1.Responda con sus palabras las siguientes preguntas:

1. ¿Qué son las llamadas al sistema (system call)?

Son rutinas del Sistema Operativo que funcionan como interfaz entre los programas de usuario y el sistema operativo, que son utilizadas por los programas de usuarios para acceder al hardware.

1. ¿Cuál es su función principal en una computadora?

Su función es actuar como interfaz entre el sistema operativo y los programas de usuario para que los últimos puedan acceder al hardware del sistema. De esta manera, permiten que los programas interactúen de manera segura y controlada con el hardware y los recursos del sistema, sin que los usuarios o aplicaciones necesiten acceder directamente a estos (lo cual podría ser riesgoso). El sistema operativo actúa como intermediario, garantizando seguridad, estabilidad y multitarea

1. ¿Son accesibles al usuario? ¿Cómo es esta interacción?

Sí, pero no directamente. Los usuarios (o sus programas) acceden a las llamadas al sistema mediante APIs (interfaces de programación, como las bibliotecas estándar de C, por ejemplo open(), read(), fork()). Estas APIs abstraen la complejidad y ocultan los detalles de bajo nivel. Los usuarios finales no invocan llamadas al sistema manualmente, pero sus aplicaciones sí lo hacen constantemente "tras bambalinas".

2. Investigue qué es el estándar POSIX. ¿De dónde viene su nombre?

1. ¿Cuál es la motivación para su creación?

POSIX (Portable Operating System Interface) es un conjunto de estándares definidos por el IEEE (Institute of Electrical and Electronics Engineers) para garantizar la compatibilidad entre sistemas operativos, especialmente aquellos basados en Unix. Su objetivo es que el software desarrollado para un sistema POSIX pueda ejecutarse en otro con mínimas modificaciones.

El nombre POSIX fue propuesto por Richard Stallman (fundador del proyecto GNU) a solicitud del IEEE. Es un acrónimo de:

Portable

Operating

System

Interface

X (en referencia a Unix).

1. ¿Cuál es el sistema operativo utilizado como base? ¿Por qué?

El sistema operativo utilizado como base para POSIX es **UNIX**, específicamente la variante **UNIX System V** junto con características de **BSD (Berkeley Software Distribution)**. Porque POSIX fue desarrollado por el IEEE (Institute of Electrical and Electronics Engineers) a finales de los 80 para garantizar portabilidad de aplicaciones entre sistemas operativos tipo UNIX. En esa época, había muchas variantes de UNIX con diferencias significativas, lo que dificultaba el desarrollo de software compatible entre ellas. POSIX estandariza las interfaces del sistema operativo (como llamadas al sistema, librerías, shells, etc.), basándose en las implementaciones más comunes de UNIX.

1. Enumere las partes que componen el estándar.

El estándar POSIX se divide en varias partes o secciones. Las principales son:

* POSIX.1 – Interfaces de programación (API), como llamadas al sistema, funciones de librería estándar en C (por ejemplo, fork(), exec(), read(), write()).
* POSIX.2 – Interfaz de usuario, comandos y utilidades de shell (por ejemplo, ls, cp, grep).
* POSIX.1b – Extensiones para tiempo real (señales en tiempo real, prioridades).
* POSIX.1c – Hilos (pthreads).
* POSIX.1d, 1j, 1q – Más extensiones para sistemas en tiempo real y multiprocesamiento.
* POSIX.1-2001 (o POSIX.1003.1-2001) – Consolidación de POSIX.1, POSIX.2, y otras extensiones.
* POSIX.1-2008 – Revisión más moderna, integrando y eliminando partes obsoletas.

1. Encuentre el listado de las llamadas al sistema en Linux.

Listado de system calls del Linux: <https://man7.org/linux/man-pages/man2/syscalls.2.html>

3. Investigue acerca del comando strace de Linux.

1. ¿Para qué sirve éste comando? ¿De dónde viene su nombre?

El comando strace en Linux es una herramienta de diagnóstico y depuración que se utiliza para monitorear las llamadas al sistema (syscalls) y las señales que un proceso utiliza durante su ejecución.

El nombre strace proviene de la combinación de dos palabras en inglés:

S → System (Sistema, refiriéndose a system calls o llamadas al sistema).

trace → Rastrear (seguir la ejecución paso a paso).

Por lo tanto, strace significa literalmente "rastreador de llamadas al sistema" (System Call Tracer).

1. ¿Cuál es la sintaxis del comando?

La sintaxis es la siguiente: strace [OPCIONES] <COMANDO> [ARGUMENTOS\_DEL\_COMANDO]

1. Detalle las opciones -a, -e, -o, -p, -s.

-a (Alinear columnas). Función: Alinea la columna de los valores retornados (=) para mejorar la legibilidad.

-e (Filtrar syscalls o eventos). Función: Rastrea solo ciertas syscalls, señales o eventos específicos.

-o (Guardar salida en un archivo). Función: Redirige la salida de strace a un archivo (útil para análisis posteriores).

-p (Adjuntarse a un proceso en ejecución). Función: Rastrear un proceso ya en ejecución usando su PID.

-s (Mostrar más contenido en strings). Función: Controla el número máximo de caracteres mostrados en argumentos de strings (evita truncamiento).

4. Investigue acerca del comando ps de Linux.

1. ¿Para qué sirve éste comando? ¿De dónde viene su nombre?

El comando ps (Process Status) muestra información sobre los procesos en ejecución en el sistema en el momento que se ejecutó el comando, incluyendo:

* PID (ID del proceso).
* Usuario propietario.
* Consumo de CPU y memoria.
* Estado del proceso (ej: running, sleeping, zombie).
* Comando que lo inició.

Origen del nombre: Viene de "Process Status" (Estado de Procesos), heredado de Unix antiguo. En sistemas modernos, ps obtiene datos del sistema de archivos virtual /proc.

