dos

Programa Shell

Objetivo del Ejercicio: Aprenderá a escribir un programa shell UNIX. Esto le dará la oportunidad de aprender cómo se crean los procesos hijo para realizar trabajos de gran envergadura y cómo el proceso padre puede seguir el trabajo de un proceso hijo.

Introducción

En la Parte 1, Sección 3.6, estudiaste cómo se asigna un proceso a cada puerto de entrada cuando la máquina arranca y cómo se ejecuta una copia de un programa shell en nombre de un usuario que entra en cualquier puerto. Un *shell*, o *programa de línea de comandos inter- preter*, es un mecanismo con el cual cada usuario interactivo puede enviar comandos al SO y mediante el cual el SO puede responder al usuario. Cada vez que un usuario ha iniciado sesión con éxito en el ordenador, el SO hace que el proceso de usuario asignado al puerto de inicio de sesión ejecute un shell específico. El SO no suele tener una interfaz de ventana incorporada. En su lugar, asume una simple interfaz orientada a caracteres en la que el usuario teclea una cadena de caracteres (terminada pulsando la tecla Enter o Return) y el SO responde tecleando líneas de caracteres de vuelta a la pantalla. Si la interfaz persona-ordenador va a ser una interfaz gráfica de ventanas, entonces el software que implementa el gestor de ventanas

engloba las tareas del shell que son el objetivo de este ejercicio. Así, el shell orientado a caracteres asume una pantalla con un número fijo de líneas (normalmente 25) y un número fijo de caracteres (normalmente 80) por línea.

Una vez que el shell ha inicializado sus estructuras de datos y está listo para empezar a trabajar, borra la pantalla de 25 líneas e imprime un prompt en las primeras posiciones de caracteres de la primera línea. Los sistemas Linux suelen estar configurados para incluir el nombre de la máquina como parte del prompt. Por ejemplo, mi máquina Linux se llama kiowa.cs.colorado.edu, así que el intérprete de comandos imprime, como su cadena de prompt,

kiowa>

o

bash>

dependiendo del shell que esté utilizando. A continuación, el intérprete de comandos espera a que el usuario escriba una línea de comandos en respuesta al prompt. La línea de comando puede ser una cadena como

kiowa> ls -al

terminan con un carácter enter o return (en Linux, este carácter es representado internamente por el carácter NEWLINE, '\n'). Cuando el usuario introduce una línea de comandos, el trabajo del shell es hacer que el sistema operativo ejecute el comando embebido en la línea de comandos.

Cada shell tiene su propia sintaxis y semántica del lenguaje. En el shell estándar de Linux, bash, una línea de comandos tiene la forma

comando argumento_1 argumento_2 ...

en el que la primera palabra es la orden que se va a ejecutar y las palabras restantes son los argumentos esperados por esa orden. El número de argumentos depende del comando que se ejecute. Por ejemplo, el comando de listado de directorios puede no tener argumentos -simplemente el usuario escribe ls- o puede tener argumentos precedidos por el carácter negativo "-", como en ls -al, donde a y l son argumentos. El comando determina la sintaxis de los argumentos, como cuáles pueden agruparse (como a y l en el comando ls), qué argumentos deben ir precedidos del carácter "-" y si la posición del argumento es importante.

Otros comandos utilizan una sintaxis de paso de argumentos diferente. Por ejemplo, un comando del compilador de C puede tener el siguiente

aspecto

```
kiowa> cc -g -o desviación -S main.c inout.c -lmath
```

en el que los argumentos g, o desviación, S, main.c, inout.c y lmath se pasan al compilador de C, cc.

El shell se basa en una importante convención para cumplir su cometido: La com-

para la línea de comandos suele ser el nombre de un archivo que contiene un exe- 2
programa cutable, por ejemplo, ls y cc (archivos almacenados en /bin en la mayoría de los

máquinas tipo UNIX). En algunos casos, el comando no es un nombre de archivo, sino un comando implementado en el shell. Por ejemplo, cd (cambiar directorio) suele implementarse dentro del propio shell en lugar de en un archivo en /bin. Dado que la gran mayoría de los comandos se implementan en archivos, puede pensar que el comando es en realidad un nombre de archivo en algún directorio de la máquina. Esto significa que el trabajo del shell es encontrar el archivo, preparar la lista de parámetros para el comando y luego hacer que el comando se ejecute utilizando los parámetros.

