Lab2: Un Baash

Objetivos

1. Utilizar los mecanismos de concurrencia y comunicación UNIX.
2. Implementar de manera *sencilla* un intérprete de línea de comandos (shell) al estilo de Bourne shell.

Introducción

La interfaz más tradicional de un sistema operativo tipo UNIX es el *intérprete de línea de comandos*. Este programa, que se ejecuta en modo usuario, funciona en cualquier UNIX que soporte interface de caracteres y su función es aceptar comandos ingresados por *entrada estandar* (teclado), parsearlos, ejecutar la órden y mostrar el resultado en la *salida estandar* (pantalla), para luego volver a repetir el proceso.

Por defecto UNIX ejecuta un proceso *shell* cada vez que un usuario interactivo ingresa al sistema. Aunque esto puede ser configurado de otra manera (ver el último campo de cada línea del archivo `/etc/passwd`), en la mayoría de los casos luego de ingresar nuestro nombre de usuario y contraseña, el proceso que maneja el ingreso de usuarios genera un *proceso hijo* que ejecuta un shell, con el uid/gid (identificador de usuario y grupo) correspondiente al usuario. En este momento la pantalla se suele presentar de la siguiente manera:

```
    [juan@hal juan]$
```

Después de este texto inicial llamado *prompt*, que contiene información de entorno como por ejemplo el nombre del usuario, el nombre del host y el último tramo del directorio corriente, el shell espera datos a través de la `stdin` que normalmente se asocia al dispositivo teclado. Podemos escribir el comando que deseamos que el shell ejecute, e iniciar la ejecución ingresando el caracter NEWLINE `'\n'` generalmente asociado con la tecla Enter o Return.

```
    [juan@hal juan]$ sleep 10
```

Hará que el shell ejecute un proceso con el programa binario que se encuentra en `/bin/sleep`, pasándole el argumento `"10"`.

Operación Básica

La sintáxis básica del intérprete de comandos más usual de UNIX, conocido como Bourne shell (Bourne again shell - bash, en Linux) es la siguiente,

```
    comando argumento1 argumento2 ...
```

donde el comando y los argumentos son secuencias de caracteres separados por uno o más espacios.

La semántica dice que al presionar Enter el shell buscará el `comando` dentro de los *comandos internos* y si no lo encuentra tratará de buscar un archivo ejecutable con ese nombre, siguiendo las reglas de camino de UNIX o componiendo el comando a la secuencia de la *variable de entorno* `PATH`, para luego crear un *proceso hijo* que cargará y ejecutará el contenido de ese archivo con los argumentos correspondientes.

Los *comandos internos* son manejados directamente por el shell, sin requerir de ningún archivo externo. Un ejemplo es el comando de cambio de directorio `cd`, el cual no se encuentra como archivo ejecutable en ningún lugar del árbol de directorio (el comando `find /bin /sbin /usr/bin /usr/sbin -perm +400 -type f -name cd` no devuelve nada). Con `man bash-builtins` obtenemos una lista de todos los comandos internos implementados en `bash`.

Si el comando no es un *builtin*, el shell deberá buscar un archivo dentro de su sistema de archivos, cargarlo y ejecutarlo pasándole los argumentos. El problema principal es dónde buscar este archivo. Existen tres formas:

- Camino absoluto
- Camino relativo
- Búsqueda en secuencia `PATH`

Cuando el comando comienza con `/`, este se toma como un *camino absoluto* dentro del árbol del filesystem, el shell cargará en memoria y ejecutará el comando. En cambio si el comando comienza con el nombre de un directorio, `.` o `..`, se debe seguir las reglas usuales de *camino relativo* de UNIX, cargar y ejecutar el archivo comando, relativo al camino actual (ver comando `pwd`).

Otro mecanismo entra en juego cuando el comando no comienza con un delimitador de camino absoluto o relativo. La variable de entorno `PATH`, que puede ser leida con el comando `env` o con `echo $PATH`, sirve de secuencia de caminos absolutos o relativos, separados por `':'` que serán prefijados a cada commando hasta encontrar un archivo que pueda ser leído y ejecutado.

Usemos el archivo ejecutable `/bin/date` que nos proporciona la fecha y hora del sistema para ejemplificar los mecanismos de camino absoluto, relativo y secuencia `PATH`.

```
[juan@hal juan]$ /bin/date
Sun Aug 31 15:33:22 ART 2010
[juan@hal juan]$ cd /usr
[juan@hal /usr]$ ../bin/date
Sun Aug 31 15:33:57 ART 2010
[juan@hal /usr]$ cd /
[juan@hal /]$ bin/date
Sun Aug 31 15:34:24 ART 2010
[juan@hal /]$ cd ~
[juan@hal juan]$ date
Sun Aug 31 15:35:02 ART 2010
```

Todos los comandos ejecutados por `bash` son el mismo `/bin/date`.

