffdeb77e14e) NO_PROXY=localhost,127.0.0.0/8,::1

0x0x7ffdeb77d000->environ[11]=(0x7ffdeb77e171) DESKTOP_SESSION=ubuntu

0x0x7ffdeb77d008->environ[12]=(0x7ffdeb77e188) OCAML_TOPLEVEL_PATH=/home/ismael/.opam/ocaml-latest/lib/toplevel

0x0x7ffdeb77d010->environ[13]=(0x7ffdeb77e1c9) GNOME_SHELL_SESSION_MODE=ubuntu

0x0x7ffdeb77d018->environ[14]=(0x7ffdeb77e1e9) GTK_MODULES=gail:atk-bridge

0x0x7ffdeb77d020->environ[15]=(0x7ffdeb77e205) LC_MONETARY=gl_ES.UTF-8

0x0x7ffdeb77d028->environ[16]=(0x7ffdeb77e21d) MAKEFLAGS=

0x0x7ffdeb77d030->environ[17]=(0x7ffdeb77e228) DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus,guid=8c45d9b2e512cee3459f36c46759d61b

0x0x7ffdeb77d038->environ[18]=(0x7ffdeb77e284) SYSTEMD_EXEC_PID=2354

0x0x7ffdeb77d040->environ[19]=(0x7ffdeb77e29a) DBUS_STARTER_BUS_TYPE=session

0x0x7ffdeb77d048->environ[20]=(0x7ffdeb77e2b8) LIBVIRT_DEFAULT_URI=qemu:///system

0x0x7ffdeb77d050->environ[21]=(0x7ffdeb77e2db) COLORTERM=truecolor

0x0x7ffdeb77d058->environ[22]=(0x7ffdeb77e2ef) DEBUGINFOD_URLS=https://debuginfod.ubuntu.com

0x0x7ffdeb77d060->environ[23]=(0x7ffdeb77e31e) MAKE_TERMERR=/dev/pts/1

0x0x7ffdeb77d068->environ[24]=(0x7ffdeb77e336) WAYLAND_DISPLAY=wayland-0

0x0x7ffdeb77d070->environ[25]=(0x7ffdeb77e350) IM_CONFIG_PHASE=1

0x0x7ffdeb77d078->environ[26]=(0x7ffdeb77e362) LOGNAME=ismael

0x0x7ffdeb77d080->environ[27]=(0x7ffdeb77e371) _=/usr/bin/make

0x0x7ffdeb77d088->environ[28]=(0x7ffdeb77e381) MEMORY_PRESSURE_WATCH=/sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service/app.slice/app-gnome\x2dsession\x2dmanager.slice/gnome-session-manager@ubuntu.service/memory.pressure

0x0x7ffdeb77d090->environ[29]=(0x7ffdeb77e438) XDG_SESSION_CLASS=user

0x0x7ffdeb77d098->environ[30]=(0x7ffdeb77e44f) USERNAME=ismael

0x0x7ffdeb77d0a0->environ[31]=(0x7ffdeb77e45f) TERM=xterm-256color

0x0x7ffdeb77d0a8->environ[32]=(0x7ffdeb77e473) GNOME_DESKTOP_SESSION_ID=this-is-deprecated

0x0x7ffdeb77d0b0->environ[33]=(0x7ffdeb77e49f) PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin

0x0x7ffdeb77d0b8->environ[34]=(0x7ffdeb77e507) SESSION_MANAGER=local/asus:@/tmp/.ICE-unix/2354,unix/asus:/tmp/.ICE-unix/2354

0x0x7ffdeb77d0c0->environ[35]=(0x7ffdeb77e555) PAPERSIZE=a4

0x0x7ffdeb77d0c8->environ[36]=(0x7ffdeb77e562) LC_ADDRESS=gl_ES.UTF-8

0x0x7ffdeb77d0d0->environ[37]=(0x7ffdeb77e579) GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/62241c1e_6309_4402_912b_571f93346219

0x0x7ffdeb77d0d8->environ[38]=(0x7ffdeb77e5cf) GNOME_SETUP_DISPLAY=:1

0x0x7ffdeb77d0e0->environ[39]=(0x7ffdeb77e5e6) XDG_RUNTIME_DIR=/run/user/1000

0x0x7ffdeb77d0e8->environ[40]=(0x7ffdeb77e605) XDG_MENU_PREFIX=gnome-

0x0x7ffdeb77d0f0->environ[41]=(0x7ffdeb77e61c) MAKELEVEL=1

0x0x7ffdeb77d0f8->environ[42]=(0x7ffdeb77e628) DISPLAY=:0

0x0x7ffdeb77d100->environ[43]=(0x7ffdeb77e633) LANG=gl_ES.UTF-8

0x0x7ffdeb77d108->environ[44]=(0x7ffdeb77e644) LC_TELEPHONE=gl_ES.UTF-8

0x0x7ffdeb77d110->environ[45]=(0x7ffdeb77e65d) XDG_CURRENT_DESKTOP=ubuntu:GNOME

0x0x7ffdeb77d118->environ[46]=(0x7ffdeb77e67e) XAUTHORITY=/run/user/1000/.mutter-Xwaylandauth.B0DKY2

0x0x7ffdeb77d120->environ[47]=(0x7ffdeb77e6b4) GNOME_TERMINAL_SERVICE=:1.137

0x0x7ffdeb77d128->environ[48]=(0x7ffdeb77e6d2) XDG_SESSION_DESKTOP=ubuntu

0x0x7ffdeb77d130->environ[49]=(0x7ffdeb77e6ed) XMODIFIERS=@im=ibus

0x0x7ffdeb77d138->environ[50]=(0x7ffdeb77e701)
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.webp=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:

0x0x7ffdeb77d140->environ[51]=(0x7ffdeb77ecf0) SSH_AGENT_LAUNCHER=openssh

0x0x7ffdeb77d148->environ[52]=(0x7ffdeb77ed0b)
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh

0x0x7ffdeb77d150->environ[53]=(0x7ffdeb77ed34) LC_NAME=gl_ES.UTF-8

0x0x7ffdeb77d158->environ[54]=(0x7ffdeb77ed48) SHELL=/bin/bash

0x0x7ffdeb77d160->environ[55]=(0x7ffdeb77ed58) QT_ACCESSIBILITY=1

0x0x7ffdeb77d168->environ[56]=(0x7ffdeb77ed6b) MAKE_TERMOUT=/dev/pts/1

0x0x7ffdeb77d170->environ[57]=(0x7ffdeb77ed83) GDMSESSION=ubuntu

0x0x7ffdeb77d178->environ[58]=(0x7ffdeb77ed95) LESSCLOSE=/usr/bin/lesspipe %s %s

0x0x7ffdeb77d180->environ[59]=(0x7ffdeb77edb7) LC_MEASUREMENT=gl_ES.UTF-8

0x0x7ffdeb77d188->environ[60]=(0x7ffdeb77edd2)
OPAM_SWITCH_PREFIX=/home/ismael/.opam/ocaml-latest

0x0x7ffdeb77d190->environ[61]=(0x7ffdeb77ee05) LC_IDENTIFICATION=gl_ES.UTF-8

0x0x7ffdeb77d198->environ[62]=(0x7ffdeb77ee23) QT_IM_MODULE=ibus

0x0x7ffdeb77d1a0->environ[63]=(0x7ffdeb77ee35)
PWD=/home/ismael/Escritorio/SO/PRÁCTICAS/P3

0x0x7ffdeb77d1a8->environ[64]=(0x7ffdeb77ee63)
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/usr/share/:/var/lib/snapd/desktop

0x0x7ffdeb77d1b0->environ[65]=(0x7ffdeb77eec9)
DBUS_STARTER_ADDRESS=unix:path=/run/user/1000/bus,guid=8c45d9b2e512cee3459f36c46759d61b

0x0x7ffdeb77d1b8->environ[66]=(0x7ffdeb77ef21) XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg

0x0x7ffdeb77d1c0->environ[67]=(0x7ffdeb77ef4e) LC_NUMERIC=gl_ES.UTF-8

0x0x7ffdeb77d1c8->environ[68]=(0x7ffdeb77ef65) MFLAGS=

0x0x7ffdeb77d1d0->environ[69]=(0x7ffdeb77ef6d) LC_PAPER=gl_ES.UTF-8

0x0x7ffdeb77d1d8->environ[70]=(0x7ffdeb77ef82) MEMORY_PRESSURE_WRITE=c29tZSAyMDAwMDAgMjAwMDAwMAA=

0x0x7ffdeb77d1e0->environ[71]=(0x7ffdeb77efb5) MANPATH=:/home/ismael/.opam/ocaml-latest/man

0x0x7ffdeb77d1e8->environ[72]=(0x7ffdeb77efe2) VTE_VERSION=7600

Muestra el entorno del proceso al igual que antes, pero esta vez mediante la variable global externa environ.

3-environ -addr:

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> environ -addr

environ: 0x7ffdeb77cfa8 (almacenado en 0x5deaeb26e8b8)

main arg3: 0x7ffdeb77cfa8 (almacenado en 0x7ffdeb67c7d0)

(1) (2)

1- La primera es la dirección del puntero (envp y environ). Ambas direcciones son iguales.

2- La segunda es la dirección donde se almacena dicho puntero.

-environ es una dirección global no inicializada por lo que se guarda en el segmento de datos (en el bss).

-envp (tercer argumento del main) se almacena en la pila (stack). Esto se debe a que es una variable local de la función main. Recordamos que las variables locales se almacenan en la pila. Es un puntero que apunta a la memoria que contiene las variables de entorno del programa.

7-fork:

El proceso padre crea en cada llamada un proceso hijo:

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> fork
ejecutando proceso 70112

- `fork` crea un nuevo proceso.
- El proceso **hijo** tiene el PID `70112`.
- El mensaje "ejecutando proceso 70112" indica que el proceso hijo ha sido creado y está ejecutando su código. El PID `70112` es el del hijo.

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> fork
ejecutando proceso 70114

- Se llama nuevamente a `fork()`, lo que crea otro **proceso hijo**.
- El PID del nuevo proceso hijo es `70114`, y el mensaje indica que este nuevo proceso hijo se está ejecutando.
- El proceso **padre** es el proceso que lanzó este nuevo `fork()`, es decir, el proceso con PID `70112` (el primer proceso).

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> fork
ejecutando proceso 70115

- Se ejecuta otro `fork()`, creando otro **proceso hijo**.
- El PID de este proceso hijo es `70115`.
- Este proceso hijo también está ejecutando su código y ha sido creado por el proceso con PID `70114` (el proceso padre del hijo actual).

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> pid
Pid del shell: 70115

- El comando `pid` muestra el **PID** del proceso actual (el **shell** o proceso principal).
- En este caso, el shell tiene el PID `70115`, que es el último proceso creado, es decir, el **proceso hijo más reciente**.
- Esto indica que el proceso actual es el hijo con PID `70115`, y no el proceso inicial.

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> ppid
Pid del padre del shell: 70114

- El comando `ppid` muestra el **PID del padre** del proceso actual (en este caso, el shell).
- El **padre** del proceso con PID `70115` es el proceso con PID `70114`, que es el proceso que lo creó.

- GESTIÓN DE DIRECTORIOS DE BÚSQUEDA :

8-search: search [-add dir | -del dir | clear | path]
Search muestra la lista de búsqueda que servirá paa buscar archivos ejecutables
-add dir: añade un directorio a la lista de busqueda
-del dir: elimina un directoio de la lista de búsqueda
-clear: limpia la lista de búsqueda
-path: importa los directorios del path a la lista de búsqueda

1-Lista de búsqueda inicialmente vacía
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> search
No hay directorios en la lista de búsqueda

2-Añadimos los directorios del path a la lista de búsqueda
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> search -path
Directorio '/usr/local/sbin' añadido a la lista de búsqueda
Directorio '/usr/local/bin' añadido a la lista de búsqueda
Directorio '/usr/sbin' añadido a la lista de búsqueda
Directorio '/usr/bin' añadido a la lista de búsqueda
Directorio '/sbin' añadido a la lista de búsqueda
Directorio '/bin' añadido a la lista de búsqueda
Directorio '/usr/games' añadido a la lista de búsqueda
Directorio '/usr/local/games' añadido a la lista de búsqueda
Directorio '/snap/bin' añadido a la lista de búsqueda
Directorios del PATH importados a la lista de búsqueda

3-Observamos como tenemos los directorios del path en la lista
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> search
Directorios en la lista de búsqueda:
- /usr/local/sbin
- /usr/local/bin
- /usr/sbin
- /usr/bin
- /sbin
- /bin
- /usr/games
- /usr/local/games
- /snap/bin

4-Creamos un directorio y lo añadimos a la lista
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> makedir hola
Directorio hola creado correctamente

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> search -add hola
Directorio 'hola' añadido a la lista de búsqueda
Directorio hola añadido a la lista de búsqueda

5-Observamos como el directorio se ha añadido a la lista
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> search
Directorios en la lista de búsqueda:
- /usr/local/sbin
- /usr/local/bin

- /usr/sbin
- /usr/bin
- /sbin
- /bin
- /usr/games
- /usr/local/games
- /snap/bin
- hola

6-Eliminamos un directorio de la lista
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> search -del /bin
Directorio '/bin' eliminado de la lista de búsqueda

7-Observamos como este directorio ha sido eliminado
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> search
Directorios en la lista de búsqueda:
- /usr/local/sbin
- /usr/local/bin
- /usr/sbin
- /usr/bin
- /sbin
- /usr/games
- /usr/local/games
- /snap/bin
- hola

8-Limpiamos la lista

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> search -clear
Lista de búsqueda limpiada

9-Observamos como la lista está vacía
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> search
No hay directorios en la lista de búsqueda

- FUNCIONES DE GESTIÓN Y EJECUCIÓN DE PROCESOS:

9-exec: ejecuta sin crear un nuevo proceso, el programa descrito por `progspec`
Este comando usa `exec` para **reemplazar el proceso actual del shell** por un programa que le pasamos con sus argumentos o variables. El shell **terminará después de ejecutar el comando**, ya que el proceso actual será reemplazado por el programa.

*RECORDAMOS que para que funcione estos comandos tendremos que buscarlos en la lista de ejecutables (lista de búsqueda). La mayoría de estos comandos se encuentran en el path en la carpeta /usr/local/bin

-> exec ls -l
total 980
-rw-r--r-- 1 ismael ismael 12 Nov 19 12:43 1.txt
-rwxrwxr-x 1 ismael ismael 12 Nov 19 10:34 ar.txt
-rw-rw-r-- 1 ismael ismael 4822 Dec 2 18:49 ayudaP3-25.c
-rw-rw-r-- 1 ismael ismael 32578 Dec 2 18:02 Entrega_p2.zip
-rw-rw-r-- 1 ismael ismael 161184 Dec 15 18:07 Infome_p3_prueba.odt
-rw-rw-r-- 1 ismael ismael 93268 Dec 2 18:31 LabAssignment3.pdf
-rw-rw-r-- 1 ismael ismael 4244 Dec 14 16:58 lista_busqueda.c
-rw-rw-r-- 1 ismael ismael 1145 Dec 14 12:41 lista_busqueda.h
-rw-rw-r-- 1 ismael ismael 10464 Dec 15 17:50 lista_busqueda.o
-rw-rw-r-- 1 ismael ismael 5560 Dec 10 11:04 lista.c
-rw-rw-r-- 1 ismael ismael 5296 Dec 10 11:03 lista_ficheros.c
-rw-rw-r-- 1 ismael ismael 1498 Dec 10 11:03 lista_ficheros.h
-rw-rw-r-- 1 ismael ismael 10864 Dec 15 17:50 lista_ficheros.o
-rw-rw-r-- 1 ismael ismael 1243 Dec 10 11:04 lista.h
-rw-rw-r-- 1 ismael ismael 17411 Dec 10 11:04 lista_memoria.c
-rw-rw-r-- 1 ismael ismael 2709 Dec 10 11:04 lista_memoria.h
-rw-rw-r-- 1 ismael ismael 28688 Dec 15 17:50 lista_memoria.o
-rw-rw-r-- 1 ismael ismael 10256 Dec 15 17:50 lista.o
-rw-rw-r-- 1 ismael ismael 4604 Dec 14 17:59 lista_procesos_bg.c
-rw-rw-r-- 1 ismael ismael 1771 Dec 14 17:58 lista_procesos_bg.h
-rw-rw-r-- 1 ismael ismael 11760 Dec 15 17:50 lista_procesos_bg.o
-rw-rw-r-- 1 ismael ismael 2206 Dec 14 12:43 Makefile
-rwxrwxr-x 1 ismael ismael 165440 Dec 15 18:23 p3
-rw-rw-r-- 1 ismael ismael 106345 Dec 15 18:23 p3.c
-rw-rw-r-- 1 ismael ismael 150200 Dec 15 18:23 p3.o
-rw-rw-r-- 1 ismael ismael 45034 Dec 14 18:15 P3.zip
-rw-rw-r-- 1 ismael ismael 17824 Dec 15 18:18 utils.c
-rw-rw-r-- 1 ismael ismael 2800 Dec 14 12:42 utils.h
-rw-rw-r-- 1 ismael ismael 37880 Dec 15 18:22 utils.o

ismael@asus:~/Escritorio/SO/PRÁCTICAS/P3 $ make -> SALIÓ DEL SHELL

-> exec sleep 10

(aquí no pone nada, pero sleep lo que hace es que el sistema se duerma en este caso 10 segundos y exec hace que el shell se cierre después de esos 10 segundos porque, sleep ha reemplazado el procesp del shell)
ismael@asus:~/Escritorio/SO/PRÁCTICAS/P3 $

CASO DE QUE NO EXISTA EL PROGRAMA
-> exec 23
Error: No such file or directory: '23'

-> exec
Error: No se especificó un programa para ejecutar

10-execpri: hace lo mismo que exec pero con una prioridad antes de pasarle un programa.

Las prioridades en el código van desde -20 (más alta, mayor prioridad) a 19 (mas baja, menor prioridad)

Ejemplo:
Si ejecutamos execpri 10 ls -l y después execpri -10 ls -l tenemos que ver que se ejecuta antes execpri -10 ls -l porque tiene MAYOR PRIORIDAD

11-fg (foreground): ejecuta un proceso en primer plano

Pero, **¿qué significa "primer plano"?**

Ejecutar en primer plano significa que el proceso que ejecutas **ocupa el terminal** hasta que termine. Es decir, no puedes usar la terminal para otros comandos mientras el proceso esté ejecutándose.

Cuando se ejecute un programa con fg nos mostrará su nuevo pid (pid del proceso hijo). Con ese pid y las señales podemos parar su ejecución.

0 - indica éxito

Otro número - indica error en el proceso

-> fg sleep 80

Ejecutando proceso 29969

(el programa esperará 80 segundos ocupando toda le terminal sin poder hacer nada)

Proceso terminado con código 0 (finaliza correctamente)

TERMINAL 1:

-> fg sleep 80

Ejecutando proceso 29969

(el programa esperará 80 segundos ocupando toda le terminal sin poder hacer nada, pero podemos parar su ejecución con señales desde otra terminal usando su pid)

TERMINAL 2:
kill -SIGTERM 29969

TERMINAL 1:
-> fg sleep 80 (el de antes)

Ejecutando proceso 29969

Proceso 29969 terminado por señal: TERM

12-fgpri:

Al igual que antes el programa que se le pase se ejecuta en primer plano, esta evz con una prioridad desde -20 a 19. La mayoría de las prioridades negativas no funcionan porque necesitan permisos de superusuario.

Vamos a probar este caso:
Vamos a ejecutar primero en una terminal el comando **fgpri 5 sleep 20** y en otra terminal después de haber ejecutado este comando ejecutaremos inmediatamente **fgpri 0 sleep 20**

TERMINAL1: (ejecutado primero, pero con menor prioridad)
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> fgpri 5 sleep 25
Ejecutando proceso 31487
Proceso terminado con código: 0

TERMINAL2: (ejecutado después pero con mayor prioridad)
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3) -> fgpri 0 sleep 25
Ejecutando proceso 31490
Proceso terminado con código: 0

ACABA PRIMERO LA TERMINAL 2, porque tiene mayor prioridad

para comprobar su ejecución y su prioridad podemos usar en una terminak normal:
-> ps -eo pid,ppid,ni,comm | grep sleep

13-back: ejecuta un proceso en segundo plano

14-backpri: ejecuta un proceso en segundo plano con prioridad

15-listjobs y 16-deljobs (-term | -sig):
listjobs: muestra la lista de procesos en segundo plano
deljobs:
-term: elimina los procesos terminados
-sig: elimina los procesos terminados por señal

Ejemplo completo con todo:

```
Bienvenido al Shell de Sistemas Operativos
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> search -path
Directorio '/home/ismael/.opam/ocaml-latest/bin' añadido a la lista de búsqueda
Directorio '/home/ismael/.local/bin' añadido a la lista de búsqueda
Directorio '/usr/local/sbin' añadido a la lista de búsqueda
Directorio '/usr/local/bin' añadido a la lista de búsqueda
Directorio '/usr/sbin' añadido a la lista de búsqueda
Directorio '/usr/bin' añadido a la lista de búsqueda
Directorio '/sbin' añadido a la lista de búsqueda
Directorio '/bin' añadido a la lista de búsqueda
Directorio '/usr/games' añadido a la lista de búsqueda
Directorio '/usr/local/games' añadido a la lista de búsqueda
Directorio '/snap/bin' añadido a la lista de búsqueda
El directorio '/snap/bin' ya está en la lista
Directorios del PATH importados a la lista de búsqueda

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> back
Uso: back programa [argumentos...]

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> back sleep 100
[38421] sleep

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> backpri sleep 100^[[D^[[D
[38429] 1 (priority: 0)


ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> backpri 5 sleep 100
[38434] sleep (priority: 5)

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> listjobs
PID    Usuario  Fecha            Estado         Comando        Prioridad
--------------------------------------------------------------------------------
38421  ismael   2024/12/16 18:34  ACTIVO         sleep          p=0
38434  ismael   2024/12/16 18:35  ACTIVO         sleep          p=5

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> listjobs
PID    Usuario  Fecha            Estado         Comando        Prioridad
--------------------------------------------------------------------------------
38421  ismael   2024/12/16 18:34  ACTIVO         sleep          p=0
38429  ismael   2024/12/16 18:35  TERMINADO (001) 1       p=0
38434  ismael   2024/12/16 18:35  SIGNALED (15)   sleep          p=5


ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> deljobs -term
Procesos eliminados: 1
```

```
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> listjobs
PID    Usuario  Fecha            Estado          Comando         Prioridad
---------------------------------------------------------------------------
38421  ismael   2024/12/16 18:34  ACTIVO          sleep           p=0
38434  ismael   2024/12/16 18:35  SIGNALED (15)   sleep           p=5


ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> listjobs -sig
PID    Usuario  Fecha            Estado          Comando         Prioridad
---------------------------------------------------------------------------
38421  ismael   2024/12/16 18:34  ACTIVO          sleep           p=0
38434  ismael   2024/12/16 18:35  SIGNALED (15)   sleep           p=5
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> deljobs -sig
Procesos eliminados: 1

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> listjobs
PID    Usuario  Fecha            Estado          Comando         Prioridad
---------------------------------------------------------------------------
38421  ismael   2024/12/16 18:34  TERMINADO (000) sleep           p=0
ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> listjobs -term
PID    Usuario  Fecha            Estado          Comando         Prioridad
---------------------------------------------------------------------------
38421  ismael   2024/12/16 18:34  TERMINADO (000) sleep           p=0

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> deljobs -term
Procesos eliminados: 1

ismael@asus(/home/ismael/Escritorio/SO/PRÁCTICAS/P3 ) -> listjobs
No hay procesos en segundo plano
```

INFORME SHELL

Nos tenemos que basar en el shell de referencia para hacer la práctica. Para ello tenemos una carpeta shell de referencia y dentro de ella una que se llama LINUX con varios archivos .out dentro. Accedemos a esta carpeta y abrimos la terminal. Para acceder al shell de referencia ejecutamos:
./shell.out
./shell32bit.out
Si no compila debemos darle permisos ejecutando:
chmod +x shell.out
chmod +x shell32bit.out
para comprobar los permisos:
ls -l shell.out
ls -l shell32bit.out

gcc -g -o p0.out p0.c

Para la entrega haremos un ***Makefile que cintenga valgrind y que compile solo con make. Para limpiar los ejecutables haremos make clean.

FICHEROS:
-p0.c: programa principal que contiene el main donde se ejecuta el bucle del shell

-lista.h y lista.c: definición e implementación de las funciones del TAD lista enlazada que se utiliza para hacer la función historic, que es una lista que almacena los comandos que vamos añadiendo al shell

-lista_ficheros.h y lista_ficheros.c: definición e implementación de las funciones del TAD lista enlazada que se utiliza para hacer las funciones open, close y dup relacionadas con ficheros

-utils.h y utils.c: funciones auxuliares que utilizaremos en el resto del ćodigo

-lista_memoria.h y lista_memoria.c: definición e implementación de las funciones del TAD lista enlazada que se utiliza para hacer las funciones de memoria de la p2.

-lista_busqueda.h y lista_busqueda.c: definición e implementación de las funciones del TAD lista enlazada que se utiliza para añadir los procesos en segundo plano de la p3.

-lista_procesos_bg.h y lista_procesos_bg.c: definición e implementación de las funciones del TAD lista enlazada que se utiliza para hacer las funciones de procesos en segundo plano de la p3.