Muchos programas de shell se utilizan con variantes de UNIX, como el shell Bourne original (sh), el shell C (csh) con sus características adicionales sobre sh, el shell Korn y el shell Linux estándar (bash). Todos han seguido un conjunto similar de reglas para la sintaxis de la línea de comandos, aunque cada uno tiene un superconjunto de características.

Funcionamiento básico del shell al estilo UNIX

El shell Bourne se describe en el artículo original de Ritchie y Thompson sobre UNIX [Ritchie y Thompson, 1974]. Como se describió en la subsección anterior, el shell debe aceptar una línea de comando del usuario, analizar la línea de comando y luego invocar al SO para ejecutar el comando especificado con los argumentos especificados. Esta línea de comando es una solicitud para ejecutar el programa en cualquier archivo que contenga un programa, incluyendo programas escritos por el usuario. Así, un programador puede escribir un programa C ordinario, compilarlo y hacer que el shell lo ejecute como si fuera un comando UNIX.

Por ejemplo, suponga que escribe un programa en C en un archivo llamado main.c y luego lo compila y ejecuta con comandos del shell como

```
kiowa> cc main.c
kiowa> a.out
```

Para la primera línea de comandos, el shell buscará el comando cc (el compilador de C) en el directorio /bin y luego, cuando se ejecute el comando cc, le pasará la cadena main.c. El compilador de C, por defecto, traducirá el programa en C que está almacenado en main.c y escribirá el

programa ejecutable resultante en un archivo llamado a.out en el directorio actual. La segunda línea de comandos es simplemente el nombre del archivo a ejecutar, a.out, sin ningún parámetro. El shell encuentra el archivo a.out en el directorio actual y lo ejecuta.

Considera los siguientes pasos que debe dar un caparazón para cumplir su cometido.

1. Imprime un aviso.

   Existe una cadena de prompt por defecto, a veces codificada en el shell, por ejemplo la cadena de un solo carácter %, #, o >. Cuando el intérprete de comandos se inicia, puede buscar el nombre de la máquina en la que se está ejecutando y anteponer esta cadena al carácter estándar del prompt, por ejemplo una cadena de prompt como kiowa>. El intérprete de comandos también puede diseñarse para imprimir el directorio actual como parte del prompt, lo que significa que cada vez que el usuario escribe cd para cambiar a un directorio diferente, la cadena prompt se redefine. Una vez que se determina la cadena prompt, el shell la imprime en std- out cada vez que está listo para aceptar una línea de comandos.

2. Obtener la línea de comandos.

   Para obtener una línea de comandos, el intérprete de comandos realiza una operación de lectura de bloqueo, de modo que el proceso que ejecuta el intérprete de comandos permanecerá inactivo hasta que el usuario escriba una línea de comandos en respuesta al indicador. Una vez que el usuario escribe la línea de comandos (y la termina con un carácter NEWLINE ('\n')), la cadena de línea de comandos se devuelve al shell.

3. Analiza el comando.

   La sintaxis de la línea de órdenes es trivial. El analizador sintáctico comienza en la parte izquierda de la línea de órdenes y la recorre hasta que ve un carácter de espacio en blanco (como espacio, tabulador o NEWLINE). La primera palabra es el nombre de la orden y las siguientes son los parámetros.

4. Encuentra el archivo.

   El shell proporciona un conjunto de *variables de entorno* para cada usuario. Estas variables se definen primero en el archivo .login del usuario, aunque pueden modificarse en cualquier momento utilizando el comando set. La variable de entorno PATH es una lista ordenada de rutas absolutas que especifica dónde debe buscar el shell los archivos de comandos. Si el archivo .login tiene una línea como

   ```
   set path=(. /bin /usr/bin)
   ```

   el intérprete de comandos buscará primero en el directorio actual (ya que el primer nombre de ruta completo es "." para el directorio actual), después en /bin y, por último, en /usr/bin. Si no se encuentra ningún archivo con el mismo nombre que el comando (desde la línea de comandos) en ninguno de los directorios especificados, el intérprete

de comandos notifica al usuario que no puede encontrar el comando.