1. ¿Cuál es la sintaxis del comando?

Su sintaxis es: ps [OPCIONES]

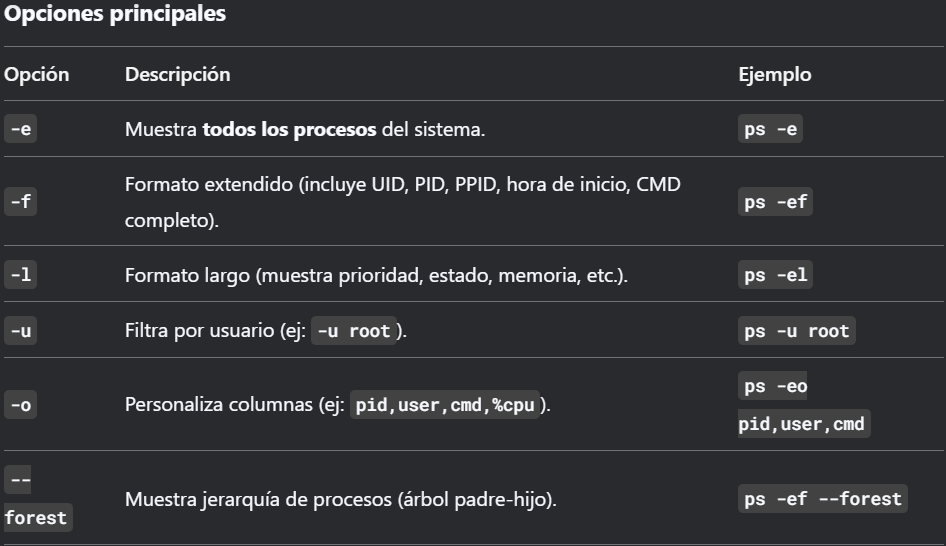
1. ¿Cuáles son los dos estilos de sintaxis de éste comando? Detalle el uso de ambos y sus distintas opciones.

ps admite dos estilos de opciones (incompatibles entre sí):

* Estilo UNIX (con guiones -), Más común en sistemas Linux modernos.

Opciones tradicionales (ej: -e, -f).

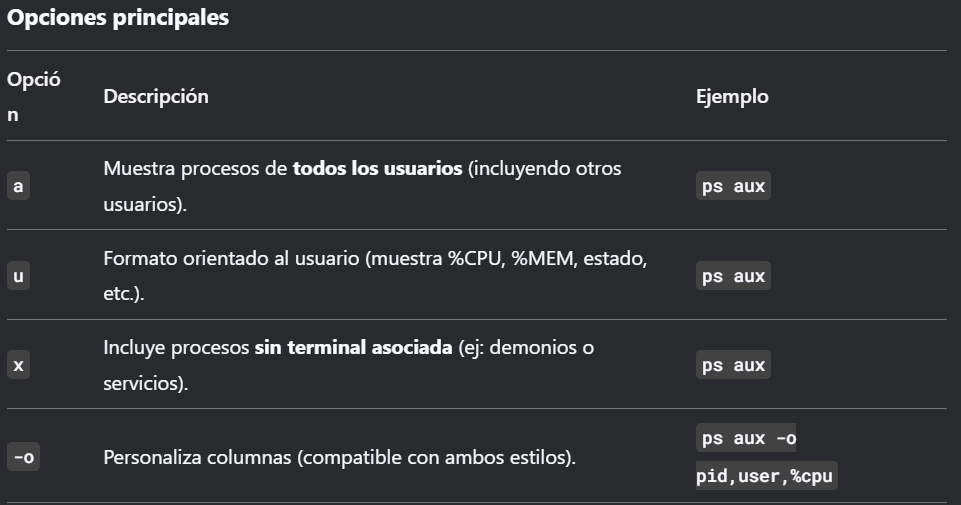
Ejemplo: ps -ef.



* Estilo BSD (sin guiones):

Agrupa opciones en bloques de letras (ej: aux). Muestra más detalles de recursos (%CPU, %MEM) por defecto.

Ejemplo: ps aux.



5. Para el lenguaje C, investigue qué archivos de cabecera contienen funciones y enumeraciones para

llamadas al sistema.

En el lenguaje C, las llamadas al sistema (system calls) y sus constantes asociadas se definen en archivos de cabecera (headers) específicos, ubicados generalmente en /usr/include/. A continuación, detallo los archivos más relevantes:



1. Describa las dos maneras que ofrece el lenguaje C para realizar llamadas al sistema.

1. Llamadas Directas (Funciones de Biblioteca GNU C / Glibc)

Mecanismo: Se utilizan funciones estándar de la biblioteca GNU C (glibc), que actúan como wrappers (envoltorios) alrededor de las llamadas al sistema reales. Estas funciones simplifican el uso, manejan errores y proporcionan una interfaz más amigable.

Características:

* Portables: Funcionan en cualquier sistema compatible con POSIX.
* Más seguras: Validan parámetros y manejan errores.
* Sintaxis familiar: Similar a otras funciones en C.

2. Llamadas Indirectas (Usando syscall())

Mecanismo: Se utiliza la función syscall() (definida en <sys/syscall.h>) para invocar directamente la llamada al sistema por su número único. Requiere conocer el número de la syscall (definido en <sys/syscall.h> como SYS\_xxx).

Características:

* Menos portable: Los números de syscall varían entre arquitecturas (x86, ARM, etc.).
* Más flexible: Permite acceder a syscalls no expuestas por glibc.
* Sin validaciones: No maneja errores automáticamente (similar a llamadas en ensamblador).

1. Busque información acerca de las siguientes llamadas al sistema: getcwd, chdir y time. Describa los parámetros de las mismas y qué valores retorna.

* getcwd - Obtener Directorio de Trabajo Actual. Obtiene la ruta absoluta del directorio de trabajo actual (current working directory).

