**Pipex**

**Inicio**

El proyecto "pipex" es esencialmente una recreación de la funcionalidad de los pipes y redirecciones en un shell UNIX. Aquí está cómo enfocaría este proyecto:

**1. Comprensión del Concepto de Pipes en UNIX**

* Entender cómo funcionan los pipes (**|**) en UNIX, que permiten la salida de un comando ser la entrada del siguiente.
* Familiarizarse con la redirección de entrada y salida (**<**, **>**).

**2. Planificación y Diseño de Estructuras de Datos**

* Diseñar estructuras para manejar comandos y argumentos.
* Decidir cómo almacenar y transferir datos entre procesos.

**3. Implementación de Funcionalidades Básicas**

* Crear una función para parsear argumentos y separarlos en comandos.
* Implementar la funcionalidad para ejecutar dos comandos con un pipe entre ellos.

**4. Manejo de Archivos**

* Abrir **archivo1** para lectura y **archivo2** para escritura.
* Utilizar **dup** o **dup2** para redirigir la entrada/salida estándar a estos archivos.

**5. Uso de llamadas al sistema UNIX**

* Utilizar **fork**, **pipe**, **execve**, y otras llamadas al sistema requeridas para la creación de procesos y manejo de pipes.
* Gestionar cuidadosamente los errores en cada llamada al sistema.

**6. Gestión de Errores y Fugas de Memoria**

* Asegurar que el programa maneja errores de forma adecuada (por ejemplo, archivos inexistentes, comandos inválidos).
* Realizar pruebas para detectar y corregir fugas de memoria.

**7. Implementación de la Parte Extra (si aplica)**

* Una vez completada la parte obligatoria, agregar la funcionalidad para manejar múltiples pipes.
* Implementar la funcionalidad de **here\_doc** con **<<** y **>>** para manejar la entrada estándar y la redirección de salida.

**8. Pruebas Rigurosas**

* Probar el programa con diferentes comandos y escenarios para asegurarse de que se comporta como se espera.
* Comparar los resultados del programa con los de la ejecución directa en el shell.

**9. Documentación y Makefile**

* Escribir un **Makefile** claro y eficiente que no realice relink innecesario.
* Documentar el código y crear un **README** explicativo.

**PROCESO**

**1. Entender el Objetivo**

El objetivo es que tu programa **pipex** funcione de manera similar a la ejecución de un comando de shell como:

shCopy code

< archivo1 comando1 | comando2 > archivo2

Esto implica tomar la salida de **comando1** aplicado a **archivo1** y usarla como entrada para **comando2**, escribiendo el resultado en **archivo2**.

**2. Manejo de Argumentos**

El programa debe aceptar cuatro argumentos:

* **archivo1**: el archivo de entrada para **comando1**.
* **comando1**: el primer comando para ejecutar en la entrada.
* **comando2**: el segundo comando cuya entrada es la salida de **comando1**.
* **archivo2**: el archivo donde se debe escribir la salida de **comando2**.

**3. Crear un Pipe**

Usa la función **pipe()** para crear un pipeline entre **comando1** y **comando2**. Un pipe es un buffer que permite la comunicación entre procesos.

**4. Uso de fork()**

Usarás **fork()** para crear procesos hijos que ejecuten **comando1** y **comando2**.

**5. Redirección de Entrada y Salida**

* En el proceso hijo que ejecuta **comando1**, redireccionarás la salida estándar (**STDOUT**) al extremo de escritura del pipe.
* En el proceso hijo que ejecuta **comando2**, redireccionarás la entrada estándar (**STDIN**) al extremo de lectura del pipe.

**6. Ejecución de Comandos**

Utilizarás **execve()** o una función similar para ejecutar **comando1** y **comando2** en los respectivos procesos hijos.

**7. Cierre de Archivos y Pipes**

Asegúrate de cerrar los descriptores de archivo y pipes que no se necesiten en cada proceso para evitar fugas de recursos.

**8. Esperar a que los Hijos Terminen**

El proceso padre debe esperar a que los procesos hijos terminen su ejecución antes de finalizar.

**Pasos Detallados**

1. **Parsear los Argumentos:** Extrae los argumentos de línea de comandos y verifica su validez.
2. **Crear un Pipe:** Llama a **pipe()** antes de **fork()** para crear un canal de comunicación entre los procesos hijo.
3. **Primer fork() para comando1:**
   * En el hijo: Redirige la salida estándar (**STDOUT**) al extremo de escritura del pipe y ejecuta **comando1** usando **execve()**.
   * En el padre: Espera y continúa.
4. **Segundo fork() para comando2:**
   * En el hijo: Redirige la entrada estándar (**STDIN**) al extremo de lectura del pipe y ejecuta **comando2**.
   * En el padre: Espera y continúa.
5. **Manejar archivo1 y archivo2:**
   * Asegúrate de que **archivo1** sea la fuente de datos para **comando1** y que la salida de **comando2** se escriba en **archivo2**.
6. **Cerrar Pipes y Archivos Descriptores:**
   * Asegúrate de cerrar todos los descriptores de archivos y pipes que no se usen en los procesos padre e hijo para evitar fugas de recursos.
7. **Esperar a los Procesos Hijos:**
   * El proceso padre debe esperar a que ambos procesos hijos terminen.

**TEORIA**

**Funciones:**

**1.Open:**

Prototipo:

int open(const char \*pathname, int flags, ... /\* mode\_t mode \*/)

**Parámetros**:

* **const char \*pathname**: Es el nombre del archivo a abrir.
* **int flags**: Indica cómo se debe abrir el archivo. Algunas de las banderas más comunes incluyen:
  + **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\_APPEND**: Escribe datos al final del archivo.
  + **O\_TRUNC**: Si el archivo ya existe y es abierto para escritura, su longitud se trunca a 0.
* **mode\_t mode** (opcional): Especifica los permisos para el nuevo archivo si se crea con **O\_CREAT**. Se ignora si no se usa **O\_CREAT**.

