75.09 - Sistemas Operativos

Nombre: Blanco, Sebastián Ezequiel

Número de legajo: 98539

Nombre de la entrega: Lab Shell

Fecha de entrega: 27 de abril de 2018

GitHub: https://github.com/BlancoSebastianEzequiel/LabShell

Profesor: Mendez, Mariano

Aclaración

Tareas programadas:

- Parte 1: Invocación de comandos
  - Busqueda en $PATH
  - Argumentos del programa
  - Imprimir variables de entorno
- Parte 2: Invocacion avanzada
  - Comandos built-in
  - Variables de entorno adicionales
  - Procesos en segundo plano
- Parte 3: Restricciones
  - Flujo estándar
  - Tuberías simples (pipes)
- Challenges promocionales
  - Pseudo-variables
  - Tuberías Múltiples
  - Segundo plano avanzado

Preguntas

- **Búsqueda en $PATH**

  ¿Cuáles son las diferencias entre la *syscall* execve(2) y la familia de *wrappers* proporcionados por la librería estándar de *C* (*libc*) exec(3)?

Execve(2) es una función proporcionada por la API del kernel en la cual se reemplaza, el address space del proceso y es reemplazado por el del programa a ejecutar. Recibe un puntero a el nombre del ejecutable, un puntero a un array de cadenas que representan los argumentos y otro punteros a un array de cadenas que son las variables de entorno que tomara el nuevo proceso.
Los wrappers de exec son funciones que proporciona la librería estándar de C donde internamente se llama a la syscall execve(2). Cada uno de los wrappers tiene las funcionalidades acotadas, es decir, se ve que algunos no usan el argumento de la variables de entorno porque en este caso, el wrapper internamente le pasa la variable externa environ.

- **Variables de entorno adicionales**

  ¿por qué es necesario hacerlo luego de la llamada a *fork(2)?*

Es necesario ya que de esta manera las variables de entorno que se escriban solo existen en el proceso hijo, una vez finalizado este proceso, las variables de entorno que se hayan agregado no estarán en el proceso padre.

Supongamos, entonces, que en vez de utilizar setenv(3) por cada una de las variables, se guardan en un array y se lo coloca en el tercer argumento de una de las funciones de exec(3).
¿el comportamiento es el mismo que en el primer caso? Explicar qué sucede y por qué.

exec(BIN, argv, envp);
envp = { “USER=nadie”, “ENTORNO=nada” }

Hipótesis:

- Busca bin en path: Esto es cierto ya que al no tenes un slash (/) , procede a buscar el ejecutable en los directorios que están en PATH.
- Se agregan al env del hijo: Esto es cierto ya que se agregan en el proceso hijo y al salir del mismo, no aparecen en el proceso padre.
- Pisa el entorno entero: Falso, Lo que ocurre es que en este caso solo aparecerá estas dos variables de entorno, es decir, que todas las variables de entorno que están en el proceso padre no estarán, si no solo las que se pasen como parámetro, es este caso, “USER=nadie”, “ENTORNO=nada”.
-

Describir brevemente una posible implementación para que el comportamiento sea el mismo.

Para ser el mismo se debe pasar como argumento en envp la variable environ y agregar ademas las variables de entorno que se quieran, de este modo se tiene la misma implementación que con la función setenv.

exec(BIN, argv, environ+envp);
envp = { “USER=nadie”, “ENTORNO=nada” }

- **Procesos en segundo plano**

Mecanismo utilizado:

Cuando se escribe un comando con un & al final, este se parsea, se le quita y se crea un proceso hijo. Si se esta ejecutando el proceso hijo, se pregunta si el comando es un proceso de background, de ser así, se ejecuta el proceso con la función exec(3).

En caso de estar en el proceso padre, se imprime el pid por pantalla y se evita llamar a la función waitpid para esperar la finalización del proceso hijo. De esta manera, si un proceso se esta ejecutando en el proceso hijo, la función waitpid espera a que finalice el proceso padre, al evitarla el proceso padre puede seguir ejecutandose. Waitpid es una syscall que suspende la ejecución del proceso llamado hasta que los hijos cambien de estado.

- **Flujo estándar**

  Investigar el significado de este tipo de redirección (2>&1) y explicar qué sucede con la salida de cat out.txt. Comparar con los resultados obtenidos anteriormente. (ls -C /home /noexiste >out.txt 2>&1)

El significado es que la salida de error sera redireccionada a la salida de file descriptor 1, es decir, a la salida estándar. Pero a su vez, vemos que la salida estándar es redireccionada al archivo out.txt, por lo tanto, la salida de error es redireccionada al archivo out.txt.

> Challenge: investigar, describir y agregar la funcionalidad del operador de redirección >> y &>

El operador >> redirecciona la salida estándar a un archivo de salida deseado agregando lo que se escribió en la salida al archivo, a diferencia de > que sobrescribe en el archivo, es decir, que el operador >> seria como un append.

El operador &>archivo redirecciona la salida estandar de error y la salida estandar hacia un archivo deseado.

Estas dos lineas son equivalentes: ls &>archivo , ls 2>&1

- **Tuberías simples (pipes)**

  Investigar y describir brevemente el mecanismo que proporciona la syscall pipe(2), en particular la sincronización subyacente y los errores relacionados.

Crea un conducto, un canal de datos unidireccional que se puede usar para la comunicación entre procesos.

Pipe te devuelve dos file descriptors que refieren a un mismo archivo, un file descriptor es de escritura y el otro de lectura. De esta manera uno puede usar esto para hacer una tubería simple entre dos comandos ya que la salida estándar de uno se redirecciona usando el file descriptor de escritura y luego se lee , redireccionando la entrada estándar al file descriptor de lectura para poder ejecutar el segundo comando.

Para esto es necesario usar un fork para que el primer comando se ejecute usando un space address distinto al del segundo comando que se ejecutara en el padre.