Sintaxis: char \*getcwd(char \*buf, size\_t size);

Parámetros:

* buf: Buffer donde se almacenará la ruta (si es NULL, la función reserva memoria automáticamente).
* size: Tamaño del buffer (en bytes).

Valor de Retorno:

* Éxito: Retorna un puntero a la ruta (buf si se proporcionó, o un nuevo puntero si buf = NULL).
* Error: Retorna NULL y setea errno (ej: si el buffer es demasiado pequeño).
* chdir - Cambiar Directorio de Trabajo. Cambia el directorio de trabajo actual del proceso.

Sintaxis: int chdir(const char \*path);

Parámetros:

* path: Ruta del directorio destino (relativa o absoluta).

Valor de Retorno:

* Éxito: 0.
* Error: -1 y setea errno (ej: si el directorio no existe).
* time - Obtener Tiempo en Segundos desde el Epoch. Retorna el tiempo actual en segundos desde el Epoch (00:00:00 UTC, 1 de enero de 1970).

Sintaxis: time\_t time(time\_t \*tloc);

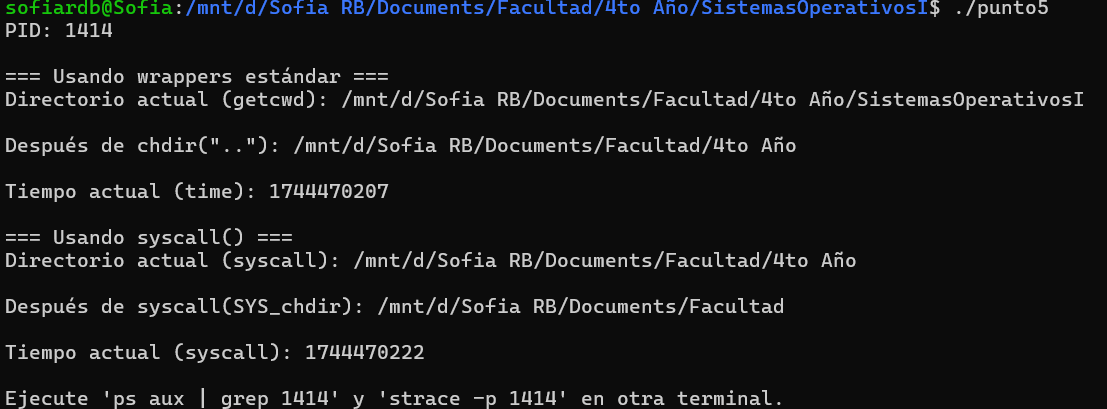
Parámetros:

* tloc: Si no es NULL, el valor se almacena también en \*tloc.

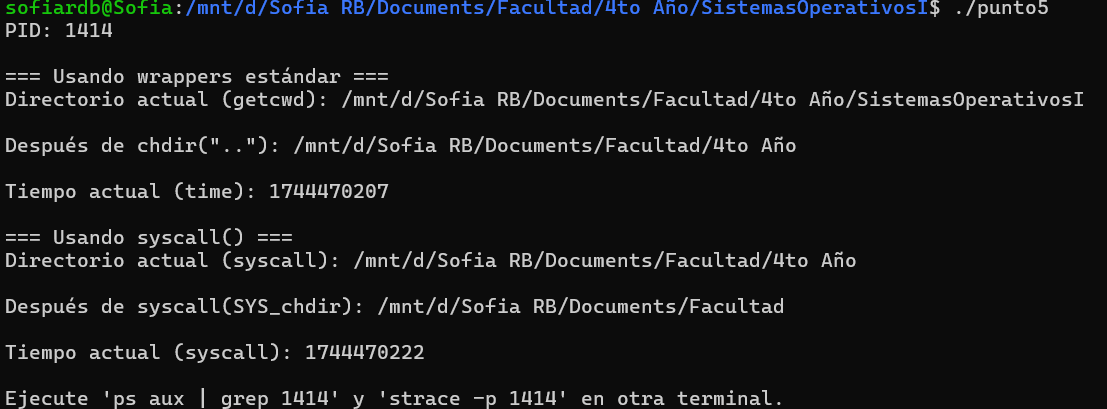
Valor de Retorno:

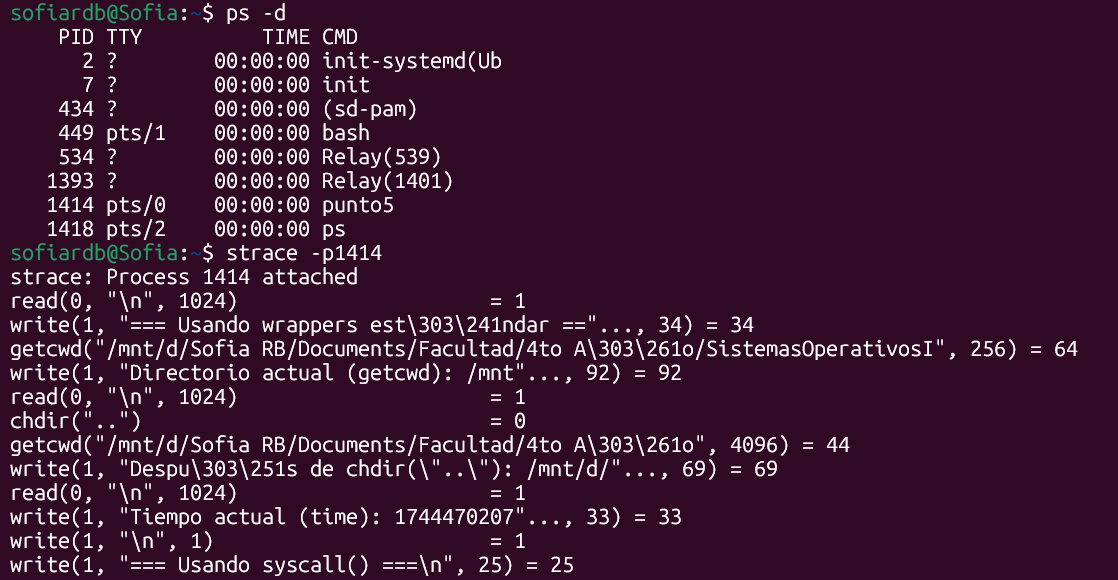
* Éxito: Segundos desde el Epoch.
* Error: (time\_t)-1 (raro, ocurre si tloc apunta a una dirección inválida).