Makefile:

```
# Variables
CC = gcc                # Compilador a usar
CFLAGS = -g -Wall -m64      # Opciones de compilación (g para depuración, Wall para mostrar todas las advertencias) y -m64 le indica al compilador que genere código para una arquitectura de 64 bits
OBJ = p3.o lista.o lista_ficheros.o utils.o lista_memoria.o lista_busqueda.o lista_procesos_bg.o # Archivos objeto a crear
TARGET = p3             # Nombre del ejecutable

# Regla por defecto que compila y ejecuta
all: $(TARGET)          # Compila el ejecutable
        ./$(TARGET)         # Ejecuta el programa

# Regla para crear el ejecutable
$(TARGET): $(OBJ)
        $(CC) $(OBJ) -o $@      # Enlaza los archivos objeto para crear el ejecutable

# Regla para compilar p1.c
p1.o: p3.c lista.h lista_ficheros.h utils.h lista_memoria.h
        $(CC) $(CFLAGS) -c $<   # Compila p1.c a p1.o

# Regla para compilar lista.c
lista.o: lista.c lista.h
        $(CC) $(CFLAGS) -c $<   # Compila lista.c a lista.o

# Regla para compilar lista_ficheros.c
lista_ficheros.o: lista_ficheros.c lista_ficheros.h
        $(CC) $(CFLAGS) -c $<   # Compila lista_ficheros.c a lista_ficheros.o

# Regla para compilar utils.c
utils.o: utils.c utils.h
        $(CC) $(CFLAGS) -c $<   # Compila utils.c a utils.o

# Regla para compilar lista_memoria.c
lista_memoria.o: lista_memoria.c lista_memoria.h
        $(CC) $(CFLAGS) -c $<   # Compila lista_memoria.c a lista_memoria.o

# Regla para compilar lista_busqueda.c
lista_busqueda.o: lista_busqueda.c lista_busqueda.h
        $(CC) $(CFLAGS) -c $<   # Compila lista_busqueda.c a lista_busqueda.o

lista_procesos_bg.o: lista_procesos_bg.c lista_procesos_bg.h
        $(CC) $(CFLAGS) -c $<   # Compila lista_procesos_bk.c a lista_procesos_bk.o

# Regla para ejecutar Valgrind en el programa
valgrind: $(TARGET)
        valgrind --leak-check=full --show-leak-kinds=all ./$(TARGET)

# Regla para ejecutar el programa con sudo (con privilegios de superusuario)
sudo: $(TARGET)
        sudo ./$(TARGET)        # Ejecuta el programa con privilegios de superusuario

# Regla para limpiar los archivos generados
clean:
        rm -f $(OBJ) $(TARGET)  # Elimina los archivos objeto y el ejecutable

.PHONY: all clean valgrind  # Indica que "all", "clean" y "valgrind" no son archivos
```

El archivo Makefile configura las reglas para compilar y enlazar el programa paso a paso, además de facilitar la limpieza y depuración del programa. También contiene valgrind, que permite detectar errores de memoria.

Variables:

- `CC`: Define el compilador a usar, en este caso `gcc`.
- `CFLAGS`: Define las opciones de compilación. `-g` es para generar información de depuración, y `-Wall` habilita todas las advertencias del compilador, lo que ayuda a detectar posibles problemas.
- `OBJ`: Lista de archivos objeto (`.o`) que se generarán a partir de los archivos fuente (`.c`).
- `TARGET`: Nombre del ejecutable final.

- **Reglas**:
- `all`: Es la regla predeterminada. Compila el ejecutable (usando las reglas establecidas) y luego lo ejecuta.
- `$(TARGET)`: Crea el ejecutable enlazando todos los archivos objeto especificados en `OBJ`. Usa `$@` para representar el objetivo (en este caso, el nombre del ejecutable).
- Reglas para compilar archivos `.c` específicos (`p2.c`, `lista.c`, etc.):
    - Cada archivo `.c` tiene su propia regla para compilarlo en un archivo objeto `.o`.
    - En cada línea, el comando `$(CC) $(CFLAGS) -c $<` compila el archivo `.c` correspondiente.
        - `-c`: Es una opción que le indica a `gcc` que solo compile el archivo y genere un archivo objeto (`.o`) en lugar de un ejecutable. Así, el archivo se enlazará posteriormente junto con otros archivos objeto para formar el ejecutable completo.
    - `$<` se refiere al primer prerequisito de la regla (en este caso, el archivo fuente `.c`).
- **Otras reglas útiles**:
- `valgrind`: Ejecuta el programa bajo *Valgrind* para detectar fugas de memoria y otros errores de memoria.
- `clean`: Elimina los archivos generados (`.o` y el ejecutable), limpiando el directorio de trabajo.
- **Declaración de `.PHONY`**:
- Especifica que `all`, `clean`, y `valgrind` no son archivos sino nombres de reglas. Esto evita conflictos si existieran archivos con estos nombres.

EXPLICACIÓN CÓDIGO Y SUS FUNCIONES
p0.c, p1.c, p2.c y p3.c

1.1- INICIO DEL CÓDIGO:
Declaración de las librerías que vamos a utilizar:

#include <stdio.h> // Librería estándar de C para operaciones de E/S
#include "utils.h" //Añadimos el archivo con las funciones auxiliares para usar en los comandos
#include <string.h> // Librería para manipulación de cadenas
#include <stdlib.h> // Librería estándar de C para funciones generales como gestión de memoria
#include <unistd.h> // Librería necesaria para funciones de obtención de PID y PPID
#include <time.h>   // Librería con funciones y tipos para manipular tiempos y fechas
#include <sys/utsname.h> // Librería que proporciona la interfaz para acceder a la función uname(), que permite obtener información del SO
#include "lista.h" // Añadimos la lista enlazada para hacer la función del histórico, en la que almacenaremos el historial de comandos
#include <ctype.h> //Librería con funciones para clasificar y convertir caracteres
#include "lista_ficheros.h" //Añadimos la lista enlazada que contiene funciones para poder trabajar con ficheros para poder hacer las funciones open, close y dup
#include <fcntl.h> // Para usar O_CREAT, O_RDONLY, O_WRONLY etc. Es una cabecera de la biblioteca estándar de C que proporciona funciones relacionadas con la manipulación de archivos y descriptores de archivos
/ usamos también esta librería para llamar a funciones como open,close, read*/*
#include <sys/stat.h> // Librería que define la estructura de datos para manejar los atributos de archivos, como el tamaño, los permisos y el tipo de archivo. Se usa comúnmente con la función mkdir para crear directorios y definir sus permisos
#include <errno.h> //Librería para manejar errores
#include <limits.h> //Librería para obtener constantes como PATH_MAX
#include <dirent.h> //Librería para manipular directorios
#include <libgen.h> //Librería para manipular rutas de archivos
#include "lista_memoria.h" //Añadimos la lista enlazada que contiene funciones para poder trabajar con la memoria

Definimos también:
-MAX_INPUT: Valor máximo de caracteres por comando
-MAX_ARGS: Valor máximo de argumentos permitidos a ingresar en los comandos del shell.out

Declaramos el histórico como list (list.h y .c) y los ficheros (lista_ficheros.c y .h)
Los veremos luego en las funciones y la lista de memoria como MemoryList (lista_memoria.c y .h)

List historico; // Declaramos la lista que hemos creado para almacenar el histórico de comandos
ListFicheros ficheros; //Declaramos la lista para manejar ficheros abiertos
*MemoryList *memoria; //Declaramos la lista para manejar la memoria*

struct cmd: Se define una estructura `cmd` que almacena el nombre de un comando y punteros a funciones que manejan uno o varios argumentos.
funcion 1 (arg): hacer referencia a los comandos que tienen más argumentos del tipo -n, -t etc
función 2(tr[]): para comandos que requieren múltiples argumentos, funciones que usarán ficheros principalmente

1.2- DEFINICIÓN DE LAS FUNCIONES QUE VAMOS A UTILIZAR
1-FUNCIONES DE LOS COMANDOS
-> FUNCIONES P0
-> FUNCIONES P1
-> FUNCIONES P2
-> FUNCIONES P3

1.3-MAIN:
En el main declaramos un buffer e inicializamos las listas (del historial y la de los ficheros) y hacemos el siguiente bucle while:

```
// BUCLE del shell
while (1) {
imprimirPrompt();
leerEntrada(input);
procesarEntrada(input);
}
```

Por último eliminamos las listas creadas una vez finaliza el bucle. Liberamos memoria

1.4-IMPLEMENTACIÓN DE LAS FUNCIONES

-imprimirPrompt:

Imprime "->" cada vez que se pide un comando.

Utlizamos esta línea *fflush(stdout);* por lo siguiente:

`fflush(stdout);` es una función en C que **fuerza el vaciado del búfer de salida** asociado con `stdout` (la salida estándar, que normalmente es la consola o terminal).

¿Por qué se utiliza?

En sistemas operativos, la salida (como texto que se imprime en la consola) se maneja mediante un **búfer**. En lugar de enviar cada carácter directamente al dispositivo de salida (pantalla, archivo, etc.), los datos se almacenan temporalmente en el búfer para optimizar el rendimiento, enviando un bloque más grande de datos en lugar de hacerlo carácter por carácter.

- **Salida en búfer**: El contenido de `stdout` (lo que ves en pantalla) no siempre se muestra inmediatamente. Puede quedarse temporalmente en un búfer interno y solo se mostrará cuando:
    - Se llena el búfer.
    - Se encuentra un carácter especial, como un salto de línea (`\n`).
    - Se ejecuta explícitamente `fflush()`.

-leerEntrada:

Leemos la entrada (comando introducido por el usuario)

Usamos fgets para leer la entrada con un máximo de caracteres y strcspn para eliminar el salto de línea.

La función `fgets` se utiliza en C para leer una línea de texto desde un flujo de entrada, como `stdin` (entrada estándar). Su estructura básica es la siguiente:

La función `fgets` se utiliza en C para leer una línea de texto desde un flujo de entrada, como `stdin` (entrada estándar). Su estructura básica es la siguiente:

```
char fgets (char *str, int n, FILE stream)
```

Parámetros:

- **`char *str`**: Un puntero a un buffer donde se almacenará la línea leída. Este buffer debe ser lo suficientemente grande para contener la cadena y el carácter nulo (`\0`) que indica el final de la cadena. -> **BUFFER**
- **`int n`**: El número máximo de caracteres a leer, incluido el carácter nulo. Esto significa que `fgets` leerá hasta `n-1` caracteres. -> **MAX_INPUT**
- **`FILE *stream`**: Un puntero al flujo de entrada desde el cual se leerá la línea. Comúnmente se usa `stdin`, pero puede ser cualquier archivo abierto. **->STDIN**

-procesarEntrada():
Permite la interacción del usuario con el sistema, procesando y ejecutando comandos de manera estructurada.
La función procesarEntrada gestiona la entrada de comandos del usuario en un shell personalizado. Primero, declara un arreglo de punteros para almacenar los argumentos del comando y obtiene el número de argumentos mediante la función TrocearCadena, que separa la entrada en partes basadas en espacios y saltos de línea. Si no se ingresan argumentos, la función termina sin hacer nada.

A continuación, se crea un objeto para almacenar el comando en el historial. Se determina la posición del nuevo comando en el historial, copiando el primer argumento (el nombre del comando) y los argumentos restantes en el objeto correspondiente. Este objeto se inserta en la lista de historial.

Luego, la función busca el comando ingresado en un arreglo que contiene los comandos disponibles. Si encuentra una coincidencia, verifica si hay una función asociada para un solo argumento o para múltiples argumentos, y llama a la función correspondiente con los argumentos adecuados.

Si el comando no se encuentra en la lista, la función imprime un mensaje de error, indicando que el comando no se pudo ejecutar.

-trocearCadena:
Utiliza strtok para dividir los comandos igresados y obtener así el número de trozos encontrados en la cadena

FUNCIONES DE LOS COMANDOS

-P0:

1-authors

2-pid:

Simplemente si no se pasa ningun argumento, devolvemos el pid con la función **getpid()** de la librería `<unistd.h>`

3-ppid:

Lo mismo que antes pero llamamos a **getppid()**

4-cd:

-cwd[PATH_MAX]- almacena el directorio actual
PATH_MAX es lo maximo que puede contener y pertenece a `<limits.h>`.
-path: guarda el nuevo directorio al que queremos cambiar

Si no se pasa argumento (solo cd) tenemos que ir al directorio Home. Para ello usamos la función **getenv()** de la librería `<stdlib.h>`.

Una vez que estamos en home, podemos acceder a los nuevos directorios que se especifican como argumentos si estos realmente existen en nuestro directorio actual.

Determinar el nuevo directorio:

- Si no se da un argumento, usa el directorio "home" del usuario (`getenv("HOME")`).
- Si se da un argumento, usa esa ruta.
- **Cambiar de directorio**:
- Intenta cambiar con `chdir(path)`.
- Si falla, muestra un mensaje de error con `perror`.
- Si tiene éxito, muestra el nuevo directorio actual con `getcwd`.

CHDIR cambia de diretorio: 0 si fue exitoso

-1 si hay error

5-date:

La función `cmd_date` muestra la fecha y/o la hora actual según el argumento que reciba. Si no recibe ningún argumento, imprime ambas. Si recibe `-t`, muestra solo la hora; con `-d`, muestra solo la fecha.

1. **Obtener el tiempo actual**:
    - `time_t t = time(NULL);` obtiene el tiempo actual en segundos desde el 1 de enero de 1970 (conocido como el Epoch).
    - `localtime(&t);` convierte ese tiempo en una estructura `tm` ajustada a la zona horaria local, que almacena la información de la fecha y hora de manera desglosada (año, mes, día, horas, minutos, segundos).
2. **Formatear la salida**:
    - `strftime` convierte la información de `tm` en una cadena con el formato deseado: `%H:%M:%S` para la hora o `%d/%m/%Y` para la fecha, que luego se imprime.

Origen de las funciones y tipos predefinidos

- **`time_t` y `time`**: Estos vienen de la biblioteca `<time.h>` de C. `time_t` es un tipo de dato para representar el tiempo en segundos, y `time` obtiene el tiempo actual en ese formato.
- **`localtime`**: También de `<time.h>`, convierte el tiempo en segundos en una estructura `tm` desglosada en fecha y hora locales.
- **`strftime`**: Permite formatear la fecha y hora en una cadena con el formato específico, y también proviene de `<time.h>`.

Este conjunto de funciones facilita manejar y dar formato al tiempo en programas en C.

6-historic

Empieza en 0

La función `cmd_historic` maneja el historial de comandos utilizando varias funciones de apoyo. Dependiendo del argumento recibido, realiza las siguientes tareas:

1. **Historial vacío**:
    - Llama a `isEmptyList(historico)` para verificar si el historial (`historico`) está vacío. Si es así, imprime un mensaje y termina.
2. **Mostrar todo el historial**:
    - Si no se pasa ningún argumento, llama a `first(historico)` para obtener el inicio del historial y luego a `printList` para imprimir todos los comandos en orden.
3. **Borrar el historial**:
    - Si el argumento es `-c`, llama a `deleteList(&historico)` para vaciar el historial y confirma la eliminación con un mensaje.

4. **Ejecutar un comando específico**:
    - Si el argumento es un número, usa `atoi(arg)` (convierte la cadena a número) y `findItem(num, historico)` para encontrar el comando en esa posición en el historial.
    - Si lo encuentra, lo obtiene con `getItem(pos, historico)`, reconstruye el comando con `strcat`, y luego llama a `procesarEntrada` para ejecutar el comando.
5. **Mostrar los últimos N comandos**:
    - Si el argumento es `-N` (como `-5`), convierte el número con `atoi(arg + 1)` y recorre la lista desde el final para mostrar los últimos N comandos usando `first` y `last` para obtener posiciones en la lista.

Funciones de Biblioteca Estándar de C:

- `strcmp` (de `<string.h>`) compara cadenas.
- `atoi` (de `<stdlib.h>`) convierte una cadena en número entero.
- `printf` imprime mensajes, y `strcat` concatena cadenas.

7-open:

La función `cmd_open` abre un archivo en el shell y lo añade a una lista de archivos abiertos. Según los argumentos:

1. **Sin nombre de archivo**: Muestra la lista de archivos abiertos y termina.
2. **Establece el modo de apertura**: Lee los argumentos para definir cómo abrir el archivo:
    - `"cr"` (`O_CREAT = 0100`): crea el archivo si no existe.
    - `"ex"` (`O_EXCL = 0200`): falla si el archivo ya existe.
    - `"ro"` (`O_RDONLY = 00`): solo lectura.
    - `"wo"` (`O_WRONLY = 01`): solo escritura.
    - `"rw"` (`O_RDWR = 02`): lectura y escritura.
    - `"ap"` (`O_APPEND = 02000`): añade al final del archivo.
    - `"tr"` (`O_TRUNC = 01000`): borra el contenido existente.
3. **Abrir el archivo**: Usa `open` para abrir (y crear, si corresponde) el archivo en el modo especificado.
4. **Agregar a la lista de archivos abiertos**: Si se abre correctamente, lo añade a la lista de archivos abiertos. Si ocurre un error, lo cierra con `close`

int open(const char *pathname, int flags, mode_t mode); -> `<fcntl.h>`)

mode -> son los permisos

Con cr (create) se crea un archivo, si este no está creado.

¿Qué son los permisos? -> Flags

En Unix y Linux, cada archivo y carpeta tiene **permisos** que controlan quién puede hacer qué con ellos. Los permisos son importantes para la seguridad del sistema.

Cómo se representan los permisos

Los permisos se representan con números en formato octal (base 8). Cada permiso se representa con un número específico:

6: Permiso de lectura y escritura (`rw`) -> 0110

- **4**: Permiso de lectura (`r`) -> 0100
- **2**: Permiso de escritura (`w`) -> 0010
- **1**: Permiso de ejecución (`x`) -> 0001

Cómo se combinan los permisos

Puedes combinar estos números para dar varios permisos a los usuarios:

- **Lectura (4)** + **Escritura (2)** = **6** (`rw-`)
- **Lectura (4)** + **Ejecución (1)** = **5** (`r-x`)
- **Escritura (2)** + **Ejecución (1)** = **3** (`-wx`)
- **Lectura (4)** + **Escritura (2)** + **Ejecución (1)** = **7** (`rwx`)

Estructura de permisos

Los permisos se organizan en tres grupos:

1. **Propietario** (el usuario que creó el archivo)
2. **Grupo** (los usuarios que pertenecen a un grupo específico)
3. **Otros** (todos los demás usuarios)

Por ejemplo, los permisos `0666` significan:

- **0**: Sin permisos especiales (el primer número).
- **6**: El propietario tiene permisos de lectura y escritura (`rw-`).
- **6**: El grupo tiene permisos de lectura y escritura (`rw-`).
- **6**: Otros usuarios tienen permisos de lectura y escritura (`rw-`).

Así que `0666` significa que todos pueden **leer** y **escribir** el archivo, pero **no** pueden ejecutarlo.

Resumen

- Los números (4, 2, 1) representan permisos (lectura, escritura, ejecución).
- Se combinan para formar permisos más complejos.
- Se agrupan en propietario, grupo y otros usuarios.
- Cada número tiene un significado que indica qué puede hacer cada grupo con el archivo.

¡Espero que esto aclare cómo funcionan los permisos de archivos en Unix/Linux!

8-close:

La función `cmd_close` se encarga de cerrar un descriptor de archivo que ha sido abierto previamente y de eliminar ese descriptor de una lista de archivos abiertos. Aquí tienes un desglose de sus pasos:

1. **Verifica argumentos**: Comprueba si se ha proporcionado un argumento (el número del descriptor de archivo). Si no se pasa ningún argumento, llama a `listarDescriptores()` para mostrar la lista de descriptores abiertos y termina la función.
2. **Convierte a número**: Convierte el argumento proporcionado (que debería ser un número de descriptor de archivo) de texto a un entero usando `atoi()`.
3. **Validación**: Verifica si el número del descriptor de archivo es válido (mayor o igual a 0). Si es menor que 0, imprime un mensaje de error.
4. **Cerrar el descriptor**: Utiliza la función `close()` para intentar cerrar el descriptor de archivo:
    - Si `close()` devuelve `-1`, significa que hubo un error al cerrar el descriptor, y se imprime un mensaje de error usando `perror()`.
    - Si se cierra correctamente, intenta eliminar el descriptor de la lista de archivos abiertos usando `eliminarDeFicherosAbiertos()`.
        - Si se elimina con éxito, imprime un mensaje de confirmación.
        - Si no se encuentra el descriptor en la lista, informa que no estaba en la lista.

Funciones y conceptos utilizados

- **`close(int fd)`**: Esta función del sistema cierra el descriptor de archivo que se le pasa como argumento. Si tiene éxito, libera los recursos asociados con el descriptor.
- **`atoi(const char *str)`**: Convierte una cadena de texto (string) en un número entero. Si la cadena no es válida, devuelve 0.
- **`listarDescriptores()`**: Función que muestra todos los descriptores de archivos actualmente abiertos. No se define en el código proporcionado, pero su propósito es ayudar al usuario.
- **`eliminarDeFicherosAbiertos(int fd, &ficheros)`**: Esta función elimina el descriptor de archivo de una lista de archivos abiertos (no se muestra en el código, pero se infiere que debe interactuar con una estructura que almacena la lista de archivos).
- **`perror(const char *s)`**: Imprime un mensaje de error en la salida estándar, que incluye la descripción del error correspondiente a `errno`.

9-dup

¿Qué hace `cmd_dup`?

La función `cmd_dup` se encarga de duplicar un descriptor de archivo existente y agregar el nuevo descriptor a una lista de archivos abiertos. A continuación, se detallan los pasos que sigue:

1. **Verifica argumentos**: Comprueba si se ha proporcionado un argumento (el número del descriptor de archivo). Si no se pasa, llama a `listarDescriptores()` para mostrar la lista de descriptores abiertos y termina la función.
2. **Valida el descriptor**: Convierte el argumento a un entero usando `atoi()`. Si el valor es negativo, llama a `listarFicherosAbiertos(ficheros)` para mostrar los archivos abiertos y termina la función.
3. **Duplica el descriptor**: Utiliza la función `dup()` para duplicar el descriptor de archivo. Si `dup()` devuelve `-1`, imprime un mensaje de error usando `perror()` y termina la función.
4. **Obtiene el nombre del archivo**: Llama a `nombreFicheroDescriptor()` para obtener el nombre del archivo asociado al descriptor original. Si no se encuentra, asigna un nombre genérico basado en el número del descriptor (por ejemplo, "entrada estándar", "salida estándar", "error estándar").
5. **Crea un nombre para el nuevo descriptor**: Usa `snprintf()` para crear una cadena que contiene el nuevo nombre del descriptor duplicado, incluyendo el número del descriptor original y su nombre.
6. **Obtiene el modo del descriptor**: Utiliza `fcntl()` con la opción `F_GETFL` para obtener el modo del descriptor original (como lectura, escritura, etc.). Si falla, imprime un mensaje de error y cierra el descriptor duplicado.
7. **Añade el nuevo descriptor a la lista**: Llama a `anadirAFicherosAbiertos()` para agregar el nuevo descriptor duplicado a la lista de archivos abiertos. Si tiene éxito, imprime un mensaje de confirmación; de lo contrario, imprime un error y cierra el descriptor duplicado.

Funciones y conceptos utilizados

- **`dup(int oldfd)`**: Duplicar un descriptor de archivo. Devuelve un nuevo descriptor que es una copia del descriptor proporcionado (`oldfd`). Si hay un error, devuelve `-1`.
- **`atoi(const char *str)`**: Convierte una cadena de texto (string) en un número entero.
- **`listarDescriptores()`**: Función que lista todos los descriptores de archivos actualmente abiertos. (No se define en el código, pero se infiere que ayuda al usuario.)
- **`nombreFicheroDescriptor(int fd, ficheros)`**: Esta función busca y devuelve el nombre del archivo asociado a un descriptor de archivo en una lista de archivos abiertos.

- **`snprintf(char *str, size_t size, const char *format, ...)`**: Formatea e imprime datos en una cadena (string) de manera segura, evitando desbordamientos.
- **`fcntl(int fd, int cmd, ...)`**: Se usa para obtener o establecer las propiedades de un descriptor de archivo. En este caso, se usa `F_GETFL` para obtener el modo del descriptor. Viene de <fcntl.h>. -> FILE CONTROL
- **`anadirAFicherosAbiertos(int fd, int modo, const char* nombre, &ficheros)`**: Añade el nuevo descriptor a una lista de archivos abiertos. La función no se define en el código proporcionado, pero se infiere que maneja la lista de archivos abiertos.
- **`perror(const char *s)`**: Imprime un mensaje de error que incluye el texto proporcionado y la descripción del error correspondiente a `errno`.

10-infosys:

Obtiene información del sistema operativo -> **struct utsanme** `<sys/utsname.h>`.

Estructura `utsname`: (uts = Unix Time Sharing)

- La función utiliza la estructura `utsname` que se define en la biblioteca `<sys/utsname.h>`. Esta estructura contiene varios campos que describen el sistema operativo:
  - `nodename`: El nombre de la red de la máquina.
  - `sysname`: El nombre del sistema operativo.
  - `release`: La versión del kernel.
  - `version`: Información adicional sobre la versión del kernel.
  - `machine`: La arquitectura del hardware (por ejemplo, x86_64).
- **Verificación de Argumentos**:
- La función comprueba si el argumento `arg` es `NULL` o una cadena vacía. Si es así, procede a obtener la información del sistema.
- Si se pasa algún argumento (es decir, `arg` no es `NULL` y no es una cadena vacía), imprime un mensaje de error indicando que el argumento es inválido y que debe usarse simplemente `infosys`.
- **Llamada a `uname`**:
- Se utiliza la función `uname` para llenar la estructura `sys_info` con información del sistema. Se usa para llamar al struct utsname.
- Si `uname` falla (devuelve -1), se imprime un mensaje de error utilizando `perror`, que muestra el error asociado con la última llamada del sistema.
- **Salida Formateada**:
- Si la llamada a `uname` tiene éxito, la función imprime la información del sistema de manera formateada:

- Muestra el nombre del host, la arquitectura, el nombre del sistema operativo, la versión del kernel y detalles adicionales de la versión.

11-help:

Muestra una lista con lo que hace cada comando.

Si se pasa un argumento que es un comando, devuelve solo lo que hace ese comando

12-salir:

Si no se pasa ningun argumento, salimos del shell

En el momento en el que salimos borramos todas las listas que antes que habíamos inicializado e incluso podríamos llegar a usar.