Ejercicios

1. ¿Cómo ejecutamos un archivo que se llama exactamente como un *builtin*?
2. ¿Por qué las recomendaciones de seguridad indican que es peligroso tener `./` en `PATH` al más puro estilo `DOS`?
3. Supongamos que existen 2 comandos posibles dentro de la secuencia que contiene `PATH`, donde el primero en la secuencia no está marcado como ejecutable y el segundo si. ¿Qué hace el intérprete `bash`, ejecuta el segundo o informa que el primero no tiene permiso de ejecución? (incorporen esta semántica a `baash`)
4. Indique qué sucede cuando se tiene un directorio en el `pwd` actual con el mismo nombre que un archivo ejecutable en el `PATH`. ¿Dice que no puede ejecutar un directorio o ejecuta el comando que encontró? (sigan esta forma en `baash`)
5. Obtengan la lógica más sencilla que unifique los tres tipos de búsqueda.
6. ¿Podemos poner directorios relativos en `PATH`? (hagan lo propio con `baash`)

Procesos en Segundo Plano

En el paradigma básico para ejecutar un comando, el shell es el proceso padre que crea a un hijo que cargará y ejecutará el comando que se encuentra en el filesystem. Mientras tanto el padre quedará esperando hasta que el proceso hijo termine.
La semántica puede ser cambiada para que el padre no espere la terminación del hijo, si le agregamos al final el símbolo '&'. Este operador, crea la posibilidad de ejecutar procesos de manera concurrente, es decir, si el comando que ejecutamos termina con & y demora un tiempo relativamente largo, todo lo que iniciemos desde el shell y el shell mismo estarán ejecutándose “al mismo tiempo”.

Esta característica resulta muy útil cuando nuestro kernel esta corriendo sobre un motherboard SMP (symmetric multiprocessing), que soporta varios microprocesadores compartiendo una única imagen de memoria principal. Si tenemos un par de programas que tardan mucho en calcular decimales de *pi* o de *e*, podemos ponerlos a correr de manera que cada uno ocupe un microprocesador, mientras utilizamos un poco de los dos micros para jugar.

```
[juan@deepblue calculosLargos]$ ./decimalesPi salidaPi.txt &
[juan@deepblue calculosLargos]$ ./decimalesE salidaE.txt &
[juan@deepblue calculosLargos]$ freeciv
```

Existe un problema con los procesos hijos en *background*, la entrada y salida estandar se comparten entre todos, con lo cual tendremos salidas de pantalla entrelazadas y las entradas por teclado serán no-deterministicamente asignadas a alguno de los procesos que están corriendo.
Este ejemplo nos muestra el lio que se puede armar.

```
[juan@hal juan]$ yes "HOLA" & yes "CHAU" &
... ud. acaba de perder el control de esa shell ...
```

Ejercicios

1. Investiguen cuales son los comandos internos para manejo de procesos en background de `bash`.
2. En el ejemplo de arriba el operador '&' funciona como operador de composición paralela. Hay un operador de composición secuencial en Bourne shell. ¿Cuál es?
3. Investiguen como `bash` forma el árbol de procesos cuando ejecutamos `cmd1 & cmd2 & cmd3 & ... & cmdN`. Piensen la respuesta y luego utilicen `pstree` para corroborarla. Apliquen los resultados de su observación a `baash`.
4. Indique cuantas letras `'a'` debería imprimir el siguiente programa:

```
fork();
fork();
fork();
printf("a");
```

Redirección de la Entrada/Salida

Cuando se crea un proceso, se le asignan tres identificadores de archivos (*file descriptors*): `stdin`, `stdout` y `stderr`. Si se lee desde `stdin`, los datos que se reciben serán dirigidos desde el teclado al file descriptor `stdin`. Similarmente, los datos escritos en `stdout`, `stderr` serán mapeados a la pantalla de la terminal.

El usuario puede redefinir el `stdin` desde la línea de comandos del shell si se provee al final del comando un `'<'` y un nombre de archivo. Este archivo reemplazará el flujo estandar de entrada, por lo que los valores que lea el comando serán extraidos desde el archivo.
Similarmente podemos redefinir el file descriptor de `stdout` con `'>'` y un nombre de archivo al final del comando, para indicar que toda la salida estandar del comando se acumule en dicho archivo.

Como ejemplo podemos recolectar estadísticas sobre un header del código fuente del kernel y ponerlo en un archivo de estadísticas.

```
[juan@hal juan]$ wc < kernel.h > kernel.h.stats
```

O, si nuestros programas generan sus resultados en pantalla, podemos ejecutarlos concurrentemente y mantener sus salidas separadas, mientras las monitoreamos con el comando `tail`.

```
[juan@deepblue juan]$ forest3d -n 0.2 > n0.2.out &
[juan@deepblue juan]$ forest3d -n 0.3 > n0.3.out &
[juan@deepblue juan]$ tail -f n0.2.out n0.3.out
```

Ejercicios

1. Las corrientes estandar `stdout` y `stderr` están dirigidas ambas a la consola. ¿Cómo podemos utilizar los opeardores de redirección para separarlas?
2. De más ejemplos de como usar la redireccion de entrada? Tenga cuidado que muchos programas deshabilitan la posibilidad de redirigir la entrada estandar a otra cosa que no sea una terminal.