**0** inicial indica que el número que sigue debe ser interpretado como un valor octa

**2. read**

**Prototipo**:

ssize\_t read(int fd, void \*buf, size\_t count);

**Explicación**:

* **read** se utiliza para leer datos de un archivo.
* **int fd** es el descriptor de archivo que obtienes al abrir un archivo con **open**.
* **void \*buf** es un puntero al buffer donde se almacenarán los datos leídos.
* **size\_t count** es la cantidad máxima de bytes que se leerán.
* Retorna el número de bytes leídos o **-1** en caso de error.

**3. write**

**Prototipo**:

ssize\_t write(int fd, const void \*buf, size\_t count);

**Explicación**:

* **write** se utiliza para escribir datos en un archivo.
* **int fd** es el descriptor de archivo.
* **const void \*buf** es un puntero al buffer que contiene los datos a escribir.
* **size\_t count** es el número de bytes a escribir.
* Retorna el número de bytes escritos o **-1** en caso de error.

**4. close**

**Prototipo**:

int close(int fd);

**Explicación**:

* **close** se utiliza para cerrar un archivo abierto.
* **int fd** es el descriptor de archivo.
* Retorna **0** en caso de éxito o **-1** en caso de error.

Con estos detalles, deberías tener una buena referencia para continuar con tu aprendizaje y documentación.

**5**. **Función fork()**

* **Propósito:** **fork()** se utiliza para crear un nuevo proceso, que es una copia del proceso actual. Este nuevo proceso se denomina proceso hijo, mientras que el proceso que llamó a **fork()** se conoce como proceso padre.
* **Sintaxis:**

#include <unistd.h>

pid\_t fork(void);

* **Parámetros:** **fork()** no toma parámetros.
* **Valor de Retorno:**
  + En el **proceso padre**, retorna el PID (identificador de proceso) del proceso hijo.
  + En el **proceso hijo**, retorna **0**.
  + Si hay un error (por ejemplo, si el sistema no puede asignar recursos suficientes para crear un nuevo proceso), retorna **-1**.

**6. Función pipe()**

* **Propósito:** **pipe()** crea un canal de comunicación unidireccional entre procesos, representado por dos descriptores de archivo: uno para lectura y otro para escritura.
* **Sintaxis:**

#include <unistd.h> int pipe(int pipefd[2]);

* **Parámetros:** **pipefd** es un array de dos enteros. **pipefd[0]** es el extremo de lectura, y **pipefd[1]** es el extremo de escritura.
* **Valor de Retorno:** Retorna **0** en caso de éxito y **-1** en caso de error.

**CONFIGURACIÓN BÁSICA DEL PIPE**

**Configuración Básica**

Imagina que tienes dos comandos, **comando1** y **comando2**. En un shell de Unix, podrías conectar estos comandos usando un pipeline así:

shCopy code

comando1 | comando2

Aquí, la salida de **comando1** se convierte directamente en la entrada para **comando2**.

**Implementación en pipex**

Para imitar esto en tu programa **pipex**, realizas los siguientes pasos:

1. **Crear un Pipe:**
   * Antes de llamar a **fork()**, llamas a **pipe()** para crear un pipe.
   * Esto te da dos descriptores de archivo: **pipefd[0]** para leer y **pipefd[1]** para escribir.
2. **Primer fork() para comando1:**
   * Creas un proceso hijo con **fork()**.
   * **En el proceso hijo (que ejecutará comando1):**
     + Rediriges la salida estándar (**STDOUT**, normalmente **fd 1**) al extremo de escritura del pipe (**pipefd[1]**).
     + Esto se hace con **dup2(pipefd[1], STDOUT\_FILENO)**.
     + Luego, ejecutas **comando1** con **execve()** o una función similar.
     + **Importante:** Cierras **pipefd[0]** en este proceso hijo, ya que no se usa para leer.
3. **Segundo fork() para comando2:**
   * Creas otro proceso hijo.
   * **En este proceso hijo (que ejecutará comando2):**
     + Rediriges la entrada estándar (**STDIN**, normalmente **fd 0**) al extremo de lectura del pipe (**pipefd[0]**).
     + Esto se hace con **dup2(pipefd[0], STDIN\_FILENO)**.
     + Luego, ejecutas **comando2**.
     + **Importante:** Cierras **pipefd[1]** en este proceso hijo, ya que no se usa para escribir.
4. **Cierre de Pipes en el Proceso Padre:**
   * En el proceso padre, cierras ambos extremos del pipe (**pipefd[0]** y **pipefd[1]**), ya que la comunicación se maneja completamente entre los procesos hijo.

**Diagrama Visual**

scssCopy code

[comando1] --(stdout)--> |pipefd[1]| ===|pipe|=== |pipefd[0]| --(stdin)--> [comando2]

* Los datos fluyen desde **comando1** a **comando2** a través del pipe.
* **pipefd[1]** es el extremo de escritura conectado a la salida estándar de **comando1**.
* **pipefd[0]** es el extremo de lectura conectado a la entrada estándar de **comando2**.

**Resumen**

En resumen, **pipe()** en **pipex** se utiliza para crear un canal entre dos procesos hijos, donde uno escribe en el pipe y el otro lee desde el pipe. Esto imita la funcionalidad de un pipeline en shell, permitiendo que la salida de un comando se pase directamente como entrada al siguiente comando. El manejo adecuado de los descriptores de archivo y la sincronización entre procesos son clave para que esto funcione correctamente.