5. Prepara los parámetros.

   El shell simplemente pasa los parámetros al comando como argv matriz de punteros a cadenas.

6. Ejecuta el comando.

La shell debe ejecutar el programa ejecutable en el archivo especificado. Las shell de UNIX siempre han sido diseñadas para proteger al proceso original de colapsar cuando ejecuta un programa. Es decir, como un comando puede ser *cualquier* archivo ejecutable, entonces el proceso que está ejecutando el shell debe pro-
tectarse en caso de que el archivo ejecutable contenga un error fatal. De algún modo, el 2
El shell quiere lanzar el ejecutable de forma que incluso si el ejecutable sufre un error fatal (que destruye el proceso que lo ejecuta), el shell permanezca ileso. El shell Bourne utiliza múltiples procesos para lograr esto mediante el uso de las llamadas al sistema estilo UNIX fork(), execve() y wait().

- bifurcarse()

  La llamada al sistema fork() *crea* un nuevo proceso que es una copia del proceso llamado, excepto que tiene su propia copia de la memoria, su propio ID de proceso (con las relaciones correctas con otros procesos), y sus propios punteros a entidades compartidas del núcleo como descriptores de fichero. Después de llamar a fork(), *dos* procesos ejecutarán la siguiente sentencia después de fork() en sus propios espacios de direcciones: el padre y el hijo. Si la llamada tiene éxito, entonces en el proceso padre fork() devuelve el ID del proceso hijo recién creado y en el proceso hijo, fork() devuelve un valor cero.

- execve()

  La llamada al sistema execve() *cambia* el programa que está ejecutando un proceso. Tiene la forma

  ```
  execve(char *ruta, char *argv[], char *envp[])
  ```

  El argumento path es la ruta de un fichero que contiene el nuevo programa a ejecutar. La matriz argv es una lista de cadenas de parámetros, y la matriz envp es una lista de cadenas y valores de variables de entorno que deberían utilizarse cuando el proceso comience a ejecutar el nuevo programa. Cuando un proceso encuentra la llamada al sistema execve(), la siguiente instrucción que ejecuta será la del punto de entrada del nuevo archivo ejecutable. Por lo tanto, el núcleo realiza una cantidad considerable de trabajo en esta llamada al sistema. Debe

  - encontrar el nuevo archivo ejecutable,
  - cargar el archivo en el espacio de direcciones utilizado actualmente por el proceso de llamada (sobrescribiendo y descartando el

programa anterior),

- establecer la matriz argv y las variables de entorno para la nueva ejecución del programa, y

- iniciar el proceso que se ejecuta en el punto de entrada del nuevo programa.

  Existen varias versiones de execve()en la interfaz de llamada al sistema, que difieren en la forma de especificar los parámetros (por ejemplo, algunas utilizan una ruta completa para el archivo ejecutable y otras no).

- espera()

  La llamada al sistema wait() es utilizada por un proceso para bloquearse hasta que el kernel le indique que se ejecute de nuevo, por ejemplo porque uno de sus procesos hijo ha terminado. Cuando la llamada wait() vuelve como resultado de la terminación de un proceso hijo, el estado del proceso hijo terminado se devuelve como parámetro al proceso que llama.

Cuando se utilizan estas tres llamadas al sistema, este es el esqueleto de código que un shell podría utilizar para ejecutar un comando.

```
if(fork() == 0) {
  // Este es el niño
  // Ejecutar en el mismo entorno que el padre
    execvp(full_pathname, command->argv, 0);

} else {
  // Este es el padre-espera a que el hijo termine
    wait(status);
}
```

Poner un proceso en segundo plano

En el paradigma normal para ejecutar un comando, el proceso padre crea un proceso hijo, lo inicia ejecutando el comando, y luego espera hasta que el proceso hijo termine. Si se utiliza el operador "and" ("&") para terminar la línea de comandos, se espera que el shell cree el proceso hijo y lo inicie ejecutando el comando designado, pero sin que el proceso padre espere a que el proceso hijo termine. Es decir, el proceso padre y el proceso hijo se ejecutarán *simultáneamente*. Mientras el proceso hijo ejecuta el comando, el proceso padre imprime otro prompt en stdout y espera a que el usuario introduzca otra línea de comando. Si el usuario inicia varios comandos, cada uno terminado por un "&", y cada uno tarda un tiempo relativamente largo en ejecutarse, entonces muchos procesos pueden estar ejecutándose al mismo tiempo.