-P1:

13-cwd: CURRENT WORK DIRECTORY

La función `cmd_cwd` se encarga de imprimir el directorio de trabajo actual del sistema. A continuación se detalla su funcionamiento:

1. **Declaración de un buffer**: Se define un array de caracteres llamado `buffer` con un tamaño de 1024 bytes. Este buffer se utilizará para almacenar la ruta del directorio actual.
2. **Verificación del argumento**: La función primero comprueba si el argumento `arg` es `NULL` o una cadena vacía. Esto se hace para determinar si debe proceder a imprimir el directorio actual.
3. **Obtención del directorio actual**: Si el argumento es nulo o vacío, se llama a la función `getcwd`, que se encuentra en la biblioteca `<unistd.h>`. Esta función intenta llenar el `buffer` con la ruta del directorio actual y devuelve un puntero a este buffer. Si `getcwd` tiene éxito, se imprime el contenido del `buffer` en la consola.
4. **Manejo de errores**: Si `getcwd` falla (es decir, devuelve `NULL`), se utiliza `perror` para imprimir un mensaje de error indicando que hubo un problema al obtener el directorio actual.
5. **Argumento no válido**: Si se pasa un argumento que no es `NULL` ni vacío, se imprime un mensaje que indica que el argumento es inválido y se sugiere usar el comando `cwd`.

Resumen de funciones y bibliotecas utilizadas:

- `getcwd`: Obtiene el directorio de trabajo actual. Se encuentra en la biblioteca `<unistd.h>`.
- `perror`: Imprime un mensaje de error basado en el valor de `errno`, y está definida en `<stdio.h>`.

14-makefile:

La función `cmd_makefile` se encarga de crear un archivo con el nombre proporcionado como argumento. Primero, verifica si se ha dado un nombre; si no es así, llama a la función `cmd_cwd` para mostrar el directorio actual y termina su ejecución.

Luego, intenta crear el archivo usando la función `open`, aplicando las siguientes banderas: `O_CREAT` para crear el archivo si no existe, `O_WRONLY` para abrirlo en modo de solo escritura, y `O_EXCL` para asegurarse de que no se sobrescriba un archivo existente. También especifica los permisos del archivo como `0644`, lo que permite que el propietario tenga permisos de lectura y escritura, mientras que otros usuarios solo pueden leer.

Si `open` devuelve `-1`, significa que hubo un error. Si el error es que el archivo ya existe (`EEXIST`), se muestra un mensaje indicando que no se puede crear el archivo. Para otros errores, se utiliza `perror` para imprimir un mensaje de error más general.

Si la creación del archivo es exitosa, se imprime un mensaje confirmando que el archivo fue creado correctamente y finalmente se cierra el descriptor de archivo usando `close` para liberar el recurso.

open:

int open (const *char file, int oflag);

```
int fd = open(arg, O_CREAT | O_WRONLY, 0644);
```

Devuelve un int porque este valor es el **descriptor de archivo** que se utiliza para referirse al archivo abierto. Si hay un error al abrir el archivo, la función devuelve `-1`.

-file: aquí le pasamos una cadena de caracteres, en este caso el argumento

-oflag:
Este es un **entero** que contiene las **banderas** (flags) que determinan cómo se abrirá el archivo. Las banderas son combinaciones de valores predefinidos que controlan el modo de apertura del archivo. Ejemplos de banderas que puedes usar son:

- `O_RDONLY`: Abre el archivo solo para lectura.
- `O_WRONLY`: Abre el archivo solo para escritura.
- `O_RDWR`: Abre el archivo para lectura y escritura.
- `O_CREAT`: Crea el archivo si no existe.
- `O_TRUNC`: Trunca el archivo a 0 si ya existe (borra su contenido).
- `O_APPEND`: Abre el archivo en modo de añadir, de modo que cualquier escritura se realice al final del archivo.

¿Y que es el número ese?:

Este es el tercer argumento, que especifica los **permisos** del archivo cuando se crea. Los permisos se expresan en formato octal (base 8).

- **`0644`** se descompone de la siguiente manera:
    - `0`: Indica que se trata de un número en formato octal.
    - `6`: Permisos para el **propietario** del archivo (4 para lectura + 2 para escritura = 6).
    - `4`: Permisos para el **grupo** (4 para lectura).
    - `4`: Permisos para **otros usuarios** (4 para lectura).

Esto significa que el propietario del archivo podrá leer y escribir en él, mientras que el grupo y otros usuarios solo tendrán permisos de lectura.

Para acabar usamos la función **close** de la misma librería para cerrar el archivo

COMO FUNCIONAN LOS PERMISOS?:

0 – sin acceso al archivo en absoluto

- 1 – solo permisos de ejecución
- 2 – solo permisos de escritura
- 3 – permisos de escritura y ejecución
- 4 – solo permisos de lectura
- 5 – permisos de lectura y ejecución
- 6 – permisos de lectura y escritura
- 7 – permisos de lectura, escritura y ejecución (permisos completos)

15-makedir:

La función `cmd_makedir` es responsable de crear un nuevo directorio en el sistema de archivos con el nombre que se le pasa como argumento. Aquí está una explicación detallada de su funcionamiento:

1. **Verificación del argumento**: Primero, la función comprueba si el argumento `arg` (el nombre del directorio) es `NULL` o si es una cadena vacía. Si es así, llama a la función `cmd_cwd(NULL)`, que probablemente muestra el directorio de trabajo actual, y luego sale de la función.
2. **Comprobación de existencia del directorio**: Utiliza la función `stat`, que se encuentra en la biblioteca `<sys/stat.h>`, para obtener información sobre el archivo o directorio cuyo nombre se pasó como argumento. Si `stat` devuelve 0 y `S_ISDIR(st.st_mode)` verifica que el resultado corresponde a un directorio, significa que el directorio ya existe. En este caso, imprime un mensaje indicando que no se puede crear el directorio porque ya existe y retorna.
3. **Creación del directorio**: Si el directorio no existe, intenta crearlo con la función `mkdir`, que también se encuentra en `<sys/stat.h>`. Se le pasan dos parámetros: el nombre del nuevo directorio y el modo de permisos, que se establece en `0777`, lo que permite la lectura, escritura y ejecución para todos los usuarios.
4. **Manejo de errores**: Después de intentar crear el directorio, se comprueba el resultado de la operación. Si `mkdir` devuelve `-1`, se utiliza `perror` para imprimir un mensaje de error. Si la creación del directorio fue exitosa, se imprime un mensaje confirmando que el directorio se ha creado correctamente.

Resumen de las funciones y bibliotecas utilizadas:

- `stat`: Obtiene información sobre un archivo o directorio. Está definida en `<sys/stat.h>`.
- `S_ISDIR`: Una macro que verifica si un archivo es un directorio.
- `mkdir`: Crea un nuevo directorio. También está definida en `<sys/stat.h>`.
- `perror`: Imprime un mensaje de error basado en el valor de `errno`, y está definida en `<stdio.h>`.

A partir de ahora usaremos la libería `<sys/stat.h>` todo el rato usando los struct stat y lstat que contienen información de directorios y archivos.

Diferencias entre `stat` y `lstat`

1. **Enlaces simbólicos**:
   - **`stat`**: Cuando se utiliza para un enlace simbólico, `stat` sigue el enlace y devuelve información sobre el archivo o directorio al que apunta el enlace. Es decir, obtiene los atributos del destino del enlace.
   - **`lstat`**: En cambio, `lstat` no sigue el enlace simbólico. Devuelve información sobre el propio enlace simbólico. Esto significa que puedes obtener información como el tamaño del enlace, sus permisos, etc., sin seguirlo al archivo al que apunta.

Campos Comunes de `struct stat`

`st_dev`:

- Tipo: `dev_t`
- Describe el número de dispositivo en el que reside el archivo.

`st_ino`:

- Tipo: `ino_t`
- El número de inodo del archivo, que es un identificador único en el sistema de archivos.

`st_mode`:

- Tipo: `mode_t`
- Contiene los permisos y tipo del archivo (por ejemplo, si es un archivo regular, directorio, enlace simbólico, etc.).

`st_nlink`:

- Tipo: `nlink_t`
- Número de enlaces duros que apuntan al inodo del archivo.

`st_uid`:

- Tipo: `uid_t`
- Identificador de usuario del propietario del archivo.

`st_gid`:

- Tipo: `gid_t`
- Identificador de grupo del archivo.

`st_size`:

- Tipo: `off_t`
- Tamaño del archivo en bytes.

`st_atime`:

- Tipo: `time_t`
- Tiempo de último acceso al archivo.

`st_mtime`:

- Tipo: `time_t`
- Tiempo de la última modificación del contenido del archivo.

`st_ctime`:

- Tipo: `time_t`
- Tiempo del último cambio de metadatos del archivo (como permisos o propietario).

`st_blksize`:

- Tipo: `blksize_t`
- Tamaño del bloque de entrada/salida óptimo para el sistema de archivos.

etc

AUXILIARES PARA LISTFILE y LISTDIR

*-LetraTF:

Se le pasa como argumento mode_t que viene del struct stat :

> `st_mode`:
>
> - Tipo: `mode_t`
> - Contiene los permisos y tipo del archivo (por ejemplo, si es un archivo regular, directorio, enlace simbólico, etc.).

Y a partir de un fichero o directorio intenta convertir el tipo de archivo a una letra para mostrar luego ese tipo de archivo en convertirModo.

El operador **&** es el operador de **bitwise AND** (y bit a bit). Se utiliza aquí para enmascarar (o "filtrar") el modo del archivo `m` para obtener solo la parte relevante que indica el tipo de archivo.

- `S_IFMT` es una constante que contiene una máscara que selecciona los bits más significativos del modo que indican el tipo de archivo. Al hacer `m & S_IFMT`, se obtienen solo esos bits, ignorando el resto que representan los permisos y otros atributos del archivo.

*-ConvierteModo:

La función `ConvierteModo` se encarga de convertir los modos de archivo en una representación textual que es fácil de entender para los usuarios. Inicializa una cadena de permisos con caracteres que representan los permisos de lectura, escritura y ejecución para el propietario, grupo y otros.

- Comienza llenando la cadena con `----------` y luego reemplaza los caracteres correspondientes con los permisos que tiene el archivo:
  - `'r'`, `'w'`, `'x'` para lectura, escritura y ejecución, respectivamente.
- También verifica y añade los bits especiales de permisos (setuid, setgid, sticky) en las posiciones adecuadas de la cadena.

Finalmente, la función incluye el tipo de archivo en la primera posición de la cadena, la cual será luego utilizada en la salida para mostrar el estado completo del archivo.

Esta función es la que se usa para imprimir los permisos de cada archivo luego en -long

POR TANTO ESTO PASA EN -LONG CON ESTAS DOS FUNCIONES:

```
---------- :
      -: la primera barra usa LetraTF. Indica el tipo de fichero
      ---: las tres siguientes usan ConvierteModo que llama a LetraF
            y determina los permisos del propietario para el tipo de
      archivo anterior
      ---: las tres siguientes usan ConvierteModo que llama a LetraF
            y determina los permisos del grupo para el tipo de archivo anterior
      ---: las tres siguientes usan ConvierteModo que llama a LetraF
            y determina los permisos de otros usuarios para el tipo de archivo
      anterior
```

16-listfile

Declaración de Variables:

- Se declaran variables para controlar las opciones que el usuario puede especificar:
  - `longFormat`: Un indicador para mostrar información detallada (1 si se usa la opción `-long`).
  - `accessTime`: Un indicador para mostrar el tiempo de último acceso al archivo (1 si se usa la opción `-acc`).
  - `linkInfo`: Un indicador para mostrar información de enlaces simbólicos (1 si se usa la opción `-link`).
- `i`: Un contador que se inicializa en 1 para iterar a través de los argumentos pasados.
- **Procesar Opciones**:
- Se utiliza un bucle `while` para procesar los argumentos que comienzan con un guion (`-`). Esto permite al usuario especificar diferentes opciones:
  - `-long`: Si se especifica, se establece `longFormat` a 1, lo que significa que se mostrará un listado detallado, incluyendo el inodo del archivo o directorio.
  - `-acc`: Si se especifica, se establece `accessTime` a 1, lo que significa que se mostrará el último tiempo de acceso.
  - `-link`: Si se especifica, se establece `linkInfo` a 1, lo que significa que se mostrará la ruta de un enlace simbólico.
- Si se encuentra una opción no reconocida, se muestra un mensaje de error y se termina la función.
- **Uso del Directorio Actual**:
- Si no se proporcionan archivos o directorios en los argumentos después de las opciones, se llama a la función `cmd_cwd` para mostrar el directorio de trabajo actual y se termina la función.
- **Procesar Archivos**:
- Un segundo bucle `while` itera sobre los argumentos restantes, procesando cada archivo o directorio:
  - Si se solicitó `longFormat`, se llama a la función `printFileInfo`, que se encarga de imprimir la información detallada sobre el archivo o directorio.
  - Si `longFormat` no está habilitado, se usa `lstat` para obtener información sobre el archivo o directorio actual. `lstat` es similar a `stat`, pero también funciona con enlaces simbólicos, devolviendo información sobre el enlace en lugar de lo que apunta.
  - Si `lstat` falla (es decir, devuelve -1), se muestra un mensaje de error y se pasa al siguiente archivo.
- **Mostrar Información**:
- Dependiendo de la opción `accessTime`, se selecciona el tiempo a mostrar (último acceso o última modificación) y se formatea utilizando `strftime` para mostrarlo en un formato legible.

- Se imprime el tamaño del archivo y el nombre del archivo o directorio.
- Si se especificó `linkInfo` y el archivo es un enlace simbólico (`S_ISLNK`), se usa `readlink` para obtener la ruta a la que apunta el enlace, y se imprime junto con la información del archivo.
- Finalmente, se imprime una nueva línea para separar la información de cada archivo o directorio.

STRUCT DIRENT:

La estructura `dirent` es parte de las interfaces de programación de aplicaciones (API) en sistemas operativos basados en Unix, y es utilizada para trabajar con directorios y sus entradas. A continuación, se detalla de dónde proviene y cómo se usa:

Origen y Definición

- **Biblioteca y Encabezado**: La estructura `dirent` está definida en el archivo de encabezado `<dirent.h>`. Esta cabecera forma parte de la biblioteca estándar de C para sistemas Unix y Linux. Para usar `dirent`, debes incluir este encabezado en tu código C con la siguiente línea:

  ```
  #include <dirent.h>
  ```

Estructura `dirent`

- **Definición**: La estructura `dirent` representa una entrada en un directorio. Generalmente, contiene al menos dos campos:
  - `d_ino`: Un número de inodo que identifica de manera única la entrada en el sistema de archivos.
  - `d_name`: Un arreglo de caracteres que contiene el nombre de la entrada del directorio (archivo o subdirectorio).

  Un ejemplo simplificado de la estructura podría verse así:

```
struct dirent {
    ino_t d_ino;           // Número de inodo
    off_t d_off;           // Offset de la próxima dirent
    unsigned short d_reclen; // Longitud de esta entrada
    unsigned char d_type;   // Tipo de archivo
    char d_name[256];       // Nombre del archivo
};
```

Uso de `dirent`

1. Abrir un Directorio: Para trabajar con directorios, primero debes abrirlo usando la función `opendir()`, que devuelve un puntero a un objeto de tipo `DIR`, que se utiliza para iterar sobre las entradas del directorio.

```
DIR *dir = opendir("/ruta/al/directorio"
```

2. **Leer Entradas**: Utilizas readdir() para leer cada entrada del directorio. Esta función devuelve un puntero a una estructura dirent para cada entrada.

```
struct dirent *entry;

while ((entry = readdir(dir)) != NULL) {

   printf("Nombre del archivo: %s\n", entry->d_name);

}
```

3.Cerrar el Directorio: Una vez que hayas terminado de leer el directorio, debes cerrarlo con `closedir()`.

```
closedir(dir);
```

17-listdir:

La función `cmd_listdir` se encarga de listar el contenido de directorios, permitiendo opciones adicionales que modifican la forma en que se presenta la información. Esta función procesa varios argumentos y, dependiendo de las opciones proporcionadas, muestra detalles sobre los archivos y subdirectorios que se encuentran en los directorios especificados. Si no se proporciona ningún directorio, la función lista el contenido del directorio actual.

Funcionalidad Detallada

1. **Procesamiento de Opciones**:
    - La función comienza procesando los argumentos pasados en `tr[]`. Busca opciones que comiencen con `-` (guion) y activa las características correspondientes:
        - `-long`: Muestra información detallada sobre los archivos, incluyendo permisos, tamaño, propietario, etc.
        - `-hid`: Permite mostrar archivos ocultos (archivos cuyos nombres comienzan con un punto `.`).
        - `-acc`: Indica que se debe mostrar la última fecha de acceso de los archivos.
        - `-link`: Muestra la información de enlaces simbólicos.
2. **Directorio Actual**:
    - Si no se especifican directorios en los argumentos, la función usa `getcwd()` para obtener y mostrar el directorio de trabajo actual.
3. **Acceso a Directorios y Archivos**:
    - Para cada directorio o archivo proporcionado, se resuelve su ruta real usando `realpath()` y se obtiene su información mediante `lstat()`. Esto permite verificar si se puede acceder al archivo y si es un enlace simbólico.
    - Si el archivo es un enlace simbólico, se maneja de forma especial, utilizando `readlink()` para obtener la ruta del destino.
4. **Lectura de Contenido del Directorio**:
    - Si se encuentra un directorio, se abre utilizando `opendir()`, y se itera sobre cada entrada del directorio con `readdir()`.
    - Durante la iteración, se verifica si los archivos son ocultos. Si la opción `-hid` no está activada, se omiten los archivos cuyos nombres comienzan con un punto (`.`).
5. **Impresión de Información**:
    - La información sobre cada archivo o directorio se imprime utilizando la función `printFileInfo()`, que incluye los permisos (usando `ConvierteModo` y `LetraTF`), el tamaño del archivo, y, si se solicita, la última fecha de acceso o la información de enlaces simbólicos.

Funciones Predefinidas Utilizadas

1. **`lstat()`**: Esta función se utiliza para obtener información sobre un archivo o directorio, incluyendo su modo (permisos y tipo), tamaño, propietario, y más. La diferencia con `stat()` es que `lstat()` no sigue enlaces simbólicos.

2. **`getcwd()`**: Permite obtener el directorio de trabajo actual. Se utiliza para mostrar el directorio en caso de que no se especifique ninguno.
3. **`opendir()`** y **`readdir()`**: Estas funciones se utilizan para abrir un directorio y leer sus entradas, respectivamente. `opendir()` retorna un puntero a un objeto que representa el directorio, y `readdir()` permite iterar sobre cada entrada dentro del directorio.
4. **`realpath()`**: Resuelve la ruta de un archivo o directorio, convirtiendo rutas relativas a absolutas y resolviendo enlaces simbólicos.
5. **`readlink()`**: Utilizada para leer el destino de un enlace simbólico, proporcionando la ruta del archivo o directorio al que apunta.

continue:

Es lo que se encarga de ignorar los archivos ocultos durante el proceso de listado

. - directorio actual

.. - directorio padre del directorio actual

18-reclist:

-RECURSIVA:

`reclist_recursive` se encarga de:

1. **Abrir un directorio** y leer su contenido.
2. **Listar los archivos y directorios dentro de ese directorio**, aplicando reglas específicas sobre si se deben mostrar archivos ocultos o no.
3. **Llamarse a sí misma** para cada subdirectorio encontrado, de manera que se pueda profundizar en la estructura de directorios sin límites, mostrando todos los niveles de contenido.

Proceso de la Función

1. **Apertura del Directorio**:
    - La función intenta abrir el directorio especificado por la ruta (`path`) utilizando `opendir`.
    - Si la apertura del directorio falla, imprime un mensaje de error y sale de la función.
2. **Lectura del Contenido del Directorio**:
    - Se utiliza un bucle que llama a `readdir` para leer cada entrada del directorio.
    - Para cada entrada, se construye la ruta completa usando `snprintf`.
3. **Obtención de Información del Archivo**:
    - Se usa `lstat` para obtener información sobre el archivo o directorio actual.
    - Dependiendo de las opciones activadas (`long_format`, `show_hidden`, etc.), se decide si se debe mostrar la entrada:
        - **Archivos ocultos**: Si la opción `show_hidden` no está activada, se omiten archivos cuyos nombres comienzan con un punto (`.`).
        - **Directorios especiales**: Se omiten los directorios especiales `.` (el directorio actual) y `..` (el directorio padre).
4. **Impresión de Información**:
    - Si la entrada debe mostrarse, se llama a `printFileInfo`, que imprime información sobre el archivo o directorio, utilizando el formato especificado.
5. **Procesamiento de Subdirectorios**:
    - Después de listar los archivos y directorios visibles, la función utiliza `rewinddir` para reiniciar la lectura del directorio.
    - Se itera nuevamente sobre las entradas del directorio:
        - Para cada entrada que no sea `.` o `..`, se verifica si es un directorio (`S_ISDIR`).
        - Si es un subdirectorio y no es oculto (o si se deben mostrar archivos ocultos), se llama a `reclist_recursive` de nuevo con la ruta del subdirectorio.
    - Esto permite que la función "baje" por la estructura de directorios y liste todos los archivos y subdirectorios en cada nivel.
6. **Cierre del Directorio**:

- Una vez que se han procesado todas las entradas, la función cierra el directorio utilizando `closedir`.

FUNCION PRINCIPAL:
La función normal `cmd_reclist` es la encargada de iniciar el proceso de listado de directorios de manera recursiva. A continuación, te explico detalladamente qué hace y cómo se relaciona con la función recursiva `reclist_recursive`.

Propósito de `cmd_reclist`

La función `cmd_reclist` tiene el objetivo de:

1. Procesar opciones de entrada que modifican el comportamiento del listado (por ejemplo, si se debe mostrar información detallada o archivos ocultos).
2. Determinar el directorio o archivos que se deben listar, usando el directorio actual si no se especifica ninguno.
3. Iniciar el proceso de listado para cada directorio o archivo proporcionado en los argumentos.

Proceso de la Función

1. **Inicialización de Opciones**:
    - Se definen variables para controlar las opciones: `long_format`, `show_hidden`, `show_access_time`, y `show_link`. Estas variables se inicializan en `0` (falso).
    - `start_index` se inicializa en `1`, para procesar los argumentos comenzando después del nombre del comando.
2. **Procesamiento de Opciones**:
    - Se itera sobre los argumentos de entrada (almacenados en `tr`) y se comprueba si cada argumento es una opción que comienza con un guion (`-`).
    - Dependiendo del argumento, se activan las correspondientes opciones (por ejemplo, `-long` activa `long_format`).
    - Si se encuentra una opción no reconocida, se imprime un mensaje de error y se finaliza la función.
3. **Determinación del Directorio de Inicio**:
    - Si no se proporcionan directorios (es decir, si `tr[i]` es `NULL` después de procesar las opciones), se llama a `cmd_cwd` para imprimir el directorio actual.
4. **Iteración Sobre Directorios/Archivos**:
    - Se itera sobre los argumentos restantes para procesar cada uno de ellos.
    - Para cada argumento, se utiliza `lstat` para obtener información sobre el archivo o directorio correspondiente.
    - Si `lstat` falla, se imprime un mensaje de error y se continúa con la siguiente entrada.
5. **Llamada a Funciones de Listado**:
    - Si el argumento es un directorio (verificado con `S_ISDIR`), se llama a `reclist_recursive` para listar su contenido de manera recursiva.

- Si el argumento es un archivo normal (verificado con `S_ISREG`), se llama a `printFileInfo` para mostrar su información.

19-revlist:

La función `revlist_recursive` realiza tres pasadas para procesar los elementos en el directorio actual de una manera específica, donde se listan primero los subdirectorios y luego los archivos, mostrando los subdirectorios en orden jerárquico antes de imprimir su contenido. Esta estructura en tres etapas logra el efecto de "subdirectorios antes", que da como resultado una lista inversa en comparación con un listado típico.

Aquí está el desglose de cada pasada:

1. **Primera pasada: almacenar subdirectorios**
    - La primera pasada usa `readdir` para iterar sobre los elementos del directorio y almacenar únicamente los subdirectorios en un arreglo (`subdirs`).
    - Esta separación permite procesar primero los subdirectorios y después los archivos. Los subdirectorios almacenados en esta pasada se guardan para su procesamiento recursivo antes de listar los archivos en el directorio actual.
    - Se ignoran los archivos regulares y otros tipos de entradas no deseadas (dependiendo de la visibilidad de los archivos ocultos controlada por `show_hidden`).
2. **Segunda pasada: procesar subdirectorios de forma recursiva**
    - En la segunda pasada, la función recorre el arreglo `subdirs` y llama recursivamente a `revlist_recursive` para cada subdirectorio almacenado.
    - Esto asegura que cada subdirectorio se procese completamente (incluyendo sus propios subdirectorios y archivos) antes de que la función vuelva al directorio actual.
    - Este paso implementa la organización jerárquica en la que cada subdirectorio y sus contenidos se listan antes de los archivos del directorio actual.
3. **Tercera pasada: listar archivos en el directorio actual**
    - Después de procesar todos los subdirectorios, la función vuelve al principio del directorio con `rewinddir` y realiza una tercera pasada, esta vez para listar solo los archivos regulares.
    - Esta organización permite listar el contenido de los subdirectorios primero, y luego los archivos en el directorio en curso.
    - También, si se usa la opción `show_hidden`, se listan los directorios especiales `.` y `..` en esta fase.

20-erase:

Esta función, llamada `cmd_erase`, se encarga de eliminar archivos o directorios vacíos, siguiendo estos pasos:

1. **Revisa si se proporciona un nombre de archivo o directorio**:
    - Si no se recibe un argumento (`arg` está vacío o `NULL`), se llama a la función `cmd_cwd` para mostrar el directorio actual y la función termina.
2. **Verifica si el archivo o directorio existe**:
    - Usa `stat` para obtener información sobre el archivo o directorio especificado en `arg`.
    - Si `stat` falla (el archivo no existe o no es accesible), muestra un mensaje de error y termina.
3. **Distingue entre archivo y directorio**:
    - Si `arg` es un **archivo regular** (`S_ISREG`):
        - Llama a `remove` para intentar eliminarlo directamente, sin importar si contiene datos.
        - Si la eliminación es exitosa, imprime un mensaje de confirmación; si falla, muestra un mensaje de error.
    - Si `arg` es un **directorio** (`S_ISDIR`):
        - Intenta abrir el directorio usando `opendir`.
        - Si `opendir` falla, muestra un mensaje de error y termina.
        - **Verifica si el directorio está vacío**:
            - Usa `readdir` para leer cada entrada en el directorio.
            - Ignora las entradas `.` y `..`, ya que son referencias al directorio actual y al padre, respectivamente.
            - Si encuentra otras entradas, significa que el directorio no está vacío; marca `isEmpty` como `0` y sale del bucle.
        - Si el directorio está vacío (`isEmpty == 1`):
            - Llama a `rmdir` para eliminarlo y muestra un mensaje de confirmación si es exitoso.
            - Si falla, muestra un mensaje de error.
        - Si el directorio **no está vacío**, imprime un mensaje indicando que no se puede borrar.
4. **Para otras condiciones**:
    - Si `arg` no es ni archivo ni directorio (por ejemplo, un enlace simbólico o un tipo especial de archivo), muestra un mensaje indicando que no es un archivo ni directorio válido.