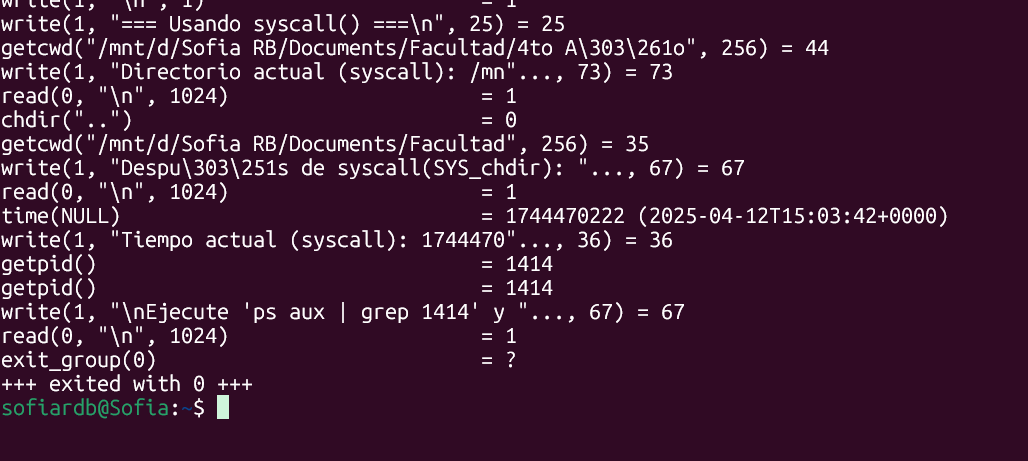
1. Escriba, compile y ejecute un programa en C que invoque estas llamadas al sistema, combinando si es posible, las dos maneras de invocar. Incluya en el código sendas presentaciones por pantallas de los resultados obtenidos.



1. Utilizando los comandos ps y strace ejecute el programa del apartado anterior para verificar los valores obtenidos y visualizar paso a paso las llamadas al sistema. Ayuda: incluya pausas en el código.







1. Describa el significado de los valores retornados por todas las llamadas al sistema observadas.

La llamada read(0, "\n", 1024) = 1. Función: Lee entrada desde el descriptor de archivo 0 (entrada estándar, normalmente el teclado).

Parámetros:

* 0: Descriptor de archivo para stdin.
* "\n": Buffer donde se almacena lo leído (en este caso, un salto de línea).
* 1024: Número máximo de bytes a leer.

Valor retornado (=1). Indica que se leyó 1 byte (el carácter \n).

La llamada write(1, "=== Usando wrappers est\303\241ndar =="..., 34) = 34. Función: Escribe datos en el descriptor de archivo 1 (salida estándar, normalmente la terminal).

Parámetros:

* 1: Descriptor de archivo para stdout.
* "=== Usando wrappers estándar =="...: Cadena a escribir (34 bytes).
* 34: Número de bytes a escribir.

Valor retornado (=34). Confirma que se escribieron 34 bytes (todos los solicitados).

Si hubiera error, retornaría -1.

La llamada getcwd("/mnt/d/Sofia RB/Documents/Facultad/4to A\303\261o/SistemasOperativosI", 256) = 64. Función: Obtiene el directorio de trabajo actual (Current Working Directory).

Parámetros:

* Buffer: "/mnt/d/Sofia RB/Documents/..." (ruta almacenada aquí).
* 256: Tamaño máximo del buffer.
* Valor retornado (=64): Indica que la ruta ocupa 64 bytes.

Si el buffer fuera insuficiente, retornaría NULL.

La llamada chdir("..") es una llamada al sistema (system call) que cambia el directorio de trabajo actual al directorio padre (parent directory) del directorio actual.

Valores retornados por chdir()

La llamada retorna un valor entero que indica éxito o error:

* 0 Éxito: El directorio fue cambiado correctamente.
* -1 Error: No se pudo cambiar el directorio (por ejemplo, si no tienes permisos o el directorio padre no existe).

La llamada time(NULL) = 1744470222 (2025-04-12T15:03:42+0000). Función: Obtiene el tiempo actual en segundos desde el epoch (1 de enero de 1970, 00:00:00 UTC).

Parámetro:

* NULL: Indica que no se almacenará el resultado en un buffer adicional.

Valor retornado (1744470222). Representa el instante exacto 12 de abril de 2025 a las 15:03:42 UTC.

Útil para timestamps, cálculos de intervalos, o registrar eventos.

La llamada getpid() = 1414. Función: Obtiene el ID del proceso actual (Process ID o PID).

Parámetros: Ninguno.

Valor retornado (1414):

* Identificador único asignado por el kernel al proceso en ejecución.
* Usado para gestionar procesos (ej: enviar señales con kill).

La llamada exit\_group(0). Función: Termina todos los hilos del proceso actual y retorna un código de estado.