Cuando se crea un proceso hijo y comienza a ejecutarse en su propio

programa, tanto el hijo como el padre esperan que su flujo de entrada (stdin) provenga del usuario a través del teclado y que su flujo de salida (stdout) se escriba en la pantalla del terminal de caracteres. Tenga en cuenta que si varios procesos hijo se ejecutan de forma concurrente y

todos esperan que el teclado defina su flujo stdin, entonces el usuario no sabrá qué proceso hijo recibirá datos en su stdin si se escriben datos en el teclado. Del mismo modo, si cualquiera de los procesos concurrentes escribe caracteres en stdout, esos caracteres se escribirán en la pantalla del terminal dondequiera que el proceso hijo
el cursor. El núcleo no prevé dar a cada 2
proceso hijo su propio teclado o terminal (a diferencia de un sistema Windows, que controla la multiplexación y demultiplexación mediante acciones explícitas del usuario).

Redirección de E/S

Un proceso, cuando se crea, tiene tres identificadores de fichero por defecto: stdin, stdout y stderr. Si lee de stdin, los datos que reciba se dirigirán desde el teclado al descriptor de fichero stdin. Del mismo modo, los datos recibidos de stdout y stderr se asignan a la pantalla del terminal.

El usuario puede redefinir stdin o stdout cada vez que introduce un comando. Si el usuario proporciona un argumento de nombre de archivo al comando y precede el nombre de archivo con un carácter de llave angular izquierda ,"<", el shell sustituirá stdin por el archivo designado; esto se denomina redirigir la entrada desde el archivo designado.

El usuario puede redirigir la *salida* (para la ejecución de un único comando) precediendo un nombre de archivo con el carácter de llave angular derecha, ">". Por ejemplo, un comando como

```
kiowa> wc < main.c > programa.stats
```

creará un proceso hijo para ejecutar el comando wc. Antes de lanzar el comando, sin embargo, redirigirá stdin para que lea el flujo de entrada del archivo main.c y redirigirá stdout para que escriba el flujo de salida en el archivo program.stats.

El shell puede redirigir la E/S manipulando los descriptores de fichero del proceso hijo. Un proceso hijo recién creado hereda los descriptores de archivo abiertos de su padre, específicamente el mismo teclado para stdin y la pantalla del terminal para stdout y stderr. (Esto explica por qué los procesos concurrentes leen y escriben el mismo teclado y pantalla). El shell puede cambiar los descriptores de fichero del hijo para que lea y escriba flujos en ficheros en lugar de en el teclado y la pantalla.

Cada proceso tiene su propia tabla de descriptor de fichero en el núcleo (llamada fileDescriptor en esta discusión, pero etiquetada de forma diferente en el código fuente); para más información sobre el descriptor de fichero, ver Parte 1, Sección 7.2. Cuando se crea el proceso, la primera entrada

en esta tabla, por convención, se refiere al teclado y las dos segundas se refieren a la pantalla del terminal.

A continuación, el entorno de ejecución de C y el kernel gestionan los símbolos stdin, stdout y stderr de forma que stdin siempre está vinculado a fileDescriptor[0] en la tabla del kernel, stdout está vinculado a fileDescriptor[1] y stderr a fileDescriptor[2].