`remove`:

- Es parte de la biblioteca estándar de C (específicamente, la cabecera `<stdio.h>`).
- **Función**: `int remove(const char *filename);`
- **Propósito**: Se usa para eliminar un archivo o un directorio vacío.
- **Detalles**: Aunque `remove` puede funcionar con directorios vacíos en algunas implementaciones, se usa principalmente para eliminar archivos.
- **Retorno**: Retorna `0` si tiene éxito, y un valor distinto de cero en caso de error (cuando el archivo no existe o no se puede eliminar, por ejemplo).

`rmdir`:

- Es parte de la biblioteca de C POSIX (especificada en `<unistd.h>`).
- **Función**: `int rmdir(const char *pathname);`
- **Propósito**: Se usa exclusivamente para eliminar directorios vacíos.
- **Detalles**: Esta función es específica para directorios y solo funcionará si el directorio está vacío.
- **Retorno**: Retorna `0` si el directorio se elimina correctamente y `-1` si hay un error (por ejemplo, si el directorio no está vacío o no tiene permisos de eliminación).

Ambas funciones son muy comunes en programas que requieren manipulación de archivos y directorios, como en sistemas de administración de archivos o utilidades de limpieza.

21-delrec:

Función `cmd_delrec`

La función `cmd_delrec` se utiliza para eliminar archivos y directorios de forma recursiva. Comienza verificando si el argumento proporcionado (que debería ser la ruta del archivo o directorio a eliminar) es válido. Si el argumento está vacío o no se puede acceder al archivo/directorio, muestra un mensaje de error y finaliza la ejecución.

A continuación, la función determina el tipo de archivo que se está intentando eliminar. Si es un archivo regular, simplemente lo elimina usando la función `remove`. Si la eliminación es exitosa, imprime un mensaje de confirmación; si no, muestra un mensaje de error.

Si el argumento es un directorio, la función intenta abrirlo. Si no puede abrir el directorio, informa del error y finaliza. Si puede abrir el directorio, verifica si está vacío. Ignora las entradas especiales que representan el directorio actual y el directorio padre. Si encuentra cualquier otra entrada, significa que el directorio no está vacío.

Si el directorio está vacío, informa que no se puede eliminar porque está vacío. Si no está vacío, llama a una función auxiliar llamada `delete_recursive`, que se encargará de eliminar todos los archivos y subdirectorios dentro de ese directorio.

Si el argumento no es un archivo regular ni un directorio válido, también muestra un mensaje de error.

Función `delete_recursive`

La función `delete_recursive` es la encargada de eliminar recursivamente todo el contenido de un directorio. Comienza abriendo el directorio especificado. Si no puede abrirlo, imprime un mensaje de error.

Luego, recorre todas las entradas del directorio. Para cada entrada, ignora las que representan el directorio actual y el directorio padre. Para las demás entradas, construye la ruta completa y verifica su tipo. Si es un subdirectorio, la función se llama a sí misma para eliminar su contenido. Si es un archivo regular, lo elimina usando la función `remove`.

Una vez que se han eliminado todos los archivos y subdirectorios dentro del directorio, la función intenta cerrar el directorio y luego eliminarlo con la función `rmdir`. Si la eliminación del directorio es exitosa, imprime un mensaje de confirmación; de lo contrario, muestra un mensaje de error.

RESUMEN FUNCIONES PREDEFINIDAS P1:

```
#include <stdio.h>      // printf, perror, remove
#include <string.h>     // strcmp, strdup
#include <stdlib.h>     // malloc, free, exit
#include <unistd.h>     // getpid, getcwd, rmdir, unlink
#include <time.h>       // time, ctime
#include <sys/utsname.h> // uname
#include <ctype.h>      // isdigit, isalpha
#include <fcntl.h>      // open, close, read
#include <sys/stat.h>   // stat, lstat, mkdir, S_ISREG, S_ISDIR
#include <errno.h>      // errno, strerror
#include <limits.h>     // PATH_MAX
#include <dirent.h>     // opendir, readdir, closedir, rewinddir
#include <libgen.h>     // basename, dirname
```

-P2:

Lista_memoria.h:

Vamos a crear 4 listas simplemente enlazadas para trabajar con las diferentes formas de asignar memoria. Crearemos una lista general y una de cada tipo (3 sublistas), la general contiene todas las sublistas por orden en el que se han añadido.

-Creamos un enum, que nos indicará de que tipo es la lista -> MALLOC, MMAP o SHARED

-Definimos nuestra estructura de datos con la siguiente información

```
typedef struct MemoryNode {
   void *addr;          // Dirección de memoria
   size_t size;         // Tamaño de la asignación
   time_t timestamp;    // Marca de tiempo de la asignación
   int fd;              // Descriptor de archivo (solo para mmap)
   int key;             // Clave de memoria compartida (solo para shared)
   MemType type;        // Tipo de asignación (malloc, mmap, shared)
   char filename[MAX_FILENAME]; // Nombre del archivo (solo para mmap)
   struct MemoryNode *next; // Siguiente nodo
} MemoryNode;
```

-Definimos nuestras partes de la lista:

```
typedef struct {
   MemoryNode *head;
   MemoryNode *tail;
} MemoryList;
```

-Y por último una estructura que determina los 4 tipos de listas:

```
typedef struct {
   MemoryList general;         //LISTA GENERAL
   MemoryList mallocList;    //LISTA PARA MALLOC
   MemoryList mmapList;     //LISTA PARA MMAP
   MemoryList sharedList;    //LISTA PARA SHARED
} MemoryManager;
```

FUNCIONES QUE USAMOS EN LA LISTA:

Funciones de inicialización y gestión de listas:

- `InitMemoryManager`: Inicializa el gestor de memoria.
- `FindMemoryBlock`: Busca un bloque de memoria por dirección.
- `InsertGeneralMemory`, `InsertMallocMemory`, `InsertMmapMemory`, `InsertSharedMemory`: Insertan bloques de memoria en las listas correspondientes. **En cada función de las sublistas después de haber añadido su contenido a su respectiva sublista llamamos a InsertGeneralMemory para insertar este contenido en la lista general.**
- `DeleteMemoryBlockFromGeneral`, `DeleteMallocMemory`, `DeleteMmapMemory`, `DeleteSharedMemory`: Elimina bloques de memoria de las listas correspondientes. **A su vez, una vez eliminado el contenido de cada sublista llamamos a la función DeleteMemoryBlockFromGeneral y según hemos eliminado el contenido de la sublista también lo haremos de la lista general.**

  ***DeleteMmapMemory: para eliminar un bloque de memoria mapeada usamos **munmap(addr,size)**

- `DeleteMmapMemorybyAddr`, `DeleteSharedMemorybyAddr:` eliminan una dirección que s les pasa si pertenece a alguno de estos dos tipos. Lo eliminan de su respectiva sublista y después de la lista general.

DeleteSharedMemory vs DeleteSharedMemoryByKey

- DeleteSharedMemory: desacopla la memoria y actualiza las listas -> **shmdt ()**

  shared memory detatch (desacoplar memoria)

- DeleteSharedMemoryByKey: elimina la clave del sistema pero no afecta a los procesos que aun están utilizando esa memoria -> **shmctl (IPC_RMID, Inter Process Communication_ Remove ID)**

  shared memory control (gestiona y controla segmentos de memoria compartida)

Funciones para limpiar recursos de memoria compartida:

- `LimpiarRecursosMemoriaCompartida`: Limpia recursos relacionados con la memoria compartida. Una vez finalizado el shell

- **Funciones de impresión**:
- Estas funciones permiten **imprimir las listas de memoria** (tanto generales como por tipo de asignación, como `malloc`, `mmap` o `shared`).

- ## Asignación de memoria

22-allocate: Asigna memoria

Esta función asigna bloques de memoria de diferentes tipos, como `malloc`, según los argumentos que se le pasen.

1. **Si no se pasan argumentos**, muestra todos los bloques de memoria.
2. Si el argumento es **`-malloc`**:
    - Si no se pasa un tamaño, muestra solo los bloques de memoria asignados con `malloc`. -> Lista específica solo con los allocate -malloc
    - Si se pasa un tamaño, intenta asignar ese espacio de memoria con `malloc` y lo agrega a una lista de bloques de memoria.

   Si el comando es incorrecto, muestra un mensaje.

%zu -> para variables de tipo size_t. Es una variable que se ajusta al tamaño adecuado según la plataforma (32 bits/64 bits)

3. Si el argumento es **-mmap:**

   Mapear un archivo en memoria, que es una forma de tratarlo como un bloque de memoria del programa. Con `mmap`, en lugar de leer el archivo de esta forma, puedes **asignar una dirección en la memoria del programa** que apunta al contenido del archivo. A partir de entonces, puedes tratar el contenido como si fuera una parte de la memoria de tu programa, lo cual hace que la lectura y modificación sean mucho más rápidas. -> do_Allocate_Mmap

4. Si el argumento es **-createshared:**

   Si no se pasan la clave y el tamaño (`tr[2]` y `tr[3]` son `NULL`), se muestra la lista de bloques de memoria compartida.

- Si se pasan la clave y el tamaño, se crea y asigna memoria compartida a través de `do_AllocateCreateshared`.

5. Si el argumento es **-shared**:

   Si no se pasan la clave y el tamaño (`tr[2]` y `tr[3]` son `NULL`), se muestra la lista de bloques de memoria compartida.

- Si se pasan la clave y el tamaño, se asigna memoria compartida existente mediante `do_AllocateShared`.

6. Si el argumento no existe: se muestra el mensaje de uso adecuado para el comando

AUXILIARES para allocate:

-ImprimirListaMemoria(lista_memoria): imprime la lista general de todos los bloques de memoria asignados

- Malloc:

-ImprimirListaMalloc (lista_memoria): imprime la lista solo de los mallocs hechos

-InsertMallocMemory(lista_memoria): para añadir el bloque de memoria a la lista de malloc y a su vez a la lista general.

- Mmap

-ImprimirListaMmap(lista_memoria): imprime la lista solo de los bloques de memoria mapeados

-MapearFichero (utils):
La función `MapearFichero` mapea un archivo en memoria, lo que significa que te permite acceder a los datos del archivo directamente desde la memoria del programa. No es necesario leer o escribir en el archivo usando funciones tradicionales como `read` o `write`. La función realiza los siguientes pasos:

1. **Abre el archivo**: Verifica que el archivo exista y lo abre en el modo adecuado (solo lectura o lectura y escritura).
2. **Mapea el archivo a memoria**: Usa `mmap` para crear una región de memoria que contiene los datos del archivo. Ahora puedes acceder al contenido del archivo como si fuera un arreglo en la memoria.

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

Guarda la información: Si el mapeo es exitoso, guarda la dirección de la memoria mapeada en una lista para gestionarlo más tarde.

Si algo falla en cualquier parte, la función devuelve `NULL`.

MAP_PRIVATE: importante esta asignación. Esto nos indicará que cualquier cambio realizado en la memoria mapeada **no se guardará en el archivo**.

-doAllocateMap (utils)

La función `do_AllocateMmap` gestiona la asignación de memoria mediante el mapeo de archivos en la memoria. Su propósito es permitir que un archivo sea mapeado a la memoria para que pueda ser accedido como si fuera un bloque de memoria, en lugar de tener que leer o escribir en el archivo directamente.

Explicación paso a paso:

1. **Verificar si hay argumentos**:
    - Si `arg[0]` (el primer argumento) es `NULL`, la función simplemente imprime la lista de bloques de memoria mapeada previamente usando `ImprimirListaMmap`. Esto significa que si no se pasan argumentos, solo muestra los bloques mapeados.

2. **Analizar los permisos**:
    - Si hay un segundo argumento (`arg[1]`), que contiene los permisos de acceso para el archivo (como "rw" para lectura y escritura, por ejemplo), la función analiza esa cadena para establecer los permisos de protección de memoria.
    - Se verifica si los caracteres 'r', 'w' o 'x' están presentes en `arg[1]` para determinar los permisos de lectura, escritura y ejecución respectivamente. Estos permisos se combinan en la variable `protection`.
3. **Mapear el archivo**:
    - Luego, se llama a la función `MapearFichero` para mapear el archivo (especificado en `arg[0]`) en la memoria. La función también obtiene un descriptor de archivo `fd`.
    - Si el mapeo falla (es decir, `p` es `NULL`), se imprime un mensaje de error.
4. **Procesar el mapeo exitoso**:
    - Si el mapeo es exitoso, se imprime un mensaje indicando que el archivo ha sido mapeado en una dirección de memoria.
    - Después, se añade este bloque de memoria mapeada a la lista de memoria mapeada (`mmapList`) y se guarda la información sobre el archivo.
    - También se restablece el desplazamiento del archivo (offset) a 0, asegurándose de que se pueda leer desde el inicio.
5. **Registrar el archivo mapeado**:
    - La función registra el archivo en la lista de ficheros abiertos, agregando el descriptor de archivo con los permisos de lectura y escritura. Además, se agrega una descripción del archivo mapeado a la lista.

Resumen:

Esta función mapea un archivo a la memoria, asignando permisos de acceso según lo indicado por el usuario. Si tiene éxito, el archivo mapeado se agrega a la lista de bloques de memoria mapeada y se registra en la lista de ficheros abiertos para su uso posterior. Si hay un error, se muestra un mensaje de error.

- Shared (USO DE <sys/shm.h> y <sys/ipc.h>)

<sys/shm.h> - se usa para trabajar con memoria compartida. `shmget()`: Crea una zona de memoria compartida.

- `shmat()`: Conecta un proceso a la zona de memoria compartida.
- `shmdt()`: Desconecta un proceso de la memoria compartida.
- `shmctl()`: Controla la memoria compartida (por ejemplo, eliminarla)

<sys/ipc.h> - se utiliza para proporcionar funciones de **comunicación entre procesos (IPC)** y para **gestionar claves de IPC (Inter Process Communication)**

`IPC_RMID`: Es una constante que se usa para eliminar recursos IPC como memoria compartida o colas de mensajes.

*-ObtenerMemoriaShmget: -> struct shmid_ds

La función `ObtenerMemoriaShmget` gestiona la creación o el acceso a un segmento de memoria compartida mediante la función `shmget`. Toma dos parámetros: una clave (`clave`) que identifica el segmento de memoria y un tamaño (`tam`) que indica la cantidad de memoria que se desea asignar. Si el tamaño es distinto de 0, la función intenta crear un nuevo segmento de memoria utilizando la bandera `IPC_CREAT | IPC_EXCL`. Si la clave es `IPC_PRIVATE`, devuelve un error. Luego, intenta adjuntar el segmento a la memoria del proceso con `shmat`. Si se produce un error en cualquier paso, la memoria compartida se elimina (si se creó) y se devuelve `NULL`. Finalmente, si se obtiene correctamente la memoria, se guarda en una lista de memoria compartida (`InsertSharedMemory`), y se retorna el puntero al segmento de memoria asignado.

-do_AllocateCreateshared

La función `do_AllocateCreateshared` permite crear y asignar un segmento de memoria compartida con una clave y un tamaño especificados por los argumentos. Toma un array de cadenas (`tr`) que contiene la clave (`tr[0]`) y el tamaño (`tr[1]`). La clave se convierte en tipo `key_t` y el tamaño en `size_t`. Si el tamaño es 0, se muestra un error indicando que no se pueden asignar bloques de 0 bytes. Si los argumentos son válidos, se llama a la función `ObtenerMemoriaShmget` para intentar crear y obtener el segmento de memoria compartida. Si la asignación tiene éxito, se muestra la dirección de memoria asignada. Si falla, se imprime un mensaje de error con la clave y el motivo del fallo.

-do_AllocateShared

La función `do_AllocateShared` permite acceder y asignar un segmento de memoria compartida existente utilizando una clave dada. Toma un array de cadenas (`tr`) con la clave del segmento de memoria. Si no se pasa ninguna clave o si es inválida, se muestra la lista de bloques de memoria compartida. La clave se convierte en `key_t` y luego se llama a la función `ObtenerMemoriaShmget` con esta clave y un tamaño de 0 (lo que indica que solo se desea acceder a la memoria existente). Si la memoria se obtiene correctamente, se imprime la dirección de la memoria compartida. Si hay un error, se muestra un mensaje con la clave y el motivo del fallo.

- ## Deasignación de memoria

23-deallocate: deasigna bloques de memoria

Comportamiento según los argumentos

1. **Sin argumentos (`tr[1] == NULL`)**
    - Muestra una lista general de todos los bloques de memoria asignados.
    - Si no hay bloques, informa al usuario.
2. **Argumento `-malloc`**
    - Si no se proporciona un tamaño (`tr[2] == NULL`), muestra una lista de todos los bloques asignados con `malloc`.
    - Si se proporciona un tamaño (`tr[2]`):
        - Convierte el tamaño en un número (`size_t`).
        - Si el tamaño es válido (positivo), elimina el bloque de memoria correspondiente tanto de la lista específica de `malloc` como de la lista general -> DeleteMallocMemory
3. **Argumento `-mmap`**
    - Si no se proporciona un argumento adicional (`tr[2] == NULL`), muestra todos los bloques asignados con `mmap`.
    - Si se proporciona un argumento adicional (`tr[2]`):
        - Lo interpreta como un nombre de archivo y elimina el bloque de memoria asociado al archivo. -> DeleteMmapMemory
4. **Argumento `-shared`**
    - Si no se proporciona un argumento adicional (`tr[2] == NULL`), muestra todos los bloques asignados como memoria compartida.
    - Si se proporciona una clave de memoria (`tr[2]`), la convierte en un valor de tipo `key_t` y elimina el bloque asociado. -> DeleteSharedMemory
5. **Argumento `-delkey`**
    - Si no se proporciona un argumento adicional, muestra cómo usar el comando correctamente.
    - Si se proporciona una clave de memoria compartida (`tr[2]`):
        - Busca el identificador de memoria compartida usando la clave.
        - Si encuentra el identificador, lo elimina del sistema usando `shmctl` con el flag `IPC_RMID`. -> **shmget y shmctcl**

Otros casos

- Si el primer argumento (`tr[1]`) no coincide con ninguna de las opciones anteriores, se asume que es una **dirección de memoria**.
- Convierte la cadena en una dirección (`void *`) con strtoul y busca si existe en la lista general de bloques asignados.
- Si se encuentra, elimina el bloque de memoria correspondiente según su tipo:
    - **MALLOC**: Elimina un bloque asignado con `malloc`.

    - **MMAP**: Elimina un bloque asignado con `mmap`. -> DeleteMmapMemorybyAddr
    - **SHARED**: Elimina un bloque asignado como memoria compartida. -> DeleteMemSharedByAddr
- Si no se encuentra la dirección o el tipo no es manejado, muestra un mensaje de error.

Resumiendo la diferencia:

- **`-shared`**: Desacopla la memoria compartida de un proceso y la elimina de la lista de bloques, **pero la memoria sigue existiendo** en el sistema y otros procesos pueden seguir usándola.
- **`-delkey`**: Elimina la clave de memoria compartida del sistema (**sin afectar a los procesos que ya tienen la memoria adjunta**). Otros procesos que ya la están utilizando seguirán teniendo acceso hasta que la desacoplen por sí mismos.

AUXILIARES:

-DeleteMallocMemory:

Esta función elimina un bloque de memoria previamente asignado mediante `malloc` (de tipo `MALLOC`) de las listas de memoria.

Desglose:

- **Parámetros**:
  - `MemoryManager *manager`: El gestor de memoria que contiene las listas de bloques de memoria.
  - `size`: El tamaño del bloque de memoria que se desea eliminar.
- **Funcionamiento**:
  - Recorre la lista de bloques asignados por `malloc` (almacenados en `mallocList`).
  - Busca un bloque de memoria cuyo tamaño coincida con el tamaño especificado (`size`).
  - Si encuentra el bloque, elimina el nodo correspondiente de la lista `mallocList`.
  - También elimina el bloque de memoria de la lista general utilizando la función `DeleteMemoryBlockFromGeneral`.
  - Luego, libera la memoria real con `free(addr)` y finalmente libera el nodo de la lista.

Uso: Se usa para liberar la memoria que ha sido asignada con `malloc`.

-DeleteMmapMemory:

Esta función elimina un bloque de memoria asignado mediante `mmap`, identificable por el nombre de archivo que se usó en la operación `mmap`.

Desglose:

- **Parámetros**:
  - `MemoryManager *manager`: El gestor de memoria que contiene las listas de bloques de memoria.
  - `filename`: El nombre del archivo usado para crear el mapeo.
- **Funcionamiento**:
  - Recorre la lista de bloques asignados por `mmap` (almacenados en `mmapList`).
  - Si encuentra un bloque cuya cadena `filename` coincide con el argumento, elimina ese bloque de la lista.
  - Después, llama a `munmap()` para liberar la memoria mapeada del sistema.
  - Finalmente, libera el nodo de la lista de memoria mapeada.

Uso: Se usa para liberar la memoria mapeada a un archivo con `mmap` basado en el nombre del archivo.

-DeleteSharedMemory:
Esta función elimina un bloque de memoria compartida identificado por una clave (`key`).

Desglose:

- **Parámetros**:
  - `MemoryManager *manager`: El gestor de memoria que contiene las listas de bloques de memoria.
  - `key`: La clave de memoria compartida.
- **Funcionamiento**:
  - Recorre la lista de bloques de memoria compartida (`sharedList`) y busca el bloque cuyo campo `key` coincida con el valor proporcionado.
  - Si lo encuentra, elimina el bloque de la lista `sharedList` y también de la lista general.
  - Desacopla la memoria compartida con `shmdt()`.
  - Luego, obtiene el identificador del segmento de memoria con `shmget()`, y si es válido, elimina el segmento de memoria compartida con `shmctl(shmid, IPC_RMID, NULL)`.
  - Finalmente, libera el nodo de la lista.

Uso: Se usa para liberar la memoria compartida identificada por una clave

-shmget: (shm = shared memory -> memoria compartida)
`shmget` es una llamada al sistema que se usa para obtener un identificador de un segmento de memoria compartida en el sistema.

Desglose:

- **Sintaxis**: int shmget(key_t key, size_t size, int shmflg);

Parámetros:

- `key`: La clave para identificar el segmento de memoria.
- `size`: El tamaño de la memoria solicitada (usualmente se pone `0` si solo se desea obtener el segmento existente).
- `shmflg`: Flags que determinan las opciones para la operación (como permisos).

- **Funcionamiento**:

- Si el segmento de memoria no existe, crea uno nuevo.
- Si ya existe, devuelve el identificador del segmento existente.
- Retorna un identificador (`shmid`) que se utiliza en otras llamadas como `shmdt` o `shmctl`

-shmctl:

`shmctl` es una llamada al sistema usada para realizar operaciones sobre un segmento de memoria compartida, como eliminar el segmento o cambiar permisos.

Desglose:

- **Sintaxis**: int shmctl(int shmid, int cmd, struct shmid_ds *buf);

Parámetros:

- `shmid`: El identificador del segmento de memoria.
- `cmd`: El comando que se desea ejecutar (por ejemplo, `IPC_RMID` para eliminar).
- `buf`: Un puntero a una estructura `shmid_ds` que contiene información sobre el segmento de memoria (usado con ciertos comandos, pero no necesario para `IPC_RMID`).

- **Funcionamiento**:

- Permite realizar diversas acciones sobre la memoria compartida.
- En el caso de `IPC_RMID`, se usa para eliminar el segmento de memoria compartida del sistema.

- DeleteMmapMemorybyAddr:
Esta función elimina un bloque de memoria mapeada por su dirección de memoria, es decir, no se usa el nombre del archivo, sino que se elimina directamente según la dirección.

Desglose:

- **Parámetros**:
    - `MemoryManager *manager`: El gestor de memoria que contiene las listas de bloques de memoria.
    - `addr`: La dirección de memoria del bloque de `mmap` que se desea eliminar.
- **Funcionamiento**:
    - Recorre la lista de bloques de `mmap` (`mmapList`) y busca el bloque cuyo campo `addr` coincida con la dirección proporcionada.
    - Si encuentra el bloque, lo elimina de la lista de `mmapList` y lo elimina también de la lista general.
    - Llama a `munmap()` para liberar la memoria mapeada.
    - Libera el nodo de la lista.

Uso: Se usa cuando quieres liberar un bloque de memoria mapeada (`mmap`) identificándola por su dirección en lugar de su nombre de archivo.

-DeleteMemSharedByAddr:
Esta función elimina un bloque de memoria compartida por su dirección de memoria, no por la clave.

Desglose:

- **Parámetros**:
  - `MemoryManager *manager`: El gestor de memoria que contiene las listas de bloques de memoria.
  - `addr`: La dirección de memoria del bloque compartido que se desea eliminar.
- **Funcionamiento**:
  - Recorre la lista de bloques de memoria compartida (`sharedList`) y busca el bloque cuyo campo `addr` coincida con la dirección proporcionada.
  - Si lo encuentra, lo elimina de la lista `sharedList` y también de la lista general.
  - Desacopla la memoria compartida con `shmdt()`.
  - Finalmente, libera el nodo de la lista.

Uso: Se usa cuando quieres liberar un bloque de memoria compartida identificándola por su dirección en lugar de su clave.

- ## Operaciones de memoria

24-memfill: memfill dirección bytes caracter

Llena x bytes de memoria en una dirección con un caracter que debe ser un valor de ASCII en decimal.

Ejemplo de como funciona (en -mmap no se reflejan los cambios en el archivo porque tiene permisos privados. Los cambios **no se verán reflejados directamente en el archivo** cuando se utiliza `mmap` con el modo `MAP_PRIVATE`.