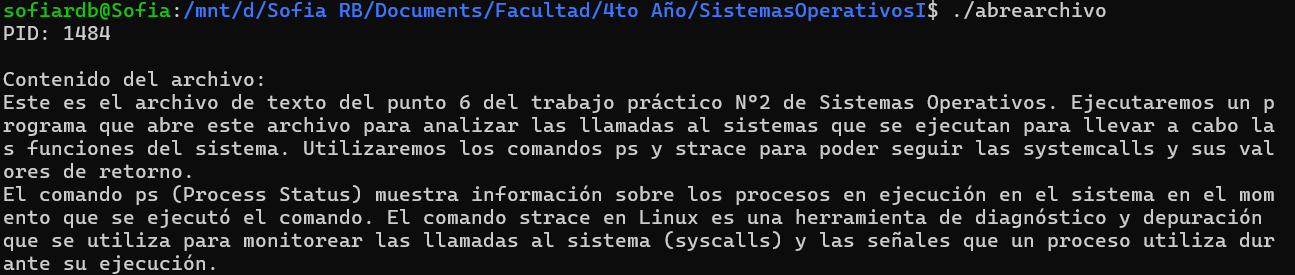
Parámetro (0):

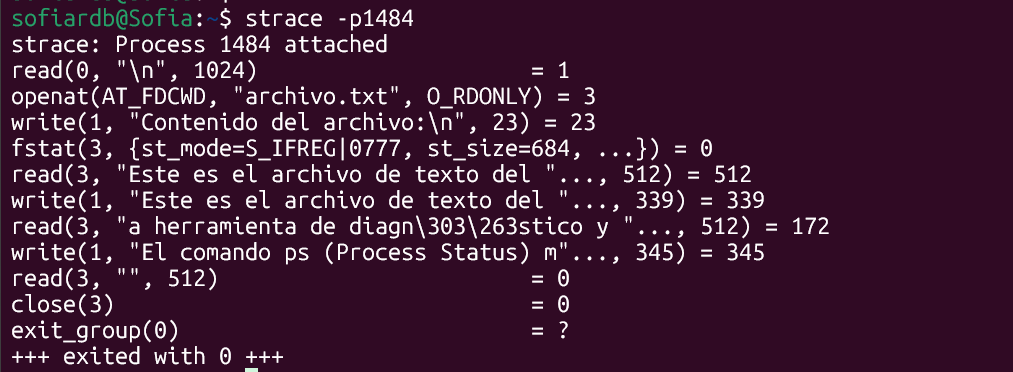
* 0: Indica éxito (convención estándar en Unix/Linux).
* Valores distintos de 0 señalan errores (ej: 1 para fallos genéricos).

6. Escriba, compile y ejecute un programa en lenguaje C que permita abrir un archivo de texto, el cual debe

contener al menos dos líneas.

1. Utilice nuevamente los comandos ps y strace para visualizar las llamadas al sistema invocadas.





1. Observe de qué manera realiza el sistema operativo la apertura y lectura del archivo, comparándola con la forma que lo hace con su código de C. ¿Qué diferencias encuentra?

|  |  |  |  |
| --- | --- | --- | --- |
| Operación | Sistema Operativo (strace) | C (Librería Estándar) | Diferencias Clave |
| Apertura | openat(AT\_FDCWD, "archive.txt", O\_RDONLY) = 3 | fopen("archive.txt", "r") | - openat retorna un descriptor crudo (3).  - fopen retorna un FILE\* con buffering. |
| Lectura | read(3, "Black hole...", 4096) | fgets(linea, siezeof(linea), archivo) | - read lee bytes exactos (sin formato).  - fgets lee líneas (hasta \n o EOF). |
| Escritura | write(1, "Black hole...", 176) | printf("%s", linea) | - write escribe bytes crudos.  - printf formatea texto (ej: %s, \n). |
| Cierre | close(3) | fclose(archivo) | - close libera el descriptor.  - fclose también libera buffers internos. |

1. Describa todas las llamadas al sistema invocadas por el programa y sus valores de retorno.

|  |  |  |  |
| --- | --- | --- | --- |
| Llamada | Parámetros | Retorno | Significado |
| read(0, "\n", 1024) | 0 (stdin), "\n", 1024 1 | 1 | Leyó 1 byte (salto de línea) de la entrada estándar. |
| openat(...) | AT\_FDCWD, "archive.txt", O\_RDONLY | 3 | Abrió el archivo y asignó el descriptor 3. |
| write(1, "Contenido..." | 1 (stdout), mensaje, 23 | 23 | Escribió 23 bytes en la salida estándar. |
| fstat(3, {...}) | Descriptor 3, struct stat | 0 | Obuvo metadata del archivo (éxito). Tamaño: 631 bytes. |
| read(3, "Black hole...") | Descriptor 3, buffer, 4096 | 631 | Leyó todo el contenido del archivo (631 bytes) en un solo bloque. |
| write(1, "Black hole...") | 1, línea 1 del archivo, 176 | 176 | Escribió la primera línea (176 bytes) en stdout. |
| write(1, "Always an angel...") | 1, línea 2, 172 | 172 | Escribió la segunda línea. |
| read(3, "", 4096) | Descriptor 3, buffer vacío | 0 | Fin del archivo (EOF). |
| close(3) | Descriptor 3 | 0 | Archivo cerrado correctamente. |
| exit\_group(0) | 0 (código de salida) | ? | Proceso terminado con éxito |

1. Respecto a la primera función del sistema operativo ¿qué conclusiones puede obtener?

openat es la syscall básica para abrir archivos en Linux.

Trabaja con descriptores (3) y flags (O\_RDONLY = solo lectura).

Cuando llamas a fopen("archivo.txt", "r"), internamente se ejecuta openat() para obtener el descriptor 3.

Luego, fopen envuelve este descriptor en una estructura FILE\* para manejar buffering.

Diferencias claves que se pueden identificar son los niveles de abstracción, ya que mientras C trabaja con FILE\* y funciones de alto nivel (fgets, printf), el SO usa descriptores (int) y llamadas al sistema (read, write). También está el buffering,  donde mientras fgets puede leer múltiples líneas en una sola llamada a read (buffering), read lee exactamente lo solicitado (ej: 4096 bytes en el strace).