Puede utilizar la llamada al sistema close() para cerrar cualquier fichero abierto, incluyendo stdin, stdout y stderr. Por convención, los comandos dup() y open() siempre utilizan la primera entrada disponible en la tabla de descriptores de fichero. Por lo tanto, un fragmento de código como el siguiente:

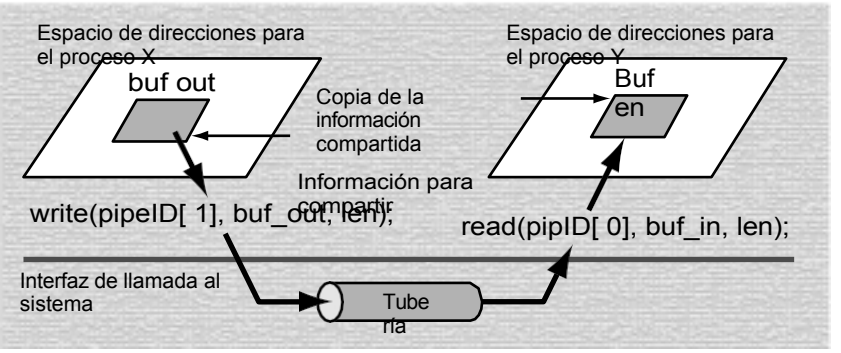
```
fid = open(foo, O_WRONLY | O_CREAT);
close(1);
dup(fid);
close(fid);
```

se garantiza la creación de un descriptor de fichero, fid, para duplicar la entrada y colocar el duplicado en fileDescriptor[1] (la entrada stdout habitual en la tabla de descriptores de fichero del proceso). Como resultado, los caracteres escritos por el proceso en stdout se escribirán en el fichero foo. Esta es la clave para redirigir tanto stdin como stdout.

Tuberías de concha

La *tubería* es el principal mecanismo IPC en Linux uniprocesador y otras versiones de UNIX. Por defecto, una tubería emplea operaciones de envío asíncrono y recepción bloqueante. Opcionalmente, la operación de recepción bloqueante puede cambiarse a una recepción no bloqueante (ver Sección 2.1.5 para más detalles sobre la invocación de operaciones de lectura no bloqueantes). Las tuberías son buffers FIFO (first-in/first out) diseñados con una API que se asemeja lo más posible a la interfaz de E/S de ficheros. Una tubería puede contener un número máximo de bytes definido por el sistema en un momento dado, normalmente 4KB. Como se indica en la Figura 2.2, un proceso puede enviar datos escribiéndolos en un extremo de la tubería y otro puede recibirlos leyendo el otro extremo de la tubería.

Figura 2.2 Flujo de información por una tubería



Una tubería está representada en el núcleo por un descriptor de fichero. Un proceso que quiere crear una tubería llama al kernel con una llamada de la siguiente forma.

```
int pipeID[2];
...
pipe(pipeID);
```


El núcleo crea la tubería como una estructura de datos FIFO del núcleo con dos identificadores de archivo. En este código de ejemplo, pipeID[0] es un puntero de archivo (un índice en la tabla de archivos abiertos del proceso) al extremo de lectura de la tubería y pipeID[1] es un puntero de archivo al extremo de escritura de la tubería.

Para que dos o más procesos utilicen tuberías para IPC (comunicación entre procesos), un ancestro común de los procesos debe crear la tubería antes de crear los procesos. Dado que el comando fork crea un hijo que contiene una copia de la tabla de archivos abiertos (es decir, el hijo tiene acceso a todos los archivos que el padre ya ha abierto), el hijo hereda las tuberías que creó el padre. Para utilizar una tubería, sólo necesita leer y escribir los descriptores de fichero adecuados.

Por ejemplo, supongamos que un padre crea una tubería. A continuación, puede crear un hijo y comunicarse con él mediante un fragmento de código como el siguiente.

```
...
pipe(pipeID);
if(fork() = = 0) { /* El proceso hijo */
  ...
  read(pipeID[0], childBuf, len);
  /* Procesar el mensaje en childBuf */
  ...
} else { /* El proceso padre */
  ...
  /* Enviar un mensaje al hijo */
  write(pipeID[1], msgToChild, len);
  ...
}
```

Se utiliza una tubería en lugar de un buzón. En un buzón, la operación de envío asíncrono es una llamada al sistema write() normal en el extremo de escritura de la tubería (descriptor de tubería pipe_id[1]) y la operación read() es una lectura de bloqueo en el extremo de lectura de la tubería (descriptor de tubería pipe_id[0]).