Los pasos que sigue son los siguientes:

1. **Validación de Argumentos:** La función comienza asegurándose de que se han pasado suficientes argumentos. Si alguno de los argumentos requeridos (`addr`, `cont`, `ch`) no está presente, muestra un mensaje de uso y termina la ejecución.
2. **Conversión de la Dirección de Memoria:** El primer argumento (`addr`) que es la dirección de memoria de inicio se convierte de una cadena hexadecimal a un puntero de tipo `void*`. La función `strtoull` se utiliza para convertir el valor hexadecimal de la cadena en una dirección de memoria.
3. **Verificación de la Dirección de Memoria:** Una vez que se obtiene la dirección de memoria, la función verifica si esa dirección está dentro de un bloque de memoria gestionado por el programa. Esto se hace utilizando la función `FindMemoryBlock`, que busca si la dirección proporcionada pertenece a un bloque válido. Si la dirección no es válida o no está gestionada, la función termina y muestra un mensaje de error.
4. **Conversión del Tamaño (Cantidad de Bytes a Llenar):** El segundo argumento (`cont`) es la cantidad de bytes que se deben llenar. Este valor se convierte de una cadena a un número entero (`size_t`). Si la cantidad de bytes es cero, muestra un mensaje de error.
5. **Verificación de los Límites del Bloque de Memoria:** La función verifica que la cantidad de bytes que se desea llenar no sobrepase los límites del bloque de memoria al que pertenece la dirección proporcionada. Si se intenta acceder fuera de los límites, la función termina y muestra un mensaje de error.
6. **Conversión del Carácter:** El tercer argumento (`ch`) es el carácter con el que se llenará la memoria. Este valor se pasa en formato decimal, y se convierte a un valor de tipo `unsigned char` (un byte). Este valor será el que se usará para llenar la memoria.
7. **Llenado de la Memoria:** Finalmente, se utiliza una función auxiliar llamada `LlenarMemoria` para llenar la memoria con el valor especificado. Esta función toma como parámetros la dirección de inicio, la cantidad de bytes y el valor del carácter. Después de realizar la operación, se imprime un mensaje confirmando cuántos bytes fueron llenados, en qué dirección, y el valor en formato ASCII y hexadecimal del carácter utilizado.

La línea **`0x%02x`** en el `printf` es un formato para imprimir un valor en **hexadecimal**.

`%x` imprime un número en formato hexadecimal y `02` asegura que el número tenga al menos 2 dígitos, rellenando con ceros a la izquierda si es necesario.

AUXILIARES:

-FindBlockMemory:

La función `FindMemoryBlock` recorre la lista de bloques de memoria gestionados por el programa, y verifica si una dirección específica (`addr`) está contenida dentro de alguno de los bloques. Si la dirección se encuentra en un bloque, la función devuelve el nodo que representa ese bloque; de lo contrario, devuelve `NULL`.

Este tipo de función es útil para gestionar y verificar las direcciones de memoria dentro de un sistema que maneja bloques de memoria dinámicos, como los que se asignan con `malloc`, `mmap`, etc.

-LlenarMemoria:

La función auxiliar `LlenarMemoria` es la encargada de realizar el llenado real de la memoria. Toma tres parámetros:

- **`addr`**: la dirección de memoria donde comenzar a llenar.
- **`cont`**: la cantidad de bytes que se deben llenar.
- **`ch`**: el valor con el que se deben llenar esos bytes.

Dentro de esta función, simplemente se utiliza la función estándar `memset` para llenar la memoria en la dirección `addr` con el valor `ch` durante `cont` bytes. El uso de `memset` es ideal para este tipo de operaciones, ya que optimiza el llenado de bloques de memoria de manera eficiente.

25-memdup: memdump addr cont

Vuelca el contenido de cont bytes de memoria en la dirección addr a la pantalla. Vuelca códigos hexadecimales, y para los caracteres imprimibles, el carácter asociado.

La función `cmd_memdump` realiza las siguientes tareas:

1. Verifica que se hayan proporcionado los argumentos necesarios (dirección y cantidad de bytes).
2. Convierte la dirección de memoria y la cantidad de bytes a los tipos de datos apropiados.
3. Comprueba si la dirección está dentro de un bloque de memoria gestionado por el programa.
4. Verifica que el rango de memoria solicitado no exceda el tamaño del bloque.
5. Imprime el contenido de la memoria en formato hexadecimal y de caracteres:

1. Muestra la dirección al inicio de cada línea.
2. Imprime los códigos hexadecimales de cada byte.
3. Al final de cada línea (o al final del volcado), muestra los caracteres imprimibles correspondientes.
4. Los caracteres no imprimibles se representan con un punto ('.').

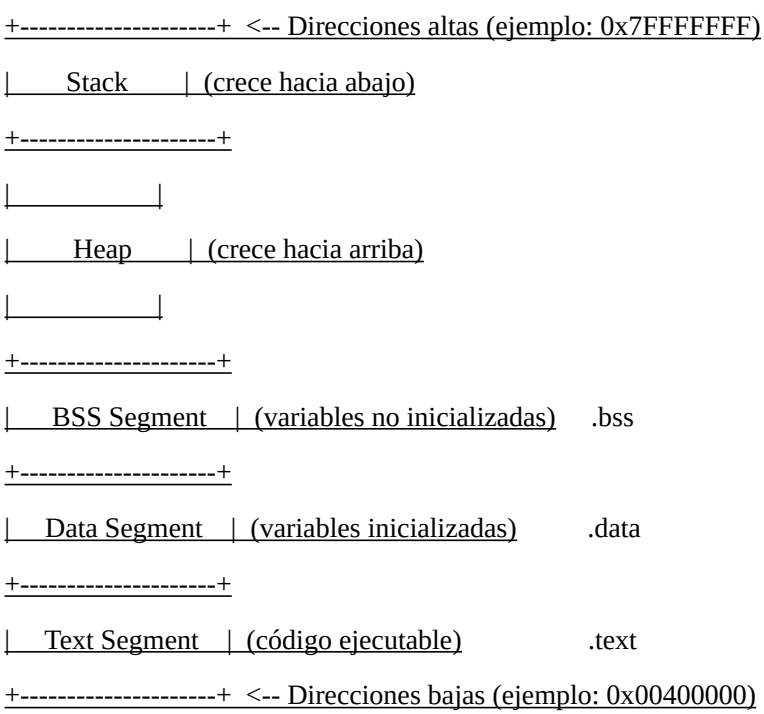
Esta función permite al usuario examinar el contenido de la memoria en una dirección específica, mostrando tanto los valores hexadecimales como su representación en caracteres, lo que es útil para depuración y análisis de memoria.

26-memory:

Organización de memoria con el segmento de texto

En un programa típico en memoria, la organización puede verse así:

Segmento	Propósito
Text Segment	Código ejecutable del programa (instrucciones máquina).
Data Segment	Variables globales y estáticas inicializadas.
BSS Segment	Variables globales y estáticas no inicializadas.
Heap	Memoria dinámica (reservada con `malloc` o `new`).
Stack	Variables locales y registros de función.



-memory -all
Imprime todo(`-funcs`, `-vars` y `-blocks`).

`-memory -funcs`
Imprime las direcciones de 3 funciones del programa (authors, pid, cwd) y 3 funciones de biblioteca (printf, strcmp, malloc).

`-memory -vars`
Imprime las direcciones de 3 variables externas (3 externas inicializadas, 3 estáticas, 3 estáticas inicializadas y 3 variables automáticas (locales)

`-memory -blocks`
Imprime la lista de bloques asignados (ImprimirListaMemoria)

`-memory -pmap`
Muestra la salida del comando `pmap` para el proceso del shell (equivalente a `vmmap` en macOS). -> Do_pmap

Esta implementación hace lo siguiente:

1. Procesa los argumentos pasados a la función para determinar qué información mostrar.

2. Si no se pasan argumentos, muestra toda la información (equivalente a `-all`).

3. Muestra las direcciones de funciones del programa y de la biblioteca si se especifica `-funcs` o `-all`.

4. Muestra las direcciones de varios tipos de variables si se especifica `-vars` o `-all`.

5. Muestra la lista de bloques de memoria asignados si se especifica `-blocks` o `-all`.

6. Llama a la función `Do_pmap()` si se especifica `-pmap`.

FUNCIÓN:

1. **Si no se pasan argumentos**, muestra todos los bloques de memoria.
2. Si el argumento es **`-funcs`**:

Imprimimos las direcciones de 3 funciones del programa: cmd_authors, cmd_pid, cmd_cwd
y 3 funciones de librería: printf, stcmp y malloc

-> memory -funcs
Funciones programa 0x5fdec6aca978, 0x5fdec6aca779, 0x5fdec6aca881
Funciones libreria 0x77de3f4600f0, 0x77de3f485b20, 0x77de3f4ad640

-FUNCIONES PROGRAMA:
En el ejemplo, sus direcciones son algo como `0x5fdec6aca978`, `0x5fdec6aca779`, `0x5fdec6aca881`.

- Estas direcciones se encuentran en el **segmento de código o texto (`.text`)** de la memoria, donde se almacenan las instrucciones ejecutables del programa.
- El **segmento de código** es una región de memoria **de solo lectura** que contiene el código binario de las funciones compiladas. Generalmente se encuentra en una región de memoria específica asignada por el compilador y el sistema operativo.

-FUNCIONES LIBRERÍA:
Estas son funciones de bibliotecas estándar como `printf`, `strcmp`, y `malloc`.

- Estas funciones no están en el código del programa, sino que se encuentran en **bibliotecas dinámicas** cargadas por el sistema operativo, como **`libc.so`** en sistemas Unix/Linux o `msvcrt.dll` en Windows.
- **Direcciones:** En el ejemplo, las direcciones son algo como `0x77de3f4600f0`, `0x77de3f485b20`, `0x77de3f4ad640`.
- Estas direcciones corresponden a las ubicaciones donde las bibliotecas estándar están cargadas en memoria. Esto ocurre durante el proceso de **carga dinámica** del programa.

AMBAS ESTÁN EN EL **SEGMENTO DE TEXTO**

3. Si el argumento es **`-vars`:**

Imprimimos las direcciones de:
-Variables locales (automáticas): auto_var1, auto_var2 y auto_var3(NUEVAS). Declaradas dentro de una función.
-Variables globales inicializadas: historico, ficheros y memoria. Definidas fuera de cualquier función
- Variables globales NO inicializadas: environ, errno, stdin. Variables globales definidas fuera de funciones, pero **no inicializadas**.
-Variables estáticas: static_var1, static_var2, y static_var3 (NUEVAS).
Variables declaradas como `static` dentro de una función o un archivo y **con valor inicializado**.
-Variables estáticas no inicializadas: static_init_var1, static_init_var2 y static_init_var3 (NUEVAS). Variables declaradas como `static` dentro de una función o un archivo y **con valor inicializado**.

Variables locales	0x7fffe024881c,	0x7fffe0248818,	0x7fffe0248814
Variables globales	0x5fdec6ada2f8,	0x5fdec6ada2f0,	0x5fdec6ada2f4
Var (N.I.)globales	0x5fdec6aebe68,	0x5fdec6aebe70,	0x5fdec6aebe78
Variables staticas	0x5fdec6ada2fc,	0x5fdec6ada300,	0x5fdec6ada304
Var (N.I.)staticas	0x5fdec6aef760,	0x5fdec6aef768,	0x5fdec6aef770

Tipo de Variable	Inicialización	Segmento en Memoria	Tiempo de Vida	Alcance
Locales	Manual	**Stack**	Hasta salir del bloque	Dentro de la función
Globales inicializadas	Manual	**Data segment**	Todo el programa	Todo el programa
Globales no inicializadas	Automática (0)	**BSS (Block Started by Symbol)**	Todo el programa	Todo el programa
Estáticas inicializadas	Manual	**Data segment**	Todo el programa	Limitado al bloque o archivo
Estáticas no inicializadas	Automática (0)	**BSS**	Todo el programa	Limitado al bloque o archivo

En el lenguaje C, las variables se almacenan en diferentes áreas de la memoria dependiendo de su tipo y ámbito. Explicación de la memoria de cada uno de los tipos de variables:

1. **Variables locales (`auto_var1`, `auto_var2`, `auto_var3`)**:
    - **Almacenamiento**: **Pila (Stack)**
    - Las variables locales son aquellas que se definen dentro de una función y su vida útil está limitada al tiempo de ejecución de la función. Por defecto, las variables locales en C son variables **automáticas** (`auto`), lo que significa que se almacenan en la pila.

    - La pila es una región de la memoria que se utiliza para almacenar variables locales y gestionar las llamadas a funciones. Las variables en la pila se crean cuando la función se invoca y se destruyen cuando la función termina.
2. **Variables globales inicializadas (`historico`, `ficheros`, `memoria`)**:
    - **Almacenamiento**: **Sección de datos (Data Segment)**
    - Las variables globales son aquellas que se definen fuera de cualquier función y son accesibles desde cualquier parte del programa. Las variables globales inicializadas se almacenan en la sección de datos de la memoria, ya que tienen un valor definido en el momento de la compilación.
    - Esta sección se divide en dos partes: una para las variables inicializadas y otra para las no inicializadas (bss). Las variables inicializadas están en la parte de datos.
3. **Variables globales NO inicializadas (`environ`, `errno`, `stdin`)**:
    - **Almacenamiento**: **Sección BSS (Block Started by Symbol)**
    - Las variables globales no inicializadas, como `environ`, `errno` o `stdin`, se almacenan en la sección BSS. Esta sección contiene variables globales que no tienen un valor asignado en el momento de la compilación, pero son inicializadas automáticamente a cero (en sistemas que lo soportan).
    - Estas variables tienen una dirección en memoria, pero su valor por defecto es cero hasta que se asignen explícitamente.
4. **Variables estáticas (`static_var1`, `static_var2`, `static_var3`)**:
    - **Almacenamiento**: **Sección de datos (Data Segment)**
    - Las variables estáticas son variables cuya vida útil dura toda la ejecución del programa, pero su ámbito está restringido a la función o archivo en el que se definen.
    - Si las variables estáticas son inicializadas (como `static_var1`, `static_var2`, `static_var3`), se almacenan en la sección de datos (junto con las variables globales inicializadas). Si no son inicializadas, se almacenan en la sección BSS.
5. **Variables estáticas no inicializadas (`static_init_var1`, `static_init_var2`, `static_init_var3`)**:
    - **Almacenamiento**: **Sección BSS**
    - Las variables estáticas no inicializadas se almacenan en la sección BSS, al igual que las variables globales no inicializadas. Aunque son estáticas, su valor inicial es cero si no se les asigna un valor

Resumen de la memoria:

- **Sección BSS**: Para variables globales y estáticas **no inicializadas (o inicializadas a cero)**.
- **Data Segment**: Para variables globales y estáticas **inicializadas** explícitamente en el código.
- **Stack (Pila)**: Para variables locales, tanto inicializadas como no inicializadas.
- **Heap**: Para la memoria dinámica (cuando se asigna memoria con `malloc()`, `calloc()`, `free()`
- **Text Segment**: `Para las funciones tanto de librería como del programa.`

4. Si el argumento es **`-pmap:`**

Llama a la función Do_pmap que hace lo siguiente:

La función `Do_pmap` es una utilidad que muestra el **mapa de memoria del proceso actual** usando comandos específicos del sistema operativo.
1-Obtiene el PID del proceso actual: La función usa `getpid()` para obtener el identificador de proceso (PID) del programa que se está ejecutando. Este PID se almacena en la variable `elpid` en forma de cadena para pasarlo como argumento a los comandos del sistema.2

2-**Crea un proceso hijo con `fork()`**:

- La función utiliza `fork()` para crear un proceso hijo. Esto es necesario porque los comandos del sistema que se ejecutan más adelante reemplazarán completamente el contexto del proceso hijo usando `execvp().`

3- Ejecución de comandos específicos del sistema:

- En el proceso hijo (cuando `pid == 0`), intenta ejecutar varios comandos conocidos para mostrar el mapa de memoria del proceso:
  - **`pmap`**: Comando disponible en sistemas Linux y Solaris.
  - **`procstat vm`**: Alternativa para sistemas FreeBSD.
  - **`procmap`**: Usado en sistemas OpenBSD.
  - **`vmmap`**: Herramienta específica de macOS.
- Si un comando falla, se prueba el siguiente. Si todos los comandos fallan, se imprime un mensaje de error con `perror()` indicando que no se pudo ejecutar el comando.

4-Espera a que el proceso hijo termine:

- El proceso padre, que no ha sido reemplazado por `execvp()`, espera con `waitpid()` a que el proceso hijo termine su ejecución antes de continuar.

Esta compilado de manera dinámica (predeterminado de gcc):

-> file shell.out

shell.out: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), **dynamically linked,** interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=f373f13b42357bafe46bb4ba97c67cae2dc58a46, for GNU/Linux 3.2.0, stripped

El archivo `shell.out` está **vinculado dinámicamente** con bibliotecas compartidas (como `libc.so.6`), y no incluye todas las bibliotecas dentro del propio ejecutable. Por lo tanto, este archivo **no es estático** sino **dinámico**.

- **`shell.out`**: El programa ejecutable.
- **`libc.so.6`**: La biblioteca estándar de C. ld-linux-x86-64.so.2 (cargador dinámico)
- **`[anon]`**: Memoria anónima, que no está asociada a ningún archivo.
- **`[pila]`**: El stack (pila) donde se guardan variables locales y datos de la función.

Si el programa se compilase de manera estática No aparecerían regiones como `libc.so.6` o `ld-linux-x86-64.so.2`, ya que todo el código estaría dentro de la región del ejecutable (`p2`).

La función `pmap` (o el comando equivalente en tu sistema) muestra el mapa de memoria del proceso, y las direcciones de memoria se organizan de menor a mayor. La organización típica de la memoria en un proceso es la siguiente, de direcciones bajas a altas:

ESPACIO VIRTUAL DE MEMORIA DEL PROGRAMA:

- 1. Texto (código del programa)
- 2. Datos (variables globales y estáticas inicializadas)
- 3. BSS (variables globales y estáticas no inicializadas)
- 4. Heap (memoria dinámica)
- 5. Bibliotecas compartidas y mapeos de memoria
- 6. Stack (pila)

- **Operaciones de archivos**

readfile- `fiche addr cont:` `lee cont bytes del archivo fiche y los copia a la dirección de memoria addr.`

writefile- fiche <- addr cont : Escribe los `cont` bytes desde la dirección de memoria `addr` en el archivo `fiche`.

read- df `addr cont`

write- df `addr cont`

27-readfile: readfile fich cont addr -> Lee `cont` bytes de un archivo en la dirección de memoria `addr`.

La finalidad de esta función **`cmd_readfile`** es leer una cantidad específica de bytes desde un archivo en el disco y copiar esos datos a una dirección de memoria proporcionada por el usuario. Es una herramienta de bajo nivel que interactúa directamente con la memoria y el sistema de archivos, probablemente usada en contextos de depuración o experimentación.

Explicación breve de su funcionamiento:

1. **Parámetros esperados:**
   - `tr[1]`: Nombre del archivo que se quiere leer.
   - `tr[2]`: Dirección de memoria donde se copiarán los datos leídos.
   - `tr[3]`: (Opcional) Cantidad de bytes a leer del archivo. Si no se pasan cont bytes, se lee todo el archivo.
2. **Validación inicial:**
   - Verifica que los argumentos necesarios estén presentes.
   - Comprueba si el archivo es accesible para lectura mediante **`access`**.
   - Convierte la dirección de memoria en formato de texto (`tr[2]`) a un puntero válido mediante la función `cadtop`.
   - Si se proporciona `tr[3]`, determina la cantidad de bytes a leer y valida que sea un valor positivo.
3. **Lectura de datos:**
   - Llama a la función `LeerFichero`, que realiza la lectura del archivo y escribe los datos en la dirección de memoria especificada.
   - Si la lectura falla, muestra un error usando **`perror`**.
   - Si tiene éxito, informa cuántos bytes se leyeron, el nombre del archivo, y la dirección de memoria donde se copiaron los datos.

Finalidad:

- Permite cargar datos desde un archivo a una dirección específica de memoria, lo cual puede ser útil para:
  - Depuración, pruebas o análisis de memoria.

- Simular cargas de datos en estructuras específicas o buffers.
- Manipulación directa de memoria en aplicaciones de bajo nivel.

Notas importantes:

- **Riesgo potencial**: Esta función es peligrosa si se usa incorrectamente, ya que permite escribir datos en cualquier dirección de memoria especificada por el usuario. Esto podría corromper la memoria del programa o causar fallos graves si no se controla adecuadamente.
- **Contexto de uso**: Normalmente, funciones de este tipo se utilizan en entornos de bajo nivel, como sistemas operativos o herramientas de depuración, y no deberían ser expuestas en aplicaciones de alto nivel sin medidas de seguridad estrictas.

AUXILIARES:

1. **`cadtop`**:
    - **Propósito**: Convierte una cadena de texto que representa una dirección de memoria (en formato hexadecimal) a un puntero.
    - **Funcionamiento**: Usa la función estándar `strtoul` para convertir la cadena a un valor numérico en base 16 (hexadecimal). Luego, convierte ese valor numérico a un puntero de tipo `void *` para que pueda ser utilizado como una dirección de memoria.
    - **Ejemplo**: Si el argumento es `"0x485a000"`, la función convierte esta cadena a un puntero que apunta a la dirección de memoria `0x485a000`.

2. LeerFichero:

Declaración de variables:

- **`struct stat s`**: Esta estructura se utiliza para obtener información sobre el archivo, como su tamaño, permisos, etc. Se usa la función `stat` para obtener esta información.
- **`ssize_t n`**: Almacena el número de bytes leídos desde el archivo.
- **`int df`**: El descriptor de archivo que se obtiene cuando se abre el archivo con la función `open`.
- **`int aux`**: Se usa para almacenar temporalmente el valor de `errno` en caso de error, para preservarlo y no perder la información del error original.
- **Obtener información del archivo y abrirlo**:
- **`stat(f, &s)`**: Se llama a la función `stat` para obtener la información del archivo especificado en `f` (nombre del archivo). Si no puede obtener esta información (por ejemplo, si el archivo no existe), la función devuelve `-1` y termina la ejecución.
- **`open(f, O_RDONLY)`**: Se abre el archivo en modo solo lectura (`O_RDONLY`). Si no puede abrir el archivo, la función también devuelve `-1` y termina la ejecución.
- **Establecer el tamaño a leer**:
- Si el parámetro `cont` (tamaño a leer) es igual a `-1` (lo cual significa que no se ha especificado un tamaño), se toma el tamaño completo del archivo usando `s.st_size` que

fue llenado por la función `stat`. Esto asegura que si no se pasa un tamaño, se lea todo el archivo.

- **Leer el archivo**:
- **`read(df, p, cont)`**: Se intenta leer `cont` bytes del archivo, comenzando desde el descriptor de archivo `df` y almacenándolos en la dirección de memoria apuntada por `p`.
    - Si la lectura es exitosa, el número de bytes leídos se almacena en `n`.
    - Si hay un error, `n` será igual a `-1`.
- **Manejo de errores**:
- Si la lectura falla (cuando `n == -1`), se guarda temporalmente el valor de `errno` (que contiene el código de error). Luego, se cierra el archivo con `close(df)` y se restablece el valor de `errno` a su valor original (el valor guardado en `aux`).
- Esto garantiza que el error que ocurrió durante la lectura no sobrescriba otros posibles errores previos.
- **Cerrar el archivo**:
- Después de intentar leer el archivo (y manejar posibles errores), el archivo se cierra usando `close(df)`.
- **Devolver el resultado**:
- Si la lectura fue exitosa, la función devuelve el número de bytes leídos (almacenado en `n`).
- Si hubo un error en cualquier parte del proceso (abrir el archivo, leerlo, etc.), devuelve `-1` para indicar que algo salió mal.

28-writefile: writefile [-o] fich addr cont | Escribe en un archivo `cont` bytes comenzando en la dirección de memoria `addr`.

La función `cmd_writefile` tiene como objetivo escribir una cantidad especificada de bytes desde una dirección de memoria a un archivo. A continuación, te explico paso a paso cómo funciona:

Propósito:

Escribir un número de bytes en un archivo. Puede sobrescribir un archivo existente si se proporciona la opción `-o`. Si no se usa la opción `-o`, se intentará crear un nuevo archivo, pero fallará si el archivo ya existe.

1. **Validación de Argumentos:**
    - La función primero valida los argumentos recibidos. Asegura que se hayan proporcionado los parámetros correctos:
        - Si se pasa `-o`, los parámetros deben incluir el archivo, la dirección de memoria y la cantidad de bytes.

        - Si no se pasa `-o`, los parámetros también deben incluir el archivo, la dirección y la cantidad de bytes.
    - Si faltan parámetros, muestra un mensaje de error y termina la ejecución.
2. **Modo Sobrescritura (opción `-o`):**
    - Si el primer argumento es `-o`, la variable `overwrite` se establece en 1 y se omite el `-o` de los parámetros. Esto indica que el archivo debe sobrescribirse si ya existe.
3. **Abrir el Archivo:**
    - La función intenta abrir el archivo en modo de escritura:
        - Si `overwrite` es 1 (modo sobrescritura), se abre el archivo con las banderas `O_WRONLY | O_CREAT | O_TRUNC`, lo que significa:
            - `O_WRONLY`: abrir para escritura.
            - `O_CREAT`: crear el archivo si no existe.
            - `O_TRUNC`: truncar el archivo (vaciarlo) si ya existe.
        - Si no se pasa `-o`, se abre con las banderas `O_WRONLY | O_CREAT | O_EXCL`, lo que significa:
            - `O_WRONLY`: abrir para escritura.
            - `O_CREAT`: crear el archivo si no existe.
            - `O_EXCL`: falla si el archivo ya existe (para evitar sobrescribirlo).
    - Si la apertura del archivo falla (por ejemplo, si el archivo ya existe y no se pasó la opción `-o`), se muestra un error.
4. **Conversión de Dirección de Memoria:**
    - La dirección de memoria donde se encuentran los datos que se desean escribir se convierte desde una cadena utilizando `cadtop()`. Si la conversión falla (es decir, si la dirección es inválida), se muestra un error y la función termina.
5. **Cantidad de Bytes a Escribir:**
    - La cantidad de bytes que se desea escribir se convierte desde una cadena utilizando `atoll()`. Si no se especifica la cantidad de bytes, se muestra un error.
    - Se valida que la cantidad de bytes sea mayor que cero. Si no lo es, se muestra un error y la función termina.
6. **Escribir en el Archivo:**
    - Si todas las validaciones pasaron, la función intenta escribir los bytes en el archivo utilizando la llamada al sistema `write()`:
        - `write(fd, addr, cont)` intenta escribir `cont` bytes desde la dirección `addr` en el archivo indicado por `fd`.
        - Si la escritura es exitosa, muestra un mensaje indicando cuántos bytes se escribieron y en qué archivo.
        - Si la escritura falla, se muestra un error.
7. **Cerrar el Archivo:**
    - Finalmente, se cierra el archivo con `close(fd)`.