Podemos concluir entonces que el código es legible y portable, gracias a las abstracciones de C y que el SO realiza operaciones crudas y eficientes, visibles con strace. Las funciones de C (fopen, fgets) simplifican el manejo de archivos, pero es útil entender las syscalls subyacentes para depurar o optimizar.

De esta forma, considerando que la primera función del sistema operativo es ser una interfaz entre el usuario y el hardware, podemos ver cómo simplifica el uso de system calls, simplifica esta tarea. A su vez, como las system calls, se hallan en la interfaz entre el usuario y el sistema operativo, por ejemplo el trabajo de leer un archivo de disco (que seria lo que hace read), se simplifica a usar 2 líneas de código (fopen() y fread()).

7. Escriba, compile y ejecute un programa en lenguaje C que permita abrir un archivo de texto, el cual debe

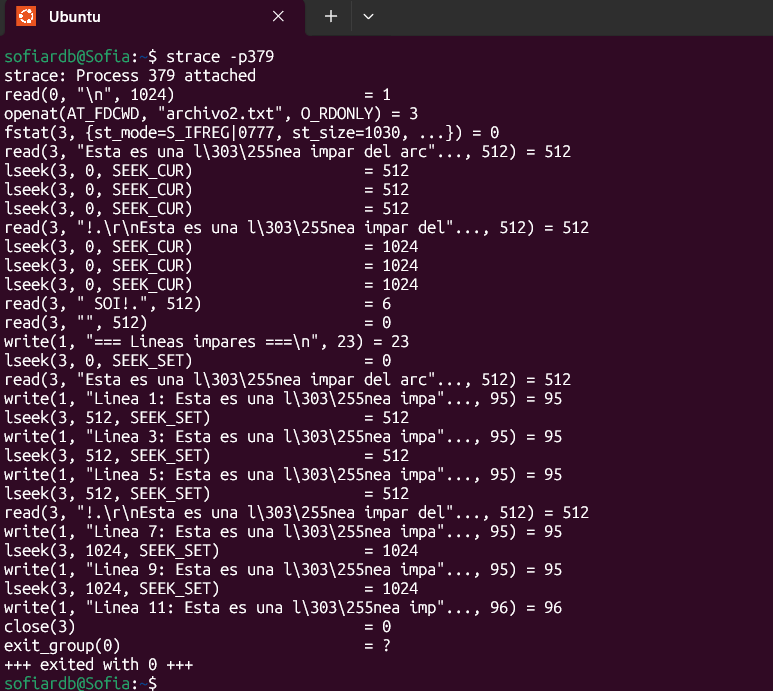
contener al menos diez líneas con la misma cantidad de caracteres cada una (al menos veinte).

1. Luego de abrir el archivo, lea del mismo y presente en pantalla las líneas impares, utilizando para ello

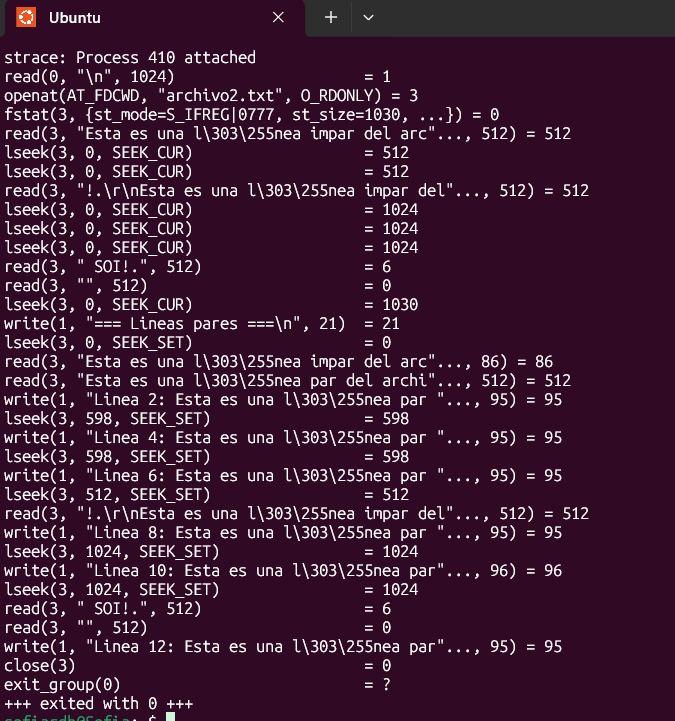
alguna función de C que le permita ubicar el puntero al archivo en la posición inicial de cada línea.

1. Repita el apartado anterior para las líneas pares.
2. Modifique el código para leer cinco caracteres, saltear diez y volver a leer cinco, y así sucesivamente.
3. Utilice nuevamente los comandos ps y strace para visualizar las llamadas al sistema invocadas.