Las tuberías permiten a los procesos copiar información de un espacio de direcciones a otro utilizando el modelo de ficheros UNIX. Los extremos de lectura y escritura de las tuberías pueden utilizarse en la mayoría de las llamadas al sistema del mismo modo que un descriptor de fichero. Además, la información

escrito y leído de la tubería es un flujo de bytes. Las tuberías UNIX no soportan explícitamente mensajes, aunque dos procesos pueden establecer su propio protocolo para proporcionar mensajes estructurados. Además, existen rutinas de biblioteca que pueden utilizarse con una tubería para comunicarse mediante mensajes.

La figura 2.3 ilustra cómo se pueden utilizar las tuberías en Linux para implementar el procesamiento concurrente entre los procesos A y B (ejecutando PROC_A y PROC_B, respectivamente), donde A realiza algún cálculo ("cálculo A1"), envía un valor x a B, realiza una segunda fase de cálculo, lee un valor y de B, y luego itera. Mientras tanto, B lee el valor x, realiza la primera fase de

Figura 2.3
Tuberías Linux

```
int A_a_B[2], B_a_A[2]; main(){
    pipe(A_a_B);
    pipe(B_a_A);
    if (fork()==0) { /* Este es el primer proceso hijo */
        close(A_to_B[0]);
        close(B_to_A[1]);
        execve("prog_A.out", ...);
        exit(1); /* Error-terminar el hijo */
    }
    if (fork()==0) { /* Este es el segundo proceso hijo */
        close(A_to_B[1]);
        close(B_to_A[0]);
        execve("prog_B.out", ...);
        exit(1); /* Error-terminar el hijo */
    }
/* Este es el código del proceso padre
    */ wait(...);
    wait(...);
}

proc_A(){
    while (TRUE) {
        <compute A1>;
        write(A_to_B[1], x, sizeof(int)); /* Utiliza esta tubería para enviar
        información */
        <compute A2>;
        read(B_to_A[0], y, sizeof(int)); /* Utiliza esta tubería para obtener
        información */
    }
}

proc_B(){
    while (TRUE) {
        read(A_to_B[0], x, sizeof(int)); /* Utiliza esta tubería para obtener
        información */
        <compute B1>;
        write(B_to_A[1], y, sizeof(int)); /* Utiliza esta tubería para enviar
        información */
        <compute B2>;
    }
}
```

calcula, escribe un valor en A, realiza más cálculos y luego itera. Un proceso que no tiene la intención de utilizar un extremo de la tubería debe cerrarse para que las condiciones de fin de archivo (EOF) puedan ser detectadas.

Se puede utilizar una *tubería con nombre* para permitir que procesos no relacionados se comuniquen con 2
entre sí. Normalmente en las tuberías, los hijos heredan los extremos de la tubería como archivo abierto de-
scripts. En las tuberías con nombre, un proceso obtiene un final de tubería utilizando una cadena que es análoga a un nombre de archivo pero que está asociada a una tubería. Esto permite a cualquier conjunto de procesos intercambiar información utilizando una *tubería pública* cuyos nombres finales son nombres de fichero. Cuando un proceso utiliza una tubería con nombre, la tubería es un recurso de todo el sistema, potencialmente accesible por cualquier proceso. Al igual que los ficheros deben ser gestionados para que no sean compartidos inadvertidamente entre muchos procesos a la vez, las tuberías con nombre deben ser gestionadas, utilizando los comandos del sistema de ficheros.

Lectura de múltiples flujos de entrada

Como con cualquier fichero, el extremo de lectura de una tubería, un descriptor de fichero o un socket puede ser configurado, con una llamada a IOCTL(), para utilizar la semántica de no-bloqueo. Después de que la llamada ha sido emitida en el descriptor, una lectura en el flujo retorna inmediatamente, con el código de error establecido a EAGAIN. Además, read() devolverá el valor 0, indicando que no se ha leído ninguna información en el buffer. Alternativamente, el programa puede determinar si read() tuvo éxito comprobando el valor de longitud para ver si es distinto de cero.

La figura 2.4 ilustra el uso de ioctl() para cambiar el extremo de lectura de una tubería de su comportamiento de bloqueo por defecto a un comportamiento de no bloqueo. Puede aplicar esta técnica a cualquier descriptor de fichero, incluyendo stdin.