29-read: read df addr cont

Esta función hace lo siguiente:

1. Comprueba que se hayan proporcionado los parámetros necesarios.
2. Convierte los parámetros a los tipos de datos adecuados.
3. Verifica que el descriptor de archivo esté dentro del rango válido (0-19).
4. Verifica que la dirección de memoria y el número de bytes a leer sean válidos.
5. Abre el archivo asociado al descriptor de archivo.
6. Lee el número especificado de bytes desde el descriptor de archivo en la dirección de memoria.
7. Cierra el archivo después de leer los datos.

La función `cmd_read` está diseñada para leer datos de un archivo ya abierto y almacenarlos en una dirección de memoria específica. Esta función valida los parámetros de entrada, verifica las condiciones necesarias y realiza la lectura si todo es correcto. Su finalidad principal es ofrecer una manera controlada de leer datos desde un archivo abierto (identificado por un descriptor de archivo) hacia una región de memoria.

Explicación paso a paso

1. **Validación de parámetros de entrada**:
    - La función espera tres argumentos en el array `tr`:
        1. `tr[1]`: Descriptor de archivo (`df`) como una cadena.
        2. `tr[2]`: Dirección de memoria (`addr`) como una cadena.
        3. `tr[3]`: Cantidad de bytes a leer (`cont`) como una cadena.
    - Si alguno de estos argumentos falta, la función informa al usuario y termina.
2. **Conversión de parámetros**:
    - Convierte los valores de entrada de cadenas de texto a tipos adecuados:
        1. `atoi(tr[1])`: Convierte el descriptor de archivo de cadena a entero.
        2. `cadtop(tr[2])`: Convierte la dirección de memoria (en formato de cadena) a un puntero.
        3. `atoll(tr[3])`: Convierte la cantidad de bytes a leer a un entero largo (`size_t`).
3. **Validación del descriptor de archivo**:
    - Comprueba que el descriptor de archivo (`df`) está dentro de un rango válido (0-19 en este caso).
    - Verifica que el descriptor de archivo esté en la lista de archivos abiertos utilizando `nombreFicheroDescriptor(df, ficheros)`. Si no está, informa al usuario y termina.
4. **Validación de la dirección de memoria**:
    - Verifica que la dirección de memoria (`addr`) no sea nula. Si la conversión falló, la función informa al usuario y termina.
5. **Validación de la cantidad de bytes**:

- Comprueba que la cantidad de bytes a leer (`cont`) sea mayor que cero. Si no, informa al usuario y termina.

6. **Lectura desde el archivo**:
    - Utiliza la función del sistema `read(df, addr, cont)` para leer datos del archivo identificado por el descriptor `df` y almacenarlos en la memoria apuntada por `addr`.
    - `read` devuelve la cantidad de bytes leídos o `-1` si ocurre un error.
    - Si ocurre un error durante la lectura, muestra un mensaje con el error correspondiente (`strerror(errno)`).
    - Si la lectura es exitosa, informa al usuario la cantidad de bytes leídos, el descriptor de archivo y la dirección de memoria donde se copiaron los datos.

BASICAMENTE BUSCA EL DESCRIPTOR EN LA LISTA DE DESCRIPTORES Y COPIA EL CONTENIDO DE ESE ARCHIVO EN LA DIRECCIÓN DE MEMORIA. SE RECOMIENDA HACER UN OPEN ANTES PARA VER LA LISTA DE DESCRIPTORES ABIERTOS

30-write: write df addr cont

Write hace las mismas comprobaciones que read, pero en vez de utiizar read, utiliza write, que se utilizá para poder modificar el contenido del archivo en cuestión de la dirección de memoria.

31-recurse:

La función llama a una función recursiva Recursiva con un valor de n proporcionado, que será el número de recursiones que hará recursiva.

AUXILIAR RECURSIVA:

Esta función es un ejemplo clásico de recursión que muestra cómo se comportan las variables **locales automáticas** (en el stack) y las **variables estáticas** (en el segmento de datos estático) en términos de memoria durante las llamadas recursivas.

1. **Parámetro `n`:** Muestra cómo el parámetro de la función cambia en cada llamada recursiva y dónde está ubicado en la memoria.
2. **Variables locales (automáticas):** Son las variables que se declaran dentro de una función. Se alojan en el *stack* y tienen un espacio distinto en memoria para cada llamada recursiva.
3. **Variable `estatico`:** Es una variable estática que se aloja en el *segmento de datos estático*. Esta variable conserva su valor a lo largo de todas las llamadas a la función (tanto recursivas como no recursivas).

La función imprime las direcciones de memoria de estas variables y del parámetro `n` para ilustrar cómo se asignan y dónde están almacenadas.

-> recurse 3 **V.LOCAL HEAP** **V.ESTÁTICA DS**

parametro: 3(0x7fff054cbc9c) array 0x7fff054cbca0, arr estatico 0x5fe48eacd420

parametro: 2(0x7fff054cb46c) array 0x7fff054cb470, arr estatico 0x5fe48eacd420

parametro: 1(0x7fff054cac3c) array 0x7fff054cac40, arr estatico 0x5fe48eacd420

parametro: 0(0x7fff054ca40c) array 0x7fff054ca410, arr estatico 0x5fe48eacd420

Tipo de Variable	Inicialización	Segmento en Memoria	Tiempo de Vida	Alcance
Locales	Manual	**Stack**	Hasta salir del bloque de la función, cambia dirección en cada llamada recursiva.	Dentro de la función
Estáticas inicializadas	Manual	**Data segment**	Todo el programa. Se mantiene su dirección de memoria a lo largo del programa	Limitado al bloque o archivo

RESUMEN FUNCIONES PREDEFINIDAS P2:

#include <stdio.h> // printf, perror, remove

#include <string.h> // strcmp, strdup

#include <stdlib.h> // malloc, free, exit, **strtoul** (string to unsigned long)**, atoll** (ASCII to long long)

#include <unistd.h> // chdir, getpid, getuid, getcwd, rmdir, execv, **access, **read y write, *lseek**

#include <time.h> // time, ctime

#include <fcntl.h> // open, close, read

#include <sys/stat.h> // stat, lstat,

#include <errno.h> // errno, strerror

#include <sys/mman.h> //**mmap, munmap, PROT_WRITE, PROT_READ, PROT_EXEC**

*mmap = mapea memoria / munmap = desmapea memoria

<sys/shm.h> // **shmget** (obtiene/crea un segmento de memoria) , **shmat** (adjunta un segmento de memoria compartida a un proceso), **shmdt** (desacopla un segmento de memoria compartida), **shmctl** (Controla el comportamiento de los segmentos de memoria compartida, por ejemplo los elimina o cambia sus permisos)

<sys/ipc.h> // se necesita también la librería anterior:

//**IPC_RMID** (Se utiliza en la función `shmctl()` para eliminar un segmento de memoria compartida), **IPC_CREAT, IPC_EXCL, SHM_RDONLY**...

*read y write:

Lectura (read):

- **Accede al archivo o flujo de datos**: Se conecta a un archivo ya abierto (identificado por un descriptor de archivo) o a otro flujo de datos (como sockets o pipes).
- **Copia el contenido**: Lee hasta la cantidad de bytes solicitada desde el archivo y los coloca en un búfer de memoria proporcionado por el programa.
- **No modifica el archivo**: Los datos permanecen intactos en el archivo de origen, ya que solo se realiza una copia de los datos en memoria.

Escritura (write):

- **Accede al archivo o flujo de datos**: Utiliza un descriptor de archivo (o flujo) para determinar dónde escribir.
- **Copia los datos desde la memoria**: Toma el contenido de un búfer de memoria especificado por el programa y lo escribe en el archivo o flujo asociado.
- **Modifica el archivo**: Los datos escritos sobrescriben el contenido existente del archivo (a menos que se escriban en una posición nueva, dependiendo del offset o modo del archivo).

*lseek:

Se utiliza aquí para obtener el tamaño del archivo (al mover el puntero al final) y luego restablecer el puntero al inicio del archivo para futuras operaciones.

-P3:

-Lista búsqueda:

Es una lista en la que añadiremos los directorios quw quereamos y en la que la funcionaron Ejecutable de utils buscará archivos ejecutables para ejecutar luego. Si hacemos search -path añadiremos a esta lista todo el PATH, que justamente contiene ejecutables. Con el path podemos usar nuestra shell los comandos básicos de linux. -> La lista se maneja con el COMANDO SEARCH

DECLARACIÓN:

```
// Estructura de un nodo
typedef struct NodoBusqueda {
    char *directorio;                   //un nodo contiene un directorio
    struct NodoBusqueda *siguiente;     // un puntero al siguiente nodo
} NodoBusqueda;

// Estructura de la lista de búsqueda
typedef struct {
    NodoBusqueda *cabeza;
    NodoBusqueda *actual;               // Para iterar sobre la lista
} ListaBusqueda;

// Funciones para manejar la lista de búsqueda
void inicializarListaBusqueda(ListaBusqueda *);          //inicializa la lista
bool directorioExiste(ListaBusqueda *, const char *);    // comprueba si existe un directorio
void agregarDirectorio(ListaBusqueda *, const char *);   //añade un directorio a la lista
bool eliminarDirectorio(ListaBusqueda *, const char *);  //elimina un directorio de la lista
void limpiarListaBusqueda(ListaBusqueda *);              //elimina todo el contenido de la lista
void mostrarListaBusqueda(const ListaBusqueda *);        //muestra la lista por orden de inserción

// Funciones para iterar sobre la lista
char *SearchListFirst(ListaBusqueda *);
char *SearchListNext(ListaBusqueda *);
```

// Función para importar directorios del PATH

void importarPATH(ListaBusqueda *); // importa los directorios del PATH a la lista. Utiliza la función getenv (PATH), para obtener el contenido de la variable de entorno que se llame PATH. Utilizamos strtok para quedarnos con el contenido simplemente y obtener asi todos los directorios (que están separados por :)

Con arg3 main PATH=/home/ismael/.opam/ocaml-latest/bin:/home/ismael/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin(0x7ffc6611545e) @0x7ffc66114728

-Lista procesos bg (en segundo plano)

Utilizamos esta lista para añadir todos los procesos que se ejecuten en nuestro shell en segundo plano -> utilizando los comandos back y backpri. Utilizamos listjobs para llamar a la lista y deljobs para eliminar partes de la lista. deljobs -term elimina los procesos terminados de manera normal y -sig elimina los procesos eliminados por señal.

DECLARACIÓN:

```
// Enumerado para el estado de las señales
typedef enum {
    FINISHED,          // TERMINADO. El proceso fue acabado
    STOPPED,           // DETENIDO. El proceso se paró  (con SIGSTOP por ejemplo)
    SIGNALED,          // TERMINADO CON SEÑAL. El proceso fue acabado con señal.
    ACTIVE             // El proceso sigue ACTIVO y se está ejecutando.
} ProcessStatus;

// Estructura para representar un proceso en segundo plano
typedef struct BackgroundProcess {
    pid_t pid;                      // PID del proceso
    time_t launch_time;             // Tiempo de lanzamiento (hora de inicio)
    char user[50];                  // Usuario que lanzó el proceso
    ProcessStatus status;           // Estado del proceso
    int return_value;               // Valor de retorno o señal
    char* command_line;             // Línea de comando ejecutada
    int priority;                   // Prioridad del proceso
    struct BackgroundProcess* next;     // Puntero al siguiente proceso
} BackgroundProcess;
```

// Funciones para manejar la lista de procesos en segundo plano

void Init_backgroundProcesses(BackgroundProcess*); // Inicializa la lista

void Add_backgroundProcess(BackgroundProcess*, pid_t, const char*,int); // Añade un proceso

void Update_processStatus(BackgroundProcess*, pid_t, ProcessStatus, int); // Actualiza el estado de un proceso

void Remove_backgroundProcess(BackgroundProcess*, pid_t); // Elimina un proceso de la lista

void Imprimir_backgroundProcesses(BackgroundProcess*); // Imprime los procesos en segundo plano

void Delete_backgroundProcesses(BackgroundProcess*); // Elimina todos los procesos de la lista

¿Qué es el UID?

El **UID** (User Identifier) es un número único que el sistema operativo asigna a cada usuario en un sistema. **Cada usuario tiene su propio UID**, y el sistema usa estos números para identificar quién está haciendo qué en el sistema (por ejemplo, si un usuario tiene permiso para leer o escribir archivos).

- **UID 0** es **siempre el usuario root**, que tiene permisos administrativos completos en el sistema.
- Los **demás usuarios** tienen UID diferentes, que el sistema usa para saber qué pueden hacer.

UID real vs UID efectivo:

1. UID Real (RUID):

- El **UID real** es el **UID** del **usuario** que lanzó el proceso.
- Es como el "dueño original" del proceso. Si tú inicias un programa en tu máquina, el UID real será el tuyo.
- **Ejemplo**: Si tu usuario tiene UID 1001, entonces el UID real del proceso será 1001.

2. UID Efectivo (EUID):

- El **UID efectivo** es el **UID** que el sistema usa para verificar si un proceso tiene permisos para hacer algo (como leer o escribir un archivo).
- A veces, un proceso puede **asumir el UID de otro usuario** para poder hacer cosas que normalmente no podría hacer. Por ejemplo, un programa que ejecuta tareas como **root** (UID 0) aunque lo haya lanzado un usuario normal.
    - **Ejemplo típico**: Un programa lanzado por un usuario normal puede cambiar su **UID efectivo** a 0 (root) para poder hacer tareas que solo el superusuario puede hacer (como instalar software). Después de hacer esas tareas, el programa vuelve a su **UID real**.

Resumen simple:

- **UID real** = Es el UID del **usuario** que ejecuta el proceso.
- **UID efectivo** = Es el UID que el sistema usa para **decidir si el proceso puede hacer algo** (por ejemplo, si tiene permisos de administrador).

Ejemplo:

Imagina que eres un **usuario normal** con UID 1001. Cuando inicias un programa, su **UID real** será **1001**.

Ahora, si ese programa necesita hacer algo que solo **root** puede hacer (como cambiar configuraciones importantes), el **UID efectivo** del programa puede cambiar a **0** (root), aunque el UID real siga siendo **1001**.

Cuando el programa termina, su **UID efectivo** vuelve a ser **1001**, como el de su **usuario real**.

Para poder ver los usuarios que TENEMOS ACTUALMENTE EN EL SISTEMA (fuera del shell), podemos usar **cat /etc/passwd** y nos enseñara los usuarios y sus credenciales.

Para crear un nuevo usuario:

1-Crear usuario:

sudo useradd user1

2-Mover ejecutable y cambiar permisos:

cp shell.out /tmp

chmod 4755 /tmp/shell.out

3-Cambiar de usuario:

su user1

Contraseña: ...

- ## Funciones de credenciales

32-getuid:

Esta función muestra el **UID real** y el **UID efectivo** del proceso que está ejecutando el shell.

¿Qué hace?

- **`getuid()`**: Obtiene el **UID real** (el UID del usuario que lanzó el proceso).
- **`geteuid()`**: Obtiene el **UID efectivo** (el UID con el que el sistema verifica los permisos del proceso).
- Usa **`getpwuid()`** para obtener el **nombre de usuario** asociado a cada UID.

Salida esperada en el shell:

Credencial real: 1000, (ismael)

Credencial efectiva: 1000, (ismael)

33-setuid: setuid [-l] <uid|username>

Establece la credencial efectiva del proceso (-l para el nombre). Sin argumento se le pasa el UID.

En este caso solo se cambia la CE con geteuid. getuid cambia las dos credenciales a la vez.

La función `cmd_setuid` permite cambiar la **credencial efectiva** de un proceso a un nuevo UID. Dependiendo de los argumentos proporcionados, la función puede cambiar el UID efectivo de dos maneras:

1. **Por nombre de usuario**: Si se pasa el argumento `-l` seguido de un nombre de usuario, la función busca el UID correspondiente a ese usuario en la base de datos del sistema. Para ello, utiliza la función **`getpwnam`** que devuelve la información del usuario, incluida su ID de usuario (UID). Luego, cambia la credencial efectiva del proceso a ese UID usando la función `seteuid`.
2. **Por UID numérico**: Si el primer argumento no es `-l`, la función interpreta el argumento como un UID numérico. Utiliza `strtol` para convertir la cadena proporcionada en un número entero y luego valida que la conversión fue exitosa. Si el valor es válido, cambia el UID efectivo del proceso con `seteuid`.

Si el cambio de UID es exitoso, la función muestra el nuevo UID efectivo y el nombre de usuario correspondiente (si está disponible). Si ocurre un error en cualquiera de los pasos, como un nombre de usuario no válido o un UID incorrecto, se muestra un mensaje de error adecuado.

Las funciones clave utilizadas en `cmd_setuid` incluyen:

- **`getpwnam`**: Para obtener la información del usuario por nombre.
- **`strtol`**: Para convertir una cadena en un número (UID).
- **`seteuid`**: Para cambiar el UID efectivo del proceso.
- **`geteuid`** y **`getpwuid`**: Para obtener y mostrar el UID efectivo actual y su nombre asociado.

Casos para ismael (UID 1000): el archivo lo creó Ismael

1. **ismael ejecuta `setuid -l user1`**:
    - **Ejecutado por**: Ismael -> CR: ismael / CE : ismael
    - **Credencial real**: 1000 (ismael)
    - **Credencial efectiva**: 1001 (user1)
    - **Explicación**: Cambia el UID efectivo de Ismael a `user1` (UID 1001).
2. **ismael ejecuta `setuid 1001`**:
    - **Ejecutado por**: ismael -> CR: ismael / CE : ismael
    - **Credencial real**: 1000 (ismael)
    - **Credencial efectiva**: 1001 (user1)
    - **Explicación**: Cambia el UID efectivo de Ismael a 1001 (user1).
3. **Ismael ejecuta `setuid 1000`**:
    - **Ejecutado por**: ismael CR: ismael / CE : ismael
    - **Credencial real**: 1000 (ismael)
    - **Credencial efectiva**: 1000 (ismael)
    - **Explicación**: El UID efectivo sigue siendo el mismo que el real.
4. **Ismael ejecuta `setuid 1002`** (UID no existente):
    - **Ejecutado por**: ismael -> CR: ismael / CE : ismael
    - **Credencial real**: 1000 (ismael)
    - **Credencial efectiva**: 1000 (ismael)
    - **Explicación**: Error, el UID 1003 no existe, no cambia el UID efectivo.

Casos para user1 (UID 1001): el archivo lo creó Ismael

1. **user1 ejecuta `setuid -l ismael`**:
    - **Ejecutado por**: user1 -> CR: user1 / CE : user1
    - **Credencial real**: 1001 (user1)
    - **Credencial efectiva**: 1000 (ismael)
    - **Explicación**: Cambia el UID efectivo de `user1` a `ismael` (UID 1001).
2. **user1 ejecuta `setuid 1000`**:
    - **Ejecutado por**: user1 -> CR: user1 / CE : user1
    - **Credencial real**: 1001 (user1)
    - **Credencial efectiva**: 10010(Ismael)
    - **Explicación**: Cambia el UID efectivo de `user1` a 1001 (Ismael).
3. **user1 ejecuta `setuid 1001`**:
    - **Ejecutado por**: user1
    - **Credencial real**: 1001 (user1) -> CR: user1 / CE : user1
    - **Credencial efectiva**: 1001 (user1)
    - **Explicación**: El UID efectivo sigue siendo el mismo que el real.
4. **user1 ejecuta `setuid 1003`** (UID no existente):
    - **Ejecutado por**: user1 -> CR: user1 / CE : user1
    - **Credencial real**: 1001 (user1)
    - **Credencial efectiva**: 1001 (user1)
    - **Explicación**: Error, el UID 1003 no existe, no cambia el UID efectiva-

- ## Funciones del PATH y de las variables de entorno

Las **variables de entorno** son una forma de almacenar información en el sistema operativo que puede ser usada por programas o procesos. Son como "notas" o "valores" que el sistema guarda para ayudar a los programas a saber cosas importantes, como:

- Dónde están ciertos archivos.
- Cómo deben comportarse los programas.
- Información sobre el usuario o el sistema.

VARIABLES DE RETORNO BÁSICAS:

1-PATH: Una lista de directorios donde el sistema busca programas ejecutables.

2-HOME: La ruta al directorio personal del usuario.

3-SHELL: La ruta al programa que actúa como tu shell (terminal).

4-USER: El nombre del usuario actual que ha iniciado sesión.

5-LANG: El idioma y codificación que debe usar el sistema.

6-PWD: El directorio actual donde estás trabajando en la terminal.

7- _: donde se almacena el ejecutable de nuestro código.

MANERAS DE ACCEDER A LAS VARIABLES DE ENTORNO EN C:

1-Variable externa **environ**(extern char **environ): `environ` está declarada en otro lugar (biblioteca estándar del sistema operativo, <unistd.h>) y no se define en tu programa directamente. Es un puntero a un arreglo de cadenas (arreglo de punteros). Cada cadena es una variable de entorno en formato: **NOMBRE=VALOR**

2-Tercera variable del main **envp** (environ se utiliza para obtener la tercera variable del main, char *envp []): Es un arreglo de cadenas que contiene las variables de entorno pasadas al programa. Cada cadena tiene la forma **`NOMBRE=VALOR.`** Similar a environ, pero envp solo está disponible en el main y es un argumento que recibe el programa.

3-Usando la función **getenv**(es una funcion estándar de la librería <stdlib.h> que se utiliza para obtener el valor de una variable de entorno específica): Recibe el nombre de la variable como argumento y devuelve su valor o `NULL` si no se encuentra la variable.

34-showvar: showvar v1 v2 ... (v1, v2 etc son variables de entorno. Se pueden ver con showvar)

La función `cmd_showvar` tiene como objetivo mostrar las variables de entorno del sistema, su valor y sus direcciones de memoria. Dependiendo de si se pasan o no argumentos al comando, la función actúa de una manera diferente:

1. **Si no se pasan argumentos:**
    - La función recorre todas las variables de entorno disponibles (que están en el arreglo `envp[]`), y para cada variable, imprime:
        - La dirección de memoria de la variable de entorno en `envp[]`.
        - El índice de la variable dentro del arreglo.
        - La dirección de memoria de la cadena que contiene la variable de entorno (en el formato `NOMBRE=VALOR`).
        - El contenido de la variable, es decir, su nombre y valor.
2. **Si se pasan argumentos (nombres de variables de entorno):**
    - La función busca cada nombre de variable especificado en los argumentos en tres lugares diferentes:

    a. **En el arreglo `envp[]`:**

        - Este arreglo contiene las variables de entorno que fueron pasadas al programa al ejecutarlo.
        - Si la variable de entorno se encuentra en `envp[]`, se extrae y muestra su valor, junto con las direcciones de memoria del arreglo y el valor de la variable.

    b. **En el arreglo global `environ[]`:**

        - Este arreglo es una variable global que también contiene las variables de entorno, pero no es pasada como parámetro al programa, se encuentra disponible globalmente.
        - Si se encuentra la variable en `environ[]`, se muestra su valor y las direcciones de memoria, similar al paso anterior.

    c. **Usando la función `getenv()`:**

        - `getenv()` es una función estándar que recibe el nombre de una variable de entorno y devuelve su valor si existe. Si se encuentra la variable, se imprime su valor y la dirección de memoria asociada al valor.

Comparación entre `envp[]` y `environ`

Característica	`envp[]`	`environ`
Acceso	Se recibe como argumento en `main`	Variable global externa
Disponibilidad	Solo si se declara en `main`	Siempre disponible
Uso	Argumento explícito en funciones	Acceso global en cualquier parte del programa

Ejemplo salida general:

-> showvar

0x7ffd929a52d8->main arg3[0]=(0x7ffd929a601e) SHELL=/bin/bash

(1) (2) (3) (4)

1-Es la dirección de memoria donde se encuentra el puntero que apunta al primer elemento de `arg3[]`, que es el arreglo de argumentos pasados al programa.

2-Hace referencia al primer argumento pasado al programa. `arg3` es el nombre que el código utiliza para referirse al arreglo de argumentos del `main`.

3-Es la dirección de memoria donde se encuentra almacenada la cadena de texto que contiene el valor de `arg3[0]`.

4-NOMBRE = VALOR (nombre de la variable de entorno = valor asignado a esa variable).

Ejemplo salida variable/s específica/s:

-> showvar _

ismael@asus(~/Escritorio/SO/PRÁCTICAS/SHELL_REFERENCIA/0LINUX_P3)-> showvar _

Con arg3 main _=./shell.out(0x7ffd929a6fde) @0x7ffd929a54f0

Con environ _=./shell.out(0x7ffd929a6fde) @0x7ffd929a54f0

Con getenv ./shell.out(0x7ffd929a6fe0)

(1) (2) (3) (4)

1-Tipo que usamos para acceder a la variables

2-NOMBRE = VALOR

3-Corresponden a la dirección del **valor** de la variable en memoria,

4-corresponden a la **dirección de la entrada** de la variable en la tabla de entorno o estructura en memoria

¿PORQUE LAS DIRECCIONES DE MEMORIA ARG3 Y ENVIRON SON LAS MISMAS?

- **Ambos apuntan a la misma memoria**: La lista de variables de entorno en `environ` y las variables de entorno pasadas a través de `envp` (o `arg3`) a menudo son apuntadas y gestionadas por el sistema operativo de manera similar.
- **En muchos sistemas operativos**: El sistema operativo puede asignar las mismas direcciones de memoria para las variables de entorno accesibles a través de `arg3[]` y `environ[]`. Por lo tanto, es normal que estas dos listas de variables apunten a la misma zona de memoria.