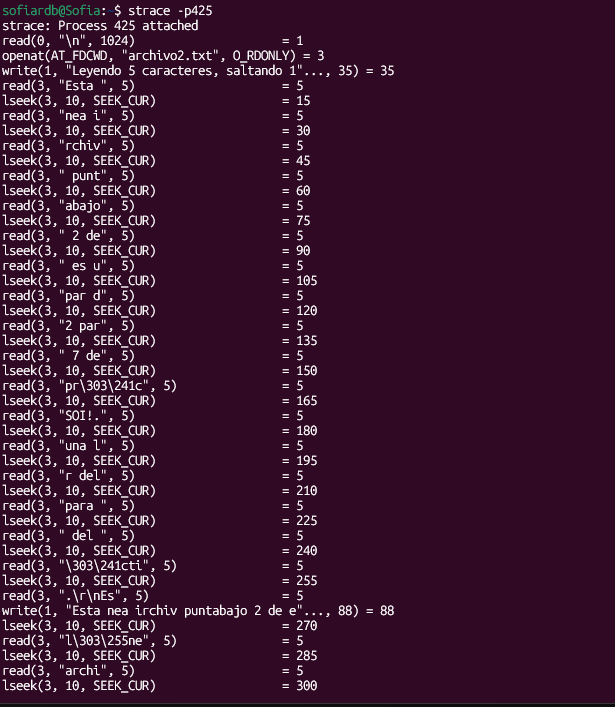
Leerimpares

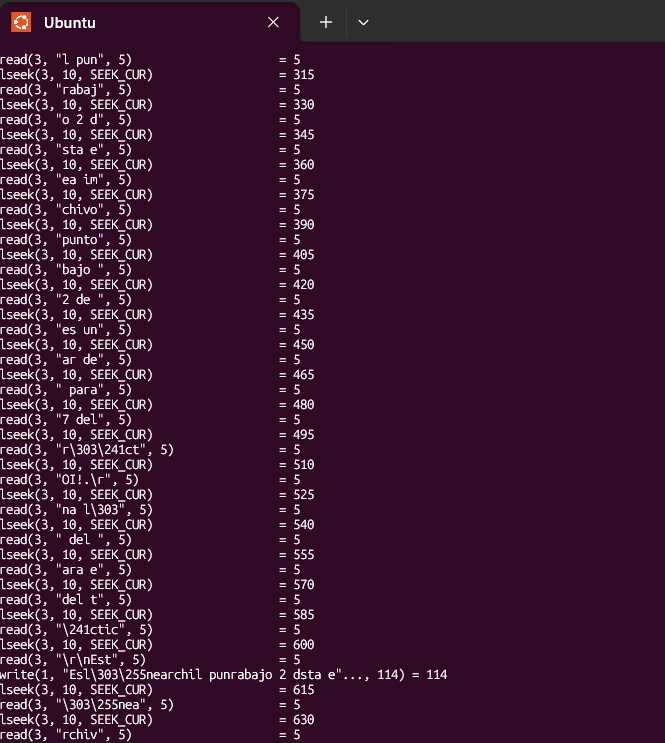


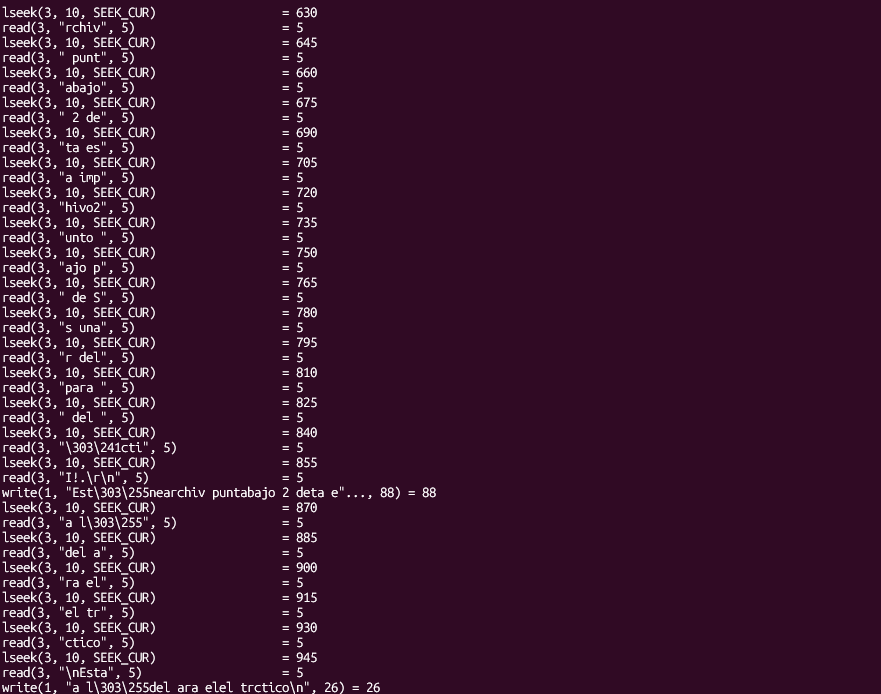
Leerpares

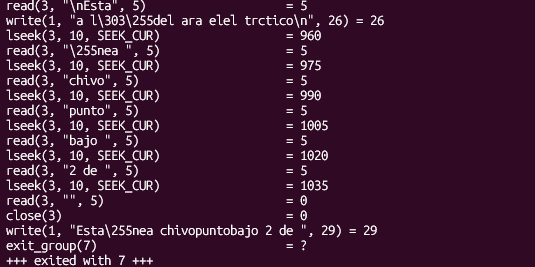


Salta10









1. Observe de qué manera realiza el sistema operativo el posicionamiento del puntero al archivo,

comparándola con la forma que lo hace con su código de C. ¿Qué diferencias encuentra?

En el código que lee líneas impares y pares, se utilizan funciones de alto nivel como fgets y fseek para navegar por el archivo. El sistema operativo, sin embargo, maneja estas operaciones a través de llamadas crudas como read y lseek. La principal diferencia radica en el buffering: mientras que fgets lee líneas completas y maneja buffers internos para optimizar la lectura secuencial, el SO trabaja directamente con bloques de bytes sin interpretar saltos de línea. Esto hace que las operaciones del SO sean menos intuitivas pero más precisas en términos de posicionamiento. Por ejemplo, cuando el código en C usa fseek para saltar a una línea específica, el SO traduce esto en un lseek que ajusta el puntero a un offset exacto en bytes, ignorando la estructura lógica del archivo.