El comando select() permite a un proceso sondear todos sus flujos de entrada abiertos para determinar cuáles contienen datos. Entonces puede ejecutar una operación de lectura de bloqueo normal en cualquier flujo que contenga datos. Tenga en cuenta que si varios procesos están leyendo la tubería y cada uno utiliza select(), puede producirse una condición de carrera. Si decide utilizar este enfoque en su solución, utilice la página man para aprender más sobre select().

Parte A

Escribe un programa en C que actúe como intérprete de línea de comandos shell para el kernel Linux. Tu programa shell debe utilizar el mismo estilo que el shell Bourne para ejecutar programas. En particular, cuando el usuario escriba una línea como

identificador [identificador [identificador]]

Figura 2.4
Sin bloqueo leer Ejemplo

```
#include       <sys/ioctl.h>
int errno;     /* Para lectura no bloqueante flag */
...
main() {
  int pipeID[2];
  ...
  pipe(pipeID);
/* Cambia el extremo de lectura de la tubería al modo sin
  bloqueo */ ioctl(pipeID[0], FIONBIO, &on);
  ...
  while(...) {
  /* Sondea el extremo de lectura
    de la tubería */ read(pipeID[0],
    buffer, BUFLEN); if (errno
    !=EAGAIN){
    /* Información entrante disponible desde la tubería--procésala */
      ...
    } else {
    /* Comprueba la tubería de entrada de nuevo más tarde--haz otras cosas
    */
      ...
    }
  }
  ...
}
```

su shell debe analizar la línea de comandos para construir argv. Debe buscar en el sistema de directorios (en el orden especificado por la variable de entorno PATH) un archivo con el mismo nombre que el primer identificador (que puede ser un nombre de archivo relativo o una ruta completa). Si se encuentra el fichero, entonces debe ejecutarse con la lista de parámetros opcional, como se hace con sh. Utilice una llamada al sistema execv() (en lugar de cualquiera de las llamadas execl()) para ejecutar el archivo que contiene el com- mand. Deberá familiarizarse con las funciones de Linux fork()y wait() y con la familia de llamadas al sistema execv() (un conjunto de interfaces para la llamada al sistema execve()).

Parte B

Añada funcionalidad al intérprete de comandos de la Parte A para que el usuario pueda utilizar el operador "&" como terminador de comandos. Un comando terminado con "&" debe ser ejecutado simultáneamente con el shell (en lugar de que el shell espere a que el hijo termine antes de pedir al usuario otro comando).

Parte C

Modifica tu programa shell de la Parte A para que el usuario pueda redirigir la entrada stdin

o descriptores de archivo stdout utilizando los caracteres "<" y ">" como nombre de archivo previo.

soluciones. También, permita a su usuario utilizar el operador de tubería, "|", para ejecutar dos procesos concurrentemente, con stdout del primer proceso siendo redirigido como el stdin para el segundo proceso. Su solución puede manejar opcionalmente sólo la redirección o sólo las tuberías en una línea de comandos (en lugar de ambas opciones en la línea de comandos
misma línea de comandos).

Actación del Problema

Organizar una solución

La introducción del ejercicio describe de forma general cómo se comporta un caparazón. También proporciona implícitamente un plan de ataque, resumido aquí. Este plan describe varias versiones de depuración que puede utilizar para la Parte A, y luego aplicar a las otras partes según sea necesario.

1. Organizar el shell para inicializar variables y luego realizar un bucle sin fin hasta que stdin detecte una condición EOF, como un carácter Ctrl-D o un mensaje de salida. Desarrolle una versión muy simple que imprima el carácter prompt y luego espere a que el usuario escriba un comando. Después de leer el comando, debe imprimirlo en stdout.
2. Perfecciona tu shell simple para que *analice* la línea de comandos. Debería de- terminar las cadenas en la línea de comandos y luego ponerlas en un archivo char
   *array argv[]. Además, calcula el valor de int argc. En esta versión de depuración, imprime el contenido de argc y argv[]. También puede detectar un comando de salida que terminará su shell.
3. En la siguiente versión de depuración, utiliza argv[0] para encontrar el archivo ejecutable. En esta versión, simplemente imprime el nombre del archivo.
   - Construir una versión simple que sólo puede encontrar los archivos de comandos que se encuentran en el directorio actual.
   - Mejore su programa para que pueda encontrar archivos de comandos especificados con una ruta absoluta.
   - Permite que su programa busque directorios según la cadena almacenada en la variable de entorno PATH del shell.
4. Crea un proceso hijo para ejecutar el comando.