¿Dónde se guardan las en la memoria virtual?

En términos de **memoria virtual**, las variables de entorno generalmente se guardan en una sección de memoria llamada **segmento de datos** o **heap** del proceso. Cuando un programa se ejecuta, el sistema operativo copia las variables de entorno desde el sistema a esta zona para que estén disponibles mientras se ejecuta el programa.

Resumen

- Las **variables de entorno** se almacenan generalmente en la **pila del proceso** (tanto con environ como con envp[])
- El sistema operativo copia las variables de entorno en la pila antes de invocar `main`.
- **`envp`** es un puntero a un arreglo que contiene las direcciones de las cadenas con las variables de entorno.
- La pila es usada porque es **eficiente**, **local al proceso**, y simplifica el acceso

35-changevar: changevar [-a | -e | -p] var valor: Cambia el contenido de una variable

-a: accede mediante el tercer argumento del main -> envp[].

-e: accede mediante la variable externa environ.

-p: utiliza la función putenv para CREAR una NUEVA variable con el valor que le pasamos.

La función `cmd_changevar` permite cambiar o crear variables de entorno en función de un **modo especificado** (`-a`, `-e`, o `-p`):

1. **Validación de argumentos**: Verifica que el usuario proporcione los argumentos necesarios (`modo`, `var`, y `valor`). Si no, muestra un mensaje de uso.
2. **Modos de operación**:
    - **`-a`**: Busca y modifica una variable en el entorno local (`envp[]`). Si no existe, muestra un error.
    - **`-e`**: Busca y modifica una variable en el entorno global (`environ`). Si no existe, muestra un error.
    - **`-p`**: Crea o modifica una variable en el entorno global usando `putenv`.
3. **Mensajes y errores**: Informa al usuario si la operación tuvo éxito o si ocurrió un error (modo inválido, variable no encontrada, o problemas de memoria).
4. **Gestión de memoria**: En el modo `-p`, reserva memoria para `putenv` y deja que esta función la administre.

36-subsvar: subsvar [-a|-e] v1 v2 val : cambia la variable1 por variable2 = VALOR

La variable 1 tiene que ser una variable de entorno existente, es decir, estar en el showvar.

La variable 2 es una variable no existente que nosostros creamos a la que le asignamos un valor y va a ser lo que reemplace a la variable 1.

-a: accedemos mediante el tercer argumento del main -> envp[]

-e: accedemos mediante la variable externa -> environ

La función `cmd_subsvar` tiene como objetivo reemplazar una variable de entorno por otra con un nuevo valor, dependiendo del entorno que se le pase (local o global). A continuación, se explica paso a paso cómo funciona la función:

1. **Verificación de parámetros**: Al principio, la función verifica si se han proporcionado suficientes parámetros. Si alguno de los parámetros requeridos está ausente (modo, var1, var2 o valor), la función imprime un mensaje indicando el formato correcto de uso y termina la ejecución.
2. **Asignación de parámetros a variables**: Si los parámetros son válidos, se asignan a variables locales:
    - `modo`: Determina si se usará el entorno local (`-a`) o global (`-e`).
    - `var1`: La variable de entorno que se desea sustituir.
    - `var2`: La nueva variable de entorno que se va a crear.
    - `valor`: El nuevo valor que se asignará a `var2`.
3. **Selección del entorno (local o global)**: Según el valor de `modo`:
    - Si `modo` es `-a`, la función usa el entorno local (`envp`).
    - Si `modo` es `-e`, usa el entorno global (`environ`). Si el modo no es válido, la función imprime un error y termina.
4. **Búsqueda de `var1` en el entorno**: Se utiliza la función `BuscarVariable` para buscar la variable `var1` en el entorno seleccionado. Si no se encuentra (`pos1 == -1`), la función imprime un mensaje indicando que `var1` no existe y termina.
5. **Verificación de la existencia de `var2`**: La función también verifica si la variable `var2` ya existe en el entorno. Si `var2` ya está presente (`pos2 != -1`), la función imprime un mensaje indicando que no se puede proceder con la sustitución porque `var2` ya existe en el entorno.
6. **Creación de la nueva variable**: Si `var1` existe y `var2` no, la función procede a crear la nueva variable en el entorno. Se reserva memoria para la nueva variable de entorno concatenando `var2` y `valor` (se incluye el carácter `=` entre ambos). La memoria se asigna usando `malloc`, y se utiliza `sprintf` para almacenar el resultado de la concatenación en la variable `nueva_var`.
7. **Sustitución de la variable**: La función reemplaza la variable en el entorno seleccionando la posición de `var1` (`pos1`) en el arreglo del entorno y asignándole la nueva variable (`nueva_var`).

8. **Manejo de errores en la asignación de memoria**: Si ocurre un error al asignar memoria para `nueva_var` (si `malloc` devuelve `NULL`), se imprime un mensaje de error y la función termina.
9. **Mensaje de confirmación**: Finalmente, la función imprime un mensaje indicando que la variable `var1` ha sido sustituida por `var2` con el valor proporcionado. También muestra si el cambio se realizó en el entorno local o global.

AUXILIARES:

1. **Función `BuscarVariable`**

Objetivo: Buscar una variable dentro de un entorno (un array de cadenas `e[]`), devolviendo la posición en la que se encuentra la variable, o `-1` si no se encuentra.

Pasos que sigue:

1. **Definir variables:**
   - `int pos = 0;`: Inicializa la posición del array en 0.
   - `char aux[MAXVAR];`: Define un buffer (`aux`) para almacenar el nombre de la variable junto con el signo igual `=`.
2. **Preparar la variable a buscar:**
   - `strcpy(aux, var);`: Copia el nombre de la variable que se quiere buscar (`var`) en `aux`.
   - `strcat(aux, "=");`: Añade un signo igual (`=`) al final de `aux`. Así se busca la variable en formato "nombre=valor".
3. **Recorrer el entorno `e[]`:**
   - `while (e[pos] != NULL)`: Itera a través de las cadenas de entorno `e[]`, buscando la variable en cada posición.
     - `strncmp(e[pos], aux, strlen(aux))`: Compara el principio de la cadena en `e[pos]` con `aux`. Si la cadena de entorno comienza con `aux`, significa que se ha encontrado la variable.
     - `if (!strncmp(e[pos], aux, strlen(aux)))`: Si encuentra una coincidencia, devuelve la posición de la variable (`return (pos)`).
4. **Si no se encuentra la variable:**
   - `errno = ENOENT;`: Si no se encuentra la variable en todo el entorno, establece un código de error `ENOENT` (Error No Existe).
   - `return(-1);`: Devuelve `-1` para indicar que la variable no fue encontrada.

---

2. Función `CambiarVariable`

Objetivo: Cambiar el valor de una variable de entorno dentro de un entorno dado (`e[]`), utilizando la función `BuscarVariable` para localizar la variable a modificar.

Pasos que sigue:

1. **Buscar la variable a modificar:**
    - Llama a `BuscarVariable(var, e)`: Busca la variable `var` en el entorno `e[]`. Si la variable no se encuentra, `BuscarVariable` devolverá `-1`.
    - `if ((pos = BuscarVariable(var, e)) == -1)`: Si no encuentra la variable, la función devuelve `-1` y sale de la función con `return(-1);`.
2. **Reservar memoria para el nuevo valor:**
    - `char *aux;`: Define un puntero `aux` para almacenar la nueva cadena que representará la variable con su nuevo valor.
    - `if ((aux = (char *)malloc(strlen(var) + strlen(valor) + 2)) == NULL)`: Reserva memoria para la nueva cadena, que será la concatenación de `var`, el signo igual `=`, y `valor`. Si no se puede reservar memoria, devuelve `-1`.
3. **Construir la nueva variable:**
    - `strcpy(aux, var);`: Copia el nombre de la variable `var` a `aux`.
    - `strcat(aux, "=");`: Añade un signo igual (`=`) a la cadena `aux`.
    - `strcat(aux, valor);`: Añade el nuevo valor `valor` a la cadena `aux`.
4. **Actualizar el entorno:**
    - `e[pos] = aux;`: Sustituye la entrada en `e[]` en la posición `pos` con la nueva cadena `aux` (que contiene `var=valor`).
5. **Devolver la posición:**
    - `return (pos);`: Devuelve la posición donde se ha hecho el cambio (la misma en la que se encontraba la variable originalmente).

36-environ: environ [-environ | -addr]: Muestra el entorno del proceso

environ solo hace lo mismo que showvar solo. Muestran las variables de entorno y sus direcciones mediante el tercer argumento del main arg3 -> envp[]

-environ: muestra lo mismo que environ y showvar normal pero accediendo desde environ en este caso -> environ []
-addr: muestra el valor y donde se almacenan environ y el 3er arg main (muestra sus direcciones de memoria).

La función `cmd_environ` se encarga de mostrar información sobre las variables de entorno de un proceso. A continuación, se explica el comportamiento según los argumentos que recibe:

1. **Cuando no se pasan argumentos adicionales** (`tr[1] == NULL`): En este caso, la función recorre el arreglo `envp`, que contiene las variables de entorno pasadas al programa a través de los argumentos de la función `main`. Para cada variable de entorno en `envp`, la función imprime su dirección de memoria, el índice en el arreglo, la dirección de la variable de entorno y el valor de la variable de entorno.

   0x7ffde9b0->main arg3[0]=(0x7ffde9b0) USER=ismael

   0x7ffde9c0->main arg3[1]=(0x7ffde9c0) PATH=/usr/bin:/bin...

2. **Cuando** se pasa el argumento **`-environ`** (`strcmp(tr[1], "-environ") == 0`): Si se recibe el argumento `-environ`, la función recorre el arreglo global `environ`, que también contiene las variables de entorno, pero es accesible en todo el programa, no solo en la función `main`. Al igual que en el primer caso, imprime la dirección de memoria de cada elemento, su índice y el valor de la variable de entorno contenida en `environ`.

   0x7ffde9b0->environ[0]=(0x7ffde9b0) USER=ismael

   0x7ffde9c0->environ[1]=(0x7ffde9c0) PATH=/usr/bin:/bin

   ...

3. **Cuando se pasa el argumento** `-addr` (`strcmp(tr[1], "-addr") == 0`): Si se recibe el argumento `-addr`, la función imprime las direcciones de memoria tanto de `environ` como de `envp`, y también muestra las direcciones de memoria donde están almacenados estos arreglos. Esto proporciona información sobre las ubicaciones en la memoria de los arreglos que contienen las variables de entorno.

   environ: 0x7ffd545ba2e8 (almacenado en 0x5cb3959cd518)

   main arg3: 0x7ffd545ba2e8 (almacenado en 0x7ffd544b9b10)

(1) (2)

1- La primera es la dirección del puntero (envp y environ). Ambas direcciones son iguales.

2- La segunda es la dirección donde se almacena dicho puntero.

-environ es una dirección global no inicializada por lo que se guarda en el segmento de datos (en el bss).

-envp (tercer argumento del main) se almacena en la pila (stack). Esto se debe a que es una variable local de la función main. Recordamos que las variables locales se almacenan en la pila. Es un puntero que apunta a la memoria que contiene las variables de entorno del programa.

Dirección	Descripción	Parte de la memoria virtual	Explicación
`0x7ffd545ba2e8 (la misma en ambas)`	Dirección de las **variables de entorno**.	**Pila (stack)**	Esta es la dirección en la que están almacenadas las variables de entorno, que es la misma que `environ` y `envp`. Es una dirección en la pila del proceso.
`0x5cb3959cd518`	Dirección de la variable **`environ`** en la memoria global.	**Segmento de datos (globales)**	Es la dirección en la que se almacena el puntero `environ`. Es una variable global, por lo tanto está en el segmento de datos globales.
`0x7ffd544b9b10`	Dirección del puntero `**arg3 (envp)**` que apunta a las variables de entorno.	**Pila (stack)**	Esta es la dirección donde se almacena el puntero `arg3` (también conocido como `envp[]`), que apunta a las variables de entorno. Está en la pila del proceso.

En caso de un argumento desconocido: Si el argumento pasado no coincide con ninguno de los anteriores, la función imprime un mensaje indicando el uso correcto de la función, que es: `Uso: environ [-environ|-addr]`.

37-fork:

1. **Llamada a `fork()`**:
    - La función `fork()` crea un nuevo proceso hijo que es una copia casi exacta del proceso padre. Retorna un valor diferente dependiendo de si está en el proceso padre o en el hijo.
        - **En el proceso hijo**: `fork()` devuelve **0**.
        - **En el proceso padre**: `fork()` devuelve el **PID del proceso hijo**.
        - **Si hay un error en `fork()`**: devuelve **-1**.
2. **Caso en el que `pid == 0`** (Proceso hijo):
    - Si `fork()` devuelve 0, significa que el proceso actual es el **hijo**.
    - Se imprime un mensaje que indica que se está ejecutando el proceso hijo, junto con su **PID** (`getpid()` devuelve el PID del proceso actual).
    - Luego, el proceso hijo termina inmediatamente mediante la llamada a `exit(0)`, lo que indica que el hijo terminó su ejecución de manera normal con un código de salida **0**.
3. **Caso en el que `pid > 0`** (Proceso padre):
    - Si `fork()` devuelve un valor mayor que 0, significa que el proceso actual es el **padre**.
    - Se imprime un mensaje que indica que el proceso padre está esperando a que termine el hijo, y se muestra el **PID** del proceso hijo.
    - Luego, el proceso padre espera a que el hijo termine con la llamada a `waitpid(pid, &status, 0)`, que bloquea la ejecución del padre hasta que el hijo haya terminado. El parámetro `status` captura el estado de salida del hijo.
        - Si el hijo termina de forma normal (usando `exit()` o al finalizar la función principal), el padre usa la macro `WIFEXITED(status)` para verificar que el hijo terminó correctamente. En ese caso, se imprime el código de salida del hijo, que se obtiene con `WEXITSTATUS(status)`.
        - Si el hijo termina de manera anormal (por ejemplo, si termina debido a una señal), se imprime un mensaje indicando que el hijo terminó de manera anormal.
4. **Caso de error en `fork()`**:
    - Si `fork()` devuelve -1, significa que hubo un error al intentar crear el proceso hijo. En este caso, se imprime un mensaje de error usando `perror("Error en fork")`.

¿PARA QUE SE USA TODO ESTO?

1-Ejecutar tareas en paralelo

2-Aislamiento de tareas

3-Simulación de multitarea

Si no ponemos pid no dará el actual (del hijo) y ppid el anterior (del padre)

- ## Gestión de directorios de búsqueda

38-search: search [-add|-del|-clear|-path]

La función `cmd_search` maneja una lista de directorios donde el shell busca archivos ejecutables, similar a cómo funciona la variable de entorno `PATH`. La función permite mostrar, modificar y gestionar esta lista de búsqueda a través de diferentes argumentos.

Explicación paso a paso:

1. **Mostrar la lista de búsqueda (sin argumentos):** Si no se pasan argumentos a la función (es decir, `tr[1]` es `NULL`), la función simplemente muestra los directorios que están en la lista de búsqueda. Esto se hace llamando a la función `mostrarListaBusqueda(&listaBusqueda)`, que imprime los directorios que han sido agregados a la lista.
2. **Agregar un directorio a la lista de búsqueda (`-add`):** Si el primer argumento es `-add` (es decir, `strcmp(tr[1], "-add") == 0`), la función verifica si se ha proporcionado un directorio (`tr[2]`). Si no se proporciona un directorio, muestra un mensaje indicando cómo se debe usar el comando. Si se proporciona un directorio, la función `agregarDirectorio(&listaBusqueda, tr[2])` agrega ese directorio a la lista de búsqueda.
3. **Eliminar un directorio de la lista de búsqueda (`-del`):** Si el primer argumento es `-del` (es decir, `strcmp(tr[1], "-del") == 0`), la función verifica si se ha proporcionado un directorio a eliminar (`tr[2]`). Si no se proporciona, muestra un mensaje de uso. Si se proporciona un directorio, la función `eliminarDirectorio(&listaBusqueda, tr[2])` intenta eliminar ese directorio de la lista. Si el directorio es encontrado y eliminado correctamente, muestra un mensaje de confirmación; si no se encuentra, muestra un mensaje indicando que el directorio no está en la lista.
4. **Limpiar la lista de búsqueda (`-clear`):** Si el primer argumento es `-clear` (es decir, `strcmp(tr[1], "-clear") == 0`), la función limpia toda la lista de búsqueda, eliminando todos los directorios de la lista. Esto se logra llamando a la función `limpiarListaBusqueda(&listaBusqueda)`, y luego se muestra un mensaje indicando que la lista ha sido limpiada.
5. **Importar los directorios del `PATH` a la lista de búsqueda (`-path`):** Si el primer argumento es `-path` (es decir, `strcmp(tr[1], "-path") == 0`), la función importa los directorios que están en la variable de entorno `PATH` a la lista de búsqueda. Esto se hace llamando a la función `importarPATH(&listaBusqueda)`, y luego muestra un mensaje confirmando que los directorios del `PATH` han sido importados.
6. **Argumentos no válidos:** Si el primer argumento no coincide con ninguno de los casos anteriores (`-add`, `-del`, `-clear`, `-path`), la función muestra un mensaje de uso, indicando cómo debe usarse el comando correctamente.

LA LISTA DE BÚSQUEDA ES MUY IMPORTANTE PARA LAS SIGUIENTES FUNCIONES PARA PODER USAR PROGSPEC. PAARA PODER USAR COMANDOS EXTERNOS COMO LS TENDREMOS QUE :

1-IMPORTAR EL PATH EN LA LISTA DE BÚSQUEDA

search -path:

2-Si la función es distinta que una de mi shell la buscará en los ejecutables del path. Como ls pertenece a una de las capretas del path que está en la lusta de búsqueda se podrá ejeuctar y además se debe ejecutar creando un proceso en primer plano como indica el enunciado.

3-Ahora ya podremos usarlo en las funciones y podremos hacer fg, fgpri etc con estes comandos y deberían funcionar según pide el enunciado

POR TANTO ESTOS COMANDOS SIGUIENTES QUE USAN PROGSPEC SOLO FUNCIONARÁN SI LOS COMANDOS EXTERNOS QUE LES PASAMOS ESTÁN EN LA LISTA DE BÚSQUEDA

Los comandos que usemos ahora estñan en /usr/local/bin -> para saber donde está cierto comando: which comando

Ejemplo:

-> which ls

/usr/bin/ls

IMPORTANTE:

-Instalar xterm. Esto abre una nueva terminal para De esta forma, podrás ejecutar comandos **en segundo plano** o con una **prioridad específica**, y **ver los resultados** en una ventana separada.

- **Gestión y ejecución de procesos**

Prioridades: -20 a 19

¿Para qué sirven las prioridades?

Las **prioridades** en sistemas operativos determinan el orden en que los **procesos** obtienen acceso al **tiempo de CPU** (la capacidad del procesador para ejecutar instrucciones). En un sistema con múltiples procesos en ejecución, la **prioridad** ayuda al **scheduler** (planificador de procesos) a decidir qué proceso debe ejecutarse a continuación.

¿Qué hacen las prioridades en `execpri`?

En el contexto de **`execpri`**, la prioridad **no cambia el comportamiento del comando directamente**, pero le indica al **sistema operativo** cómo manejar el proceso que se está ejecutando en cuanto al tiempo que recibe para ejecutarse.

Funcionamiento de las prioridades:

1. **Valores de Prioridad:**
    - **Prioridad alta** se representa con números **negativos** (ej. `-10`, `-5`).
    - **Prioridad baja** se representa con números **positivos** (ej. `5`, `10`).
    - **La prioridad más baja** sería `19` (el valor más alto que se puede asignar a la prioridad).
    - **La prioridad más alta** sería `-20` (el valor más bajo que se puede asignar a la prioridad).
2. **Efecto de la Prioridad en `execpri`:**
    - Cuando usas `execpri`, el sistema operativo ajusta el **nivel de prioridad** del proceso antes de ejecutarlo. Si el valor es **negativo**, el proceso tiene más **prioridad** y se ejecutará más rápido.
    - Si el valor es **positivo**, el proceso tendrá **menos prioridad** y el sistema le asignará menos tiempo de CPU, lo que puede hacer que se ejecute más lentamente si hay otros procesos con mayor prioridad.
3. **¿Por qué es importante la prioridad?**
    - Los procesos con **mayor prioridad** (números negativos) son **ejecutados antes** que los de menor prioridad cuando el sistema tiene que elegir entre varios procesos. Esto es crucial en sistemas con **múltiples procesos activos**, ya que **los procesos más importantes o críticos** se ejecutan primero.
    - Si un proceso tiene **prioridad baja**, puede que tenga que **esperar más tiempo** para obtener acceso al procesador, especialmente si el sistema está muy ocupado con procesos de alta prioridad.

Si intentamos usar valores muy negativos sin permisos de superusuario (root) NOS VA A DAR UN ERROR DE PERMISOS

Señales:

Señales estándar y su descripción (por orden)

1. **`SIGHUP` (1): Hangup (colgado de terminal)**
    - **Propósito:** Se envía cuando la terminal asociada al proceso se cierra o se desconecta.
    - **Uso típico:** Recargar configuraciones en procesos en ejecución o finalizar procesos cuando la terminal se desconecta.
2. **`SIGINT` (2): Interrupción**
    - **Propósito:** Se envía cuando el usuario presiona `Ctrl+C`.
    - **Uso típico:** Interrumpir procesos en ejecución.
3. **`SIGQUIT` (3): Salida forzada**
    - **Propósito:** Se envía con `Ctrl+\` y genera un volcado de memoria antes de finalizar.
    - **Uso típico:** Depuración avanzada para analizar el estado de un programa antes de finalizar.
4. **`SIGILL` (4): Instrucción ilegal**
    - **Propósito:** Indica que el proceso intentó ejecutar una instrucción de máquina inválida.
    - **Uso típico:** Normalmente generado por el sistema operativo cuando detecta errores graves en el programa.
5. **`SIGTRAP` (5): Trampa**
    - **Propósito:** Usado para depuración cuando un programa alcanza un punto de interrupción o trampa.
    - **Uso típico:** Herramientas de depuración.
6. **`SIGABRT` (6): Aborto**
    - **Propósito:** Se usa para terminar un programa de forma controlada cuando se detectan errores graves.
    - **Uso típico:** Llamado automáticamente por funciones como `abort()`.
7. **`SIGIOT` (6): Entrada/Salida del procesador**
    - Alias de `SIGABRT`. No es común en sistemas modernos.
8. **`SIGBUS` (7): Error de bus**
    - **Propósito:** Se envía cuando un proceso accede a una dirección de memoria inválida.
    - **Uso típico:** Manejar errores en la administración de memoria.
9. **`SIGFPE` (8): Error de coma flotante**
    - **Propósito:** Indica errores matemáticos como división entre cero o desbordamiento.
    - **Uso típico:** En programas que realizan cálculos matemáticos.
10. **`SIGKILL` (9): Matar**
    - **Propósito:** Termina un proceso de manera inmediata. No puede ser capturada ni ignorada.
    - **Uso típico:** Forzar la terminación de procesos que no responden.

11. **`SIGUSR1` (10): Usuario 1**
    - **Propósito:** Señal personalizada para acciones específicas definidas por el programa.
    - **Uso típico:** Realizar tareas personalizadas como reinicios o eventos definidos por el usuario.
12. **`SIGSEGV` (11): Violación de segmento**
    - **Propósito:** Indica acceso inválido a memoria.
    - **Uso típico:** Detectar errores en el manejo de punteros o memoria.
13. **`SIGUSR2` (12): Usuario 2**
    - **Propósito:** Señal personalizada para acciones específicas definidas por el programa.
    - **Uso típico:** Igual que `SIGUSR1`.
14. **`SIGPIPE` (13): Tubería rota**
    - **Propósito:** Se envía cuando un proceso escribe en una tubería sin lector.
    - **Uso típico:** Gestión de comunicación en pipes.
15. **`SIGALRM` (14): Alarma**
    - **Propósito:** Se envía cuando se cumple un temporizador configurado con `alarm()`.
    - **Uso típico:** Temporizadores o tareas programadas.
16. **`SIGTERM` (15): Terminación**
    - **Propósito:** Solicita la terminación del proceso de manera ordenada.
    - **Uso típico:** Cierre de procesos con limpieza de recursos.
17. **`SIGCHLD` (17): Hijo terminado**
    - **Propósito:** Notifica al proceso padre que un proceso hijo terminó o cambió de estado.
    - **Uso típico:** Gestión de procesos en segundo plano o hijos.
18. **`SIGCONT` (18): Continuar**
    - **Propósito:** Reanuda la ejecución de un proceso pausado.
    - **Uso típico:** Reanudar procesos detenidos.
19. **`SIGSTOP` (19): Pausar**
    - **Propósito:** Pausa un proceso sin posibilidad de captura.
    - **Uso típico:** Pausar procesos temporalmente.
20. **`SIGTSTP` (20): Pausar interactivo**
    - **Propósito:** Se envía cuando el usuario presiona `Ctrl+Z` para pausar un proceso.
    - **Uso típico:** Pausar procesos interactivos.
21. **`SIGTTIN` (21): Entrada en terminal**
    - **Propósito:** Se envía cuando un proceso en segundo plano intenta leer de la terminal.
    - **Uso típico:** Bloquear interacciones no permitidas.
22. **`SIGTTOU` (22): Salida en terminal**
    - **Propósito:** Se envía cuando un proceso en segundo plano intenta escribir en la terminal.
    - **Uso típico:** Igual que `SIGTTIN`.

23.**`SIGURG` (23): Urgente**

- **Propósito:** Indica un evento fuera de banda en un socket.
- **Uso típico:** Gestión de eventos de red.

24.**`SIGXCPU` (24): Límite de CPU**

- **Propósito:** Se envía cuando un proceso supera el límite de tiempo de CPU.
- **Uso típico:** Controlar procesos intensivos en uso de CPU.

25.**`SIGXFSZ` (25): Límite de archivo**

- **Propósito:** Se envía cuando un proceso intenta crear un archivo más grande que el límite permitido.
- **Uso típico:** Restringir tamaños de archivo.