En el código que lee 5 caracteres y salta 10, se evidencia claramente la diferencia entre el buffering de C y las operaciones crudas del SO. Mientras que el código en C intenta leer 5 caracteres y luego saltar 10, el SO puede leer bloques más grandes (como 4096 bytes) debido al buffering interno de fread. Esto provoca que las llamadas a lseek no se correspondan con los saltos esperados, ya que el puntero se mueve dentro del buffer en lugar del archivo físico. Para sincronizar el comportamiento del código con lo que muestra strace, es necesario usar descriptores crudos (read y lseek), lo que elimina el buffering y permite ver los saltos exactos en el archivo.

Las diferencias principales entre cómo C y el SO manejan las operaciones de archivo se centran en el nivel de abstracción. C utiliza funciones como fread y fseek que añaden capas de buffering y manejo de errores, lo que mejora la eficiencia en accesos secuenciales pero oscurece las operaciones reales del SO. En cambio, el SO opera con descriptores crudos (open, read, lseek), lo que proporciona un control preciso pero requiere más atención al manejar offsets y buffers. Por ejemplo, mientras fseek en C puede ajustar punteros dentro de un buffer interno, lseek en el SO mueve físicamente el puntero del archivo, lo que es más lento pero transparente en herramientas como strace.

1. Describa todas las llamadas al sistema invocadas por el programa y sus valores de retorno.

Programa leerimpares.c:

* read(0, "\n", 1024) = 1: Lee el Enter (\n) de getchar().
* openat(AT\_FDCWD, "caracteres.txt", O\_RDONLY) = 3: Abre el archivo. Le da el descriptor 3.
* fstat(3, {...}) = 0: Obtiene metadatos del archivo (tamaño: 209 bytes). Retorna 0 (éxito).
* read(3, "aaaaaaaaaaaaaaaaaaaa\n...", 4096) = 209: Lee todo el archivo (buffering). Retorna 209 (bytes leídos).
* lseek(3, 0, SEEK\_CUR) = 209 (repetido): Ajustes internos del buffer (no cambia la posición real).
* write(1, "=== Lineas impares ===\n", 23) = 23: Escribe encabezado en stdout.
* lseek(3, 0, SEEK\_SET) = 0: Reposiciona al inicio del archivo.
* read(3, "aaaaaaaaaaaaaaaaaaaa\n...", 4096) = 209: Re-lectura del archivo.
* write(1, "Linea 1: aaaaaaaaaaaaaaaaaaaa\n", 30) = 30: Imprime línea 1.
* lseek(3, 209, SEEK\_SET) = 209 (repetido): Saltos al final del archivo (para ignorar líneas pares).
* close(3) = 0: Cierra el descriptor.
* exit\_group(0) = ?: Termina el proceso.

Programa leerpares.c:

* read(0, "\n", 1024) = 1: Lee un carácter de getchar().
* openat(AT\_FDCWD, "caracteres.txt", O\_RDONLY) = 3: Abre archivo. Le da el descriptor 3.
* fstat(3, {...}) = 0: Traer información del archivo para luego poder leerlo. Devuelve 0 porque tuvo éxito
* read(3, "aaaaaaaaaaaaaaaaaaaa\n...", 4096) = 209: Lee 209 caracteres (las 10 lineas de 20 caracteres mas los 9 saltos de línea)
* lseek(3, 0, SEEK\_CUR) = 209 (repetido): Ajustes de buffer.
* write(1, "=== Lineas pares ===\n", 21) = 21: Escribe el encabezado.
* lseek(3, 0, SEEK\_SET) = 0: Reinicia puntero al inicio.
* read(3, "aaaaaaaaaaaaaaaaaaaa\n", 21) = 21: Lee línea 1 (pero la ignora).
* write(1, "Linea 2: bbbbbbbbbbbbbbbbbbbb\n", 30) = 30: Imprime línea 2.
* lseek(3, 209, SEEK\_SET) = 209 (repetido): Saltos al final para ignorar líneas impares.
* close(3) = 0: Cierra descriptor.
* exit\_group(0) = ?: Termina proceso.

Programa saltar10.c:

* read(0, "\n", 1024) = 1: Lee el Enter (\n) presionado para continuar (de getchar()). Retorna 1 (1 byte leído).
* openat(AT\_FDCWD, "caracteres.txt", O\_RDONLY) = 3: Abre el archivo caracteres.txt en modo lectura. Retorna descriptor 3 (éxito).
* write(1, "Leyendo 5 caracteres...", 35) = 35: Escribe el mensaje inicial en la consola (stdout).
* Retorna 35 (bytes escritos).
* read(3, "aaaaa", 5) = 5: Lee los primeros 5 caracteres del archivo ("aaaaa"). Retorna 5 (bytes leídos).
* lseek(3, 10, SEEK\_CUR) = 15: Salta 10 bytes desde la posición actual. Nueva posición: 15 (éxito).
* read(3, "aaaaa", 5) = 5: Lee los siguientes 5 caracteres. Retorna 5.
* lseek(3, 10, SEEK\_CUR) = 30: Salta otros 10 bytes. Nueva posición: 30.
* Patrón repetido hasta EOF: Ciclo de read(3, ..., 5) = 5 (lectura) y lseek(3, 10, SEEK\_CUR) = N (salto).
* read(3, "", 5) = 0: Fin del archivo (EOF). Retorna 0.
* close(3) = 0: Cierra el descriptor 3. Retorna 0 (éxito).
* exit\_group(0) = ?: Termina el proceso con código 0 (éxito).

8. Modifique cualquiera de los códigos realizados, cambiando el valor de retorno de la función main().

1. ¿Puede observar algún cambio en alguna de las llamadas al sistema?

La llamada exit\_group tiene como parámetro el nuevo valor de retorno y el mensaje de salida indica que se ha terminado la ejecución del proceso con el valor 3 de salida.

1. Modifique el código utilizando una función main() de tipo void, y terminando el código utilizando la

llamada al sistema exit. ¿Qué diferencia encuentra?

La llamada exit se realiza como si retornara el valor 0 como en el programa original.