Parte A

He aquí un esqueleto de código para un programa shell.

```
int main () {
  ...
```

```
  struct
    comando_estructura {
    char *nombre;
    char *argv[];
    ...
  } *command;
  ...
// Inicialización del shell
  ...

// Bucle principal
  while(stdin no está en EOF) {
    print_to_stdout(prompt_string);
    command_line = read_from_stdin();
  // Determina el nombre del comando, y construye la lista de parámetros
    ...
  // Buscar la ruta completa del fichero
    ...
  // Lanzar el archivo ejecutable con los parámetros especificados
    ...
  }

// Terminar el shell
  ...
}
```

Determinar el nombre de la orden y la lista de parámetros

Debería reconocer el nombre argv en la command_struct de escribir programas C en sus clases de introducción a la programación (y del Ejercicio 1 de este manual). Es decir, si escribes un programa y quieres que el shell pase parámetros (desde la línea de comandos) a tu programa, entonces declaras la función pro- totype para tu programa principal con una línea como esta:

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

La convención es que cuando se ejecuta su programa ejecutable (a.out), el shell construye una matriz de cadenas, argv, con entradas argc en ella. argv[0] apunta a una cadena que tiene el nombre del comando a.out, argv [1] apunta a una cadena que especifica el primer parámetro, argv [2] apunta a una cadena que especifica el segundo parámetro, y así sucesivamente. Tu programa, cuando se ejecuta, lee la matriz argv para obtener las cadenas y luego aplica la semántica que quiera para interpretar las cadenas.

Por ejemplo, su programa podría ejecutarse con una línea de comandos de la forma

a.out foo 100

de modo que cuando comience la ejecución, argc tendrá el valor 3, argv[0] apuntará a a la cadena a.out, argv [1] apuntará a foo, y argv [2] apuntará al 2 cadena 100. Su programa puede interpretar el primer parámetro (argv[1]) como, por ejemplo, un nombre de archivo, y el segundo parámetro (argv[2]) como, por ejemplo, un número entero de registros. El intérprete de comandos simplemente trataría la primera palabra de la línea de comandos como un nombre de archivo y las palabras restantes como cadenas.

Después de que su programa lee la línea de comandos en una cadena, command_line, analiza la línea para rellenar los campos de command_struct (name y argv). La explicación de la introducción debería ser suficiente para que diseñes e implementes código para analizar la línea de comandos de forma que tengas el nombre del archivo que contiene el comando en command->name y (un puntero a) la matriz de punteros a cadenas de parámetros en command->argv.

Encontrar el nombre de ruta completo

El usuario puede haber proporcionado un nombre de ruta completo como la palabra de nombre del comando o sólo un nombre de ruta relativo que debe vincularse de acuerdo con el valor de la variable de entorno PATH. Un nombre que comience por "/", "./" o "../" es una ruta absoluta que puede utilizarse para iniciar la ejecución. De lo contrario, su programa debe buscar en cada directorio de la lista especificada por la variable de entorno PATH para encontrar la ruta relativa.

Lanzamiento del comando

El paso final en la ejecución del comando es crear un proceso hijo para ejecutar el comando especificado y luego hacer que el hijo ejecute ese comando. El siguiente esqueleto de código logrará esto.

```
if(fork() == 0) {
  // Este es el niño
  // Ejecutar en el mismo entorno que el padre
    execvp(full_pathname, command->argv, 0);

} else {
  // Este es el padre; espera a que el hijo termine
    wait(status);
}
```

Partes B y C

Obtener código general es difícil cuando se utiliza más de un operador especial en una línea de comandos, por lo que debe centrarse en hacer el ejercicio sólo cuando se utiliza "&", "<", ">", o "|" y, a continuación, no más de uno a la vez.