26.**`SIGVTALRM` (26): Alarma virtual**

- **Propósito:** Temporizador basado en tiempo de CPU.
- **Uso típico:** Medir tiempo de ejecución virtual.

27.**`SIGPROF` (27): Perfilado**

- **Propósito:** Se envía cuando un temporizador de perfil se cumple.
- **Uso típico:** Herramientas de análisis de rendimiento.

28.**`SIGWINCH` (28): Cambio de ventana**

- **Propósito:** Se envía cuando cambia el tamaño de la ventana de terminal.
- **Uso típico:** Ajustar interfaces al tamaño de la terminal.

29.**`SIGIO` (29): Entrada/salida disponible**

- **Propósito:** Notifica que una operación de E/S está lista.
- **Uso típico:** Comunicación asincrónica.

30.**`SIGSYS` (31): Llamada al sistema inválida**

- **Propósito:** Se envía cuando un proceso realiza una llamada al sistema inválida.
- **Uso típico:** Depuración avanzada.

Número	Señal	Descripción
1	`SIGHUP`	Hangup detected on controlling terminal o cierre.
2	`SIGINT`	Interrupción desde el teclado (Ctrl + C).
3	`SIGQUIT`	Salida desde el teclado (Ctrl +).
4	`SIGILL`	Instrucción ilegal.
5	`SIGTRAP`	Trampa de depuración.
6	`SIGABRT`	Abortado por `abort()` (parada anormal).
7	`SIGBUS`	Error de bus (acceso no válido a memoria).
8	`SIGFPE`	Error de coma flotante (división por cero, etc.).
9	`SIGKILL`	Terminar el proceso (no puede ser ignorada).
10	`SIGUSR1`	Señal definida por el usuario 1.
11	`SIGSEGV`	Violación de segmentación (acceso inválido).
12	`SIGUSR2`	Señal definida por el usuario 2.
13	`SIGPIPE`	Tubo roto (escribir a un pipe sin lector).
14	`SIGALRM`	Temporizador (alarma).
15	`SIGTERM`	Terminación del proceso.
17	`SIGCHLD`	Cambios en el estado de un hijo.
18	`SIGCONT`	Continuar un proceso detenido.
19	`SIGSTOP`	Detener un proceso (no puede ser ignorada).
20	`SIGTSTP`	Detener desde el teclado (Ctrl + Z).
21	`SIGTTIN`	Proceso en segundo plano intenta leer del terminal.
22	`SIGTTOU`	Proceso en segundo plano intenta escribir al terminal.

Puedes obtener una lista completa de señales y sus números ejecutando el comando `kill -l` en un terminal Unix/Linux.

Diferencias entre primer y segundo plano

Característica	Primer Plano (fg)	Segundo Plano (back)
Control del shell	Bloquea el shell mientras se ejecuta.	El shell sigue disponible.
Ejecución del programa	Se ejecuta en el proceso hijo.	Se ejecuta en el proceso hijo.
Gestión de procesos	El shell espera a que termine.	Se registra en una lista (bgList).
Priorización (fgpri/backpri)	Ajusta la prioridad antes de ejecutarlo.	Ajusta la prioridad antes de ejecutarlo.
Interacción con el usuario	Visible e interactivo en la terminal.	Sin interacción directa (salida puede ser redirigida).
Estado del proceso	Se ejecuta en **estado de primer plano**.	Se ejecuta en **estado suspendido o en segundo plano**.
Reanudación	No requiere reanudación; ejecuta directamente.	Puede ser traído al primer plano con `fg`.
Identificación en el shell	No aparece en la lista de trabajos (jobs).	Aparece en la lista de trabajos (jobs).
Finalización del programa	El shell queda bloqueado hasta que termine.	Puede seguir ejecutándose después de cerrar el shell (si se usa `disown` o herramientas como `nohup`).

Aspecto	Primer Plano	Segundo Plano
Control del shell	Bloquea el shell mientras se ejecuta.	El shell queda libre para otros comandos.
Ejecución	Toma la terminal activamente.	Se ejecuta en paralelo al shell.
Interacción	Permite interacción directa.	No hay interacción directa.
Gestión	No aparece en la lista de trabajos.	Se lista con el comando `jobs`.
Reanudación	No necesita reanudarse.	Puede traerse al primer plano con `fg`.

FUNCIONES AUXIALIARES PARA LOS SIGUIENTES COMANDOS:

-Ejecutable: busca en la lista de búsqueda archivos ejecutables

-Excepve: ejecuta un comando que bsi es un ejecutable con execv o execve, permitiendo cambiar el entorno y su prioridad

Función `Ejecutable`

Esta función busca un ejecutable en una lista de directorios, de manera similar a cómo el sistema operativo buscaría un ejecutable en el PATH (aunque en este caso usa una lista personalizada).

1. **Parámetros**:
    - `ListaBusqueda *lista`: Es un puntero a la lista de directorios donde se buscará el archivo ejecutable.
    - `char *s`: Es el nombre del archivo ejecutable que estamos buscando.
2. **Proceso**:
    - **Comprobación inicial**: Si `s` es `NULL` o la lista de búsqueda está vacía (es decir, `SearchListFirst(lista)` devuelve `NULL`), simplemente devuelve `s` tal cual (esto ocurre si no hay nada que buscar).
    - **Pathname absoluto**: Si `s` ya es un pathname absoluto (por ejemplo, empieza con `/` o con `./` o `../`), la función no necesita buscar más, y devuelve el pathname tal cual. Esto es porque el archivo ya está especificado con su ruta completa, por lo que no es necesario buscarlo en los directorios de la lista.
    - **Búsqueda en la lista**: Si el archivo no tiene un pathname absoluto, la función intenta construir el path completo para el ejecutable. Lo hace iterando por cada directorio en la lista de búsqueda (`lista`), y para cada directorio, construye una cadena que combine el directorio con el nombre del archivo ejecutable (`path = p + "/" + s`).
    - **Comprobación de existencia**: Luego utiliza `lstat()` para verificar si el archivo existe en el directorio correspondiente. Si el archivo existe, devuelve la ruta completa a ese ejecutable.
    - **Si no se encuentra**: Si no se encuentra el ejecutable en ninguno de los directorios, devuelve el nombre original del archivo (`s`), lo que significa que no se ha encontrado el ejecutable.
3. **Retorno**: La función devuelve la ruta completa del ejecutable si se encuentra en la lista de directorios. Si no se encuentra, devuelve el nombre original (sin buscar).

Función `Execpve`

Esta función se encarga de ejecutar un programa usando las funciones `execv()` o `execve()`. Además, permite cambiar el entorno de ejecución y la prioridad del proceso antes de ejecutarlo.

1. **Parámetros**:
    - `ListaBusqueda *lista`: Es una lista de directorios donde se buscará el ejecutable.
    - `char *tr[]`: Es un arreglo de cadenas que contiene el comando y sus argumentos. El primer elemento de `tr[]` debe ser el nombre del ejecutable, y los siguientes son los argumentos del ejecutable.
    - `char **NewEnv`: Es un arreglo de variables de entorno (si se quiere modificar el entorno del proceso antes de ejecutarlo). Si es `NULL`, se usará el entorno predeterminado.
    - `int *pprio`: Un puntero a un valor de prioridad. Si se pasa un valor, se utilizará para cambiar la prioridad del proceso antes de ejecutarlo. Si es `NULL`, no se cambia la prioridad.
2. **Proceso**:
    - **Comprobación de validez del ejecutable**: Primero, la función comprueba si el primer argumento de `tr[]` (el ejecutable) es `NULL`, o si no se encuentra el ejecutable en la lista de directorios usando la función `Ejecutable()`. Si no se encuentra el ejecutable, se establece un código de error (`errno = EFAULT`) y se devuelve `-1`.
    - **Cambio de prioridad (si se pasa el parámetro `pprio`)**: Si se ha pasado un valor de prioridad a través del parámetro `pprio`, se intenta cambiar la prioridad del proceso actual usando la función `setpriority()`. Si no se puede cambiar la prioridad (por ejemplo, si el proceso no tiene permisos), se muestra un mensaje de error y se devuelve `-1`.
    - **Ejecutar el programa**:
        - Si `NewEnv` es `NULL`, la función llama a `execv()`, que reemplaza el proceso actual con el programa que se especifica en el primer argumento de `tr[]`, utilizando el entorno de variables de entorno predeterminado.
        - Si `NewEnv` no es `NULL`, la función llama a `execve()`, que reemplaza el proceso actual con el programa y utiliza el nuevo entorno de variables de entorno especificado en `NewEnv`.
3. **`execv()` vs `execve()`**:
    - `execv()` es una llamada al sistema que reemplaza el proceso actual con el ejecutable especificado. Toma dos argumentos: la ruta del ejecutable y un arreglo de cadenas que contiene los argumentos del ejecutable.
    - `execve()` es similar a `execv()`, pero también toma un tercer argumento que permite especificar el entorno de variables de entorno con el que debe ejecutarse el

nuevo proceso. Si se pasa `NULL`, se usan las variables de entorno predeterminadas del sistema.

4. **Retorno**:
    - Si la llamada a `execv()` o `execve()` es exitosa, el proceso actual se reemplaza por el nuevo programa, y el control nunca regresa a la función `Execpve()`.
    - Si ocurre un error, la función devuelve `-1` y ajusta el valor de `errno` para reflejar el tipo de error.

39-exec: exec progspec

Ejecuta un programa sin crear un nuevo proceso. Por tanto reemplaza el proceso actual del shell con el programa especificado.

1. **Inicialización de variables**:
    - Se crea un arreglo `env_vars[100]` para almacenar las variables de entorno pasadas como argumentos.
    - Se inicializan otras variables como `args` (para almacenar los argumentos del programa a ejecutar) y `env_count` (para contar cuántas variables de entorno se pasan).
2. **Procesamiento de las variables de entorno**:
    - La función recorre los argumentos de entrada (`tr[]`), buscando aquellos que tienen el formato "VAR=valor", que indican que son variables de entorno.
    - Estos argumentos se almacenan en el arreglo `env_vars`, y el contador `env_count` se incrementa para llevar el seguimiento de cuántas variables de entorno se han encontrado.
3. **Obtención de los argumentos del programa**:
    - Después de procesar las variables de entorno, los argumentos restantes en `tr[]` (que no son variables de entorno) son los que se utilizarán para ejecutar el programa. Estos se asignan a `args`.
4. **Comprobación de la existencia del programa**:
    - Si `args[0]` es `NULL` (lo que indica que no se proporcionó un programa para ejecutar), la función muestra un error y termina.
5. **Creación de un nuevo entorno (si se pasaron variables de entorno)**:
    - Si se pasaron variables de entorno (es decir, `env_count > 0`), se crea un nuevo arreglo `new_environ` para almacenar las variables de entorno.
    - Se copia cada una de las variables de entorno desde `env_vars` a `new_environ`.
6. **Ejecutar el programa**:
    - La función intenta ejecutar el programa con la función `Execpve()`, pasando la lista de directorios de búsqueda (presumiblemente de donde se busca el ejecutable), los argumentos del programa y las variables de entorno (si se pasaron).
    - Si el programa se ejecuta correctamente, el proceso actual del shell se reemplaza con el nuevo programa, lo que significa que el shell ya no continuará ejecutándose.
7. **Manejo de errores**:
    - Si `Execpve` falla, se verifica el tipo de error. Si es un "No such file or directory" (`ENOENT`), se muestra un mensaje de error adecuado.
    - Si ocurre otro tipo de error, se muestra un mensaje de error general con el tipo de error (`strerror(errno)`).
8. **Liberar la memoria**:
    - Si se creó un nuevo entorno, se libera la memoria asignada para las variables de entorno (`new_environ`), para evitar fugas de memoria.

9. **Finalización**:
    - Al final de la función, el shell termina su ejecución cuando el programa especificado reemplaza el proceso actual.

¿Qué hace `exec()`?

- La función `exec()` reemplaza el **proceso actual** por **otro programa**. Esto significa que el código del programa actual se detiene, y el nuevo programa (que se le pasa como argumento a `exec()`) empieza a ejecutarse en el mismo proceso.
- **Importante**: El **PID** (identificador del proceso) **no cambia**. El proceso sigue siendo el mismo, pero el contenido de ese proceso (su código y datos) se reemplaza por el nuevo programa.

¿Cómo se hace esto?

Cuando se llama a `exec()`, el **núcleo del sistema operativo** se encarga de reemplazar el espacio de direcciones del proceso actual con el espacio de direcciones del nuevo programa. Esto incluye su código, datos y contexto de ejecución. Todo lo que estaba ejecutándose en el proceso anterior (como funciones en el código) desaparece, y el nuevo programa comienza a ejecutarse.

¿En qué parte del código se realiza el reemplazo?

El reemplazo ocurre inmediatamente después de la llamada a `exec()`. Si `exec()` tiene éxito, el programa que llama a `exec()` **ya no continúa** ejecutándose; en su lugar, el nuevo programa comienza su ejecución. Si `exec()` falla, el proceso sigue ejecutándose como antes, pero se devuelve un error.

Resumen:

- **`exec()` reemplaza el proceso actual** (con el mismo PID) por un nuevo programa.
- El proceso original **no continúa** después de la llamada a `exec()` si todo va bien.
- El nuevo programa **toma el control** del proceso y se ejecuta en su lugar.

40-execpri: execpir prio progspec

Hace lo mismo que exec pero con prioridad.

Explicación de la función `cmd_execpri`:

1. **Argumentos**: El primer argumento es la prioridad que se quiere establecer para el proceso, y el segundo es el nombre del programa a ejecutar.
2. **Ejecución**: Llama a `Execpve` con la prioridad y los argumentos del programa. `Execpve` ejecuta el programa de la misma manera que `execv` o `execve`, pero además de los argumentos, acepta una prioridad (`priority`), la cual es utilizada para ajustar la prioridad del proceso antes de ejecutar el programa.
3. **Sin variables de entorno**: No se crean ni se modifican variables de entorno en esta función, por lo que no es necesario manejar ni pasar un entorno nuevo, como en el caso de `cmd_exec`.

¸PORQUE EN EXEC USAMOS UN NUEVO ENTORNO Y EN EXECPRI NO?:

1. **Para `exec`**: Si se especifican variables de entorno en el comando, usa `new_environ` y pasa el entorno modificado a `execve`. Si no se especifican variables de entorno, usa `execv` sin `new_environ`.
2. **Para `execpri`**: Dado que se especifica un cambio de prioridad y no se menciona la necesidad de modificar el entorno, puedes **prescindir de `new_environ`** en la mayoría de los casos, a menos que se incluyan variables de entorno explícitas.

Entonces, **no es necesario usar `new_environ` en ambos casos a menos que haya una razón explícita para modificar el entorno**, como lo indican las variables de entorno en el comando de ejecución (por ejemplo, cuando el comando incluye `VAR1 VAR2 VAR3 ...`).

41-fg:

Paso a paso de lo que hace `cmd_fg`:

1. **Comprobar argumentos:**
    - Verifica que el usuario ha introducido un programa a ejecutar.
    - Si no hay argumentos, muestra un mensaje de uso incorrecto.
2. **Crear un proceso hijo:**
    - Usa `fork()` para dividir el proceso en dos:
        - **Proceso hijo (`pid == 0`)**: Ejecuta el programa especificado con `Execpve` mostrando el nuevo pid`.`
        - **Proceso padre (`pid > 0`)**: Espera al proceso hijo con `waitpid`.
3. **Ejecutar el programa:**
    - El proceso hijo utiliza `Execpve` para buscar el programa en los directorios de la lista de búsqueda (`listaBusqueda`) y lo ejecuta.
    - Si no encuentra el programa, muestra un mensaje de error.
4. **Esperar al proceso hijo:**
    - El proceso padre espera a que el hijo termine.
    - Una vez que el proceso hijo termina, el padre evalúa cómo finalizó el proceso:
        - Si terminó por una señal (como `SIGTERM` o `SIGKILL`), muestra el nombre de la señal usando `NombreSenal()`.
        - Si terminó normalmente, muestra el código de salida.

42-fgpri:

Esta función permite ejecutar un programa en primer plano (el programa bloquea la terminal hasta su terminación) asignándole una prioridad específica.

- **Parámetros**: Toma dos argumentos: el primero es la prioridad que se asignará al proceso y el segundo es el programa que se ejecutará.
- **Proceso**: Si no se proporcionan los argumentos correctamente, se muestra un mensaje de uso. Luego, la prioridad es convertida de texto a número entero usando `atoi()`. Se crea un proceso hijo con `fork()`, y si es el hijo, el proceso ejecuta el programa con la prioridad dada utilizando la función `Execpve`. Si la ejecución falla, muestra un error y termina el proceso. Si el proceso es el padre, espera a que el hijo termine, mostrando el estado de la terminación (ya sea por señal o por código de salida).

43-back

44-backpri:

DIFERENCIAS ENTRE L CREACIÓN DE UN PROCESO EN PRIMER O EN SEGUNDO PLANO:

Diferencias principales:

1. **Espera del proceso padre**:
    - **Primer plano (fg)**: El padre espera a que el hijo termine antes de continuar (usando `waitpid`).
    - **Segundo plano (back)**: El padre no espera al hijo. El hijo sigue ejecutándose mientras el padre puede hacer otras cosas -> NO HAY WAITPID
2. **Grupo de procesos**:
    - **Primer plano (fg)**: El proceso hijo pertenece al mismo grupo de procesos que el padre, y su ejecución está vinculada al terminal.
    - **Segundo plano (back)**: El proceso hijo se coloca en un **nuevo grupo de procesos** con `setpgid(0, 0)` para que sea independiente del terminal y el shell, permitiéndole ejecutarse en segundo plano.
3. **Manejo de procesos**:
    - **Primer plano (fg)**: El proceso padre tiene que manejar el resultado de la ejecución del hijo (si terminó con error, señal, etc.).
    - **Segundo plano (back)**: El proceso padre no maneja el resultado del hijo en tiempo real, pero puede registrar el proceso en una lista de procesos en segundo plano.

Conclusión:

- **Primer plano**: El padre espera a que el hijo termine (bloquea su ejecución).
- **Segundo plano**: El padre no espera al hijo, y el hijo se ejecuta independientemente del padre.

45-listjobs:

Muestra la lista de procesos en segudno plano

1-Primero actualiza el estado de los procesos para ver como están actualemente -> update_background_processes()

2-Si esta vacía muestra un mensaje de que está vacía

3-Si NO está vacía muestra la lista con los procesos (procesos ejecutados con back y backpri)

46-deljobs: deljobs [-term | -sig]

Elimina procesos de la lista

-term: elimina los procesos terminados de forma normal de la lista

-sig: elimina los procesos terminados por señal de la listaBusqueda

FUNCIÓN AUXILIAR QUE ACTUALIZA LOS PROCESOS:

`update_background_processes`:
Esta función actualiza el estado de los procesos en segundo plano. Recorre la lista de procesos en segundo plano (`bgList`) y usa `waitpid()` con las opciones `WNOHANG` (sin bloquear) y `WUNTRACED` (para obtener el estado de los procesos detenidos). Según el estado que devuelva `waitpid()`, actualiza el estado del proceso: `FINISHED` si terminó normalmente, `SIGNALED` si terminó por una señal, `STOPPED` si está detenido, o `ACTIVE` si el proceso sigue en ejecución

FUNCIÓN AUXILIAR QUE LEE COMANDOS EXTERNOS AL SHELL:

RESUMEN FUNCIONES PREDEFINIDAS P3:

<stdio.h> // printf, perror, remove

<string.h> // strcmp, strdup

<stdlib.h> // malloc, free, exit, strtoul (string to unsigned long)**,** atoll (ASCII to long long)

getenv (const char *name): Recupera el valor de una variable de entorno especificada por su nombre (argumento `name`). Devuelve un puntero a una cadena de caracteres que contiene el valor de la variable de entorno si existe, o `NULL` si no se encuentra la variable.

putenv(char *string): Modifica una variable de entorno o la añade si no existe. La cadena `string` debe estar en el formato `"NOMBRE=VALOR"`. Si la variable ya existe, su valor se actualiza; si no, se crea una nueva variable de entorno.

<unistd.h> // chdir, getpid, getcwd, rmdir, execv, access, read y write, *seek

getuid(): devuelve el ID de usuario real (UID) del proceso que realiza la llamada

geteuid(): devuelve el ID de usuario efectivo (EUID), que es el UID que se usa para determinar los permisos de acceso

setuid(uid_t uid): establece el ID de usuario real (UID) del proceso a `uid`. Esto puede ser útil para cambiar el contexto de usuario en un proceso, por ejemplo, para elevar o reducir privilegios

seteuid(uid_t uid)**:**establece el ID de usuario efectivo (EUID) del proceso. Esta función es útil para cambiar los privilegios de un proceso durante su ejecución (por ejemplo, cambiar a un UID de superusuario temporalmente)

fork(): crea un nuevo proceso duplicando el proceso actual (un hijo)

execv(): La función `exec()` no es una función específica por sí sola, sino que se refiere a una familia de funciones bajo el nombre común de `exec`. Estas funciones se utilizan para reemplazar el proceso actual por un nuevo programa. Es decir, el proceso que llama a `exec` es reemplazado completamente por el nuevo programa especificado. Se utilizan comúnmente en programación de sistemas para ejecutar un nuevo programa dentro del proceso actual, sin crear un nuevo proceso.

execve(): La función `execv()` es parte de la familia de funciones `exec` y se utiliza para ejecutar un programa, reemplazando el proceso actual. `execv` toma dos argumentos:

- **Ruta del programa**: El primer argumento es la ruta completa del archivo ejecutable que deseas ejecutar.

- **Argumentos**: El segundo argumento es un array de cadenas de caracteres que representan los argumentos del programa a ejecutar (incluido el nombre del programa en la primera posición del array).

NO SON FUNCIONES, PERO SON VARIABLES DE ESTA LIBERÍA: **environ** y **envp**

<time.h> // time, ctime

<fcntl.h> // open, close, read

<sys/stat.h> // stat, lstat,

<errno.h> // errno, strerror

<sys/mman.h> //mmap, munmap, PROT_WRITE, PROT_READ, PROT_EXEC

<sys/shm.h> // shmget, shmat, shmdt shmctl

<sys/ipc.h> // se necesita también la librería anterior:

//IPC_RMID, IPC_CREAT, IPC_EXCL, SHM_RDONLY...

<pwd.h> // **struct passwd**

getpwuid(uid_t uid): recupera la información del usuario correspondiente a un ID de usuario (UID), devolviendo una estructura `passwd*` con detalles como el nombre de usuario y el directorio home)

getpwnam(const char *name)**:** devuelve un puntero a una estructura `passwd` que contiene la información del usuario cuyo nombre se pasa como argumento (`name`).

<sys/wait.h>

waitpid(): Permite al proceso padre esperar a que un hijo termine, especificando un proceso hijo particular o todos los hijos.

FLAGS:

- **`WIFEXITED(status)`**: Verifica si el hijo terminó normalmente.
- **`WEXITSTATUS(status)`**: Obtiene el código de salida del hijo si terminó normalmente.
- **`WIFSIGNALED(status)`**: Verifica si el hijo terminó por una señal.
- **`WTERMSIG(status)`**: Obtiene la señal que causó la terminación del hijo.
- **`WIFSTOPPED(status)`**: Verifica si el hijo fue detenido.

- **`WSTOPSIG(status)`**: Obtiene la señal que causó la detención del hijo.
- **`WIFCONTINUED(status)`**: Verifica si el hijo fue continuado.
- **`WNOHANG`**: Usado con `waitpid()` para no bloquearse.
- **`WUNTRACED`**: Usado con `waitpid()` para retornar también si el hijo fue detenido.
- **`WALL`**: Usado con `waitpid()` para esperar a todos los procesos hijos.

<sys/resource.h>

setpriority(): cambia la prioridad de un proceso. Esto permite modificar el valor de la prioridad de un proceso, de manera que se pueda gestionar el uso de los recursos del sistema. `0` si el cambio de prioridad fue exitoso, o `-1` si hubo un error (en cuyo caso `errno` se establecerá para describir el error).

CORRECCIONES DEFENSAS:

P1:

1-Operadores

Operador | (OR bit a bit)

El operador `|` realiza una operación OR bit a bit entre dos números. Compara los bits correspondientes de los dos números y devuelve 1 si al menos uno de los bits es 1, de lo contrario, devuelve 0.

Operador & (AND bit a bit)

El operador & realiza una operación AND bit a bit entre dos números. Compara los bits correspondientes de los dos números y devuelve 1 solo si ambos bits son 1; de lo contrario, devuelve 0.

Operador |= (OR bit a bit con asignación)

El operador `|=` es una versión compuesta de `|`, que **realiza la operación OR bit a bit** y luego **asigna el resultado** a la variable a la izquierda del operador.

Operador & = (AND bit a bit con asignación)

El operador &= es una versión compuesta de &, que **realiza la operación AND bit a bit** y luego **asigna el resultado** a la variable a la izquierda del operador.

2-Makefile: que hace?

1. **Compila el código fuente (`.c`) en archivos objeto (`.o`)**:
    - Cada archivo fuente `.c` se traduce a un archivo objeto `.o` mediante el compilador con la opción `-c`.
    - `-c`: Esta opción le dice al compilador que **solo compile el código fuente a un archivo objeto** y no intente enlazar nada todavía.
    - Resultado: Archivos `.o` que contienen código intermedio.
2. **Enlaza los archivos objeto (`.o`) para generar un ejecutable**:
    - Los archivos `.o` se combinan mediante el **enlazador** para producir el ejecutable final.
    - `-o`: Esta opción especifica el **nombre del archivo ejecutable** que se debe generar.
    - Resultado: Un programa ejecutable (por ejemplo, `p2`).
3. **Gestiona dependencias**:

    - Verifica qué archivos han cambiado (comparando fechas de modificación) y solo recompila los necesarios.
    - Esto acelera el proceso porque no recompila todo el proyecto cada vez.
4. **Define reglas para tareas adicionales**:
    - Por ejemplo, puedes añadir reglas para limpiar archivos (`clean`) o depurar el programa (`valgrind`).

NOTA: 8 /10

P2:

Todo bien -> NOTA: 10/10

P3:

Todo bien -> NOTA: 10/10