Tuberías (pipes)

Las tuberías o pipes son el principal mecanismo de Comunicación entre Procesos (IPC) para sistemas UNIX. Una tubería se compone de dos puntas, una entrada y una salida, cada una conectada a un proceso. Las puntas de una tubería son un par de *file descriptors*, uno para escribir y el otro para leer. Un pipe es básicamente un buffer FIFO, donde tenemos un **productor**, sobre el *write-end* y un **consumidor** sobre el *read-end*. Como sabemos que todas las colas que se encuentran en el sistema operativo son acotadas (memoria finita), además de tener un read bloqueante cuando no hay datos para consumir, tenemos un send bloqueante cuando la capacidad del buffer está excedida.

El shell provee la funcionalidad de un pipe a través del operador '|', denominado justamente **pipe**, el cual conecta la salida estandar del proceso lanzado por el comando de la izquierda del pipe, con la entrada estandar del proceso que se genera con el comando a la derecha del símbolo de tubería.

Las tuberías se pueden componer secuencialmente para generar una *linea de producción de datos*, el cual es un estilo arquitectural denominado *pipes&filters*.

Estos ejemplos cuentan cuantas veces Juan ingresó al sistema, los pids (process identifiers) de todos los mozillas, y el listado ordenado de mayor a menor de todos los usuarios del sistema que tienen `bash`.

```
[juan@hal juan]$ last juan | wc -l
[juan@hal juan]$ ps aux | grep mozilla-bin
[juan@hal juan]$ grep bash /etc/passwd | cut -d ":" -f 1 | sort -r
```

Ejercicio

1. Los pipes existen en el filesystem mientras dura el comando. ¿Dónde se encuentran y que atributos tiene?

Tareas

El problema se divide en 3 partes, de manera que su concreción se realice en

forma progresiva. Es una guía de como llegar al resultado final, nuestro `baash`, sin tener que sortear todos los problemas de diseño y técnicos de una sola vez.

Parte A

Escriba un programa en "C" que actúe com un shell de intérprete de línea de comandos para el sistema operativo Linux. Su programa deberá utilizar la misma sintaxis de Bourne shell para correr programas.

`comando argumento1 argumento2 ...`,

efectuando la búsqueda del comando según se presente como path absoluto, relativo o involucre una búsqueda en la secuencia de la variable de entorno `PATH`. El programa deberá generar un proceso hijo para correr el comando, a fin de protegerse de comportamientos malos, y pasarle los argumentos correspondientes.
Es preferible imprimir un prompt con información relevante a la izquierda de donde se introduce el comando. El nombre del host, el nombre del usuario y el camino corriente pueden ser algunos ejemplos. Los únicos *comandos internos* que se piden son `exit`, que termina con la ejecución del intérprete de comandos, y `cd` (cambiar de directorio actual). Fijense en la llamada al sistema `chdir()`.

Luego de implementar esta parte, su `baash` deberá ser capaz de invocar desde simples peticiones como `date`, que involucran búsquedas en PATH, hasta llamadas a comandos de compilación o linking utilizando path relativos y decenas de opciones.

Parte B

Agregue al shell de la parte anterior, la funcionalidad de correr programas como *procesos en background*, o mejor dicho concurrentemente con el mismo `baash`. El usuario incluirá el operador `'&'` a la derecha del último argumento para activar esta capacidad. Prueben la nueva versión de la siguiente manera:

```
    baash-0.01> ./ksamp -l 3 120 &
    baash-0.01> ./ksamp -l 1 120 &
```

Deberían obtener una salida no determinística (si efectúan varias corridas) de la información generada por el programa del laboratorio 1.

Parte C

Incorporen las funcionalidades de redirección y tuberías a `baash`, de manera tal que se interpreten los opeardores `< filename`, `> filename` y `| command2 argumento2_1 argumento2_2 ...` al final de los argumentos del comando.
Deberá soportar **sólo uno** de los cuatro operadores al mismo tiempo.

Cómo atacar los problemas

La principal fuente de información son los capítulos V, VI de:
D. M. Ritchie and K. Thompson, "*The UNIX Time-Sharing System*" [html] [pdf]

Este documento indica de manera extremadamente clara como se logró la implementación del shell.

Parte A

Se puede ir paso a paso, generando programas de prueba que realicen tareas cada vez más complejas y cercanas al enunciado de la parte A. Si usted se siente confiado respecto a poder solucionar el problema sin hacer desarrollo incremental, puede saltar estos pasos.

1. Escriba un programa que inicialice las variables e ingrese en un lazo que pida entrada por teclado y que termine sólo cuando se ingresa una condición de `EOF` (end of file) como `Ctrl-D` o el comando interno `exit`.
2. Refine el punto anterior de manera tal que haga el *parsing* de la línea de comandos y cargue `char* argv[]` y `argc`.
3. En el siguiente refinamiento, use argv[0] para encontrar el archivo ejecutable, e impriman el path completo del archivo.
4. Mejore su programa para que acepte path absolutos y relativos.
5. Active en su prototipo la posibilidad de buscar en los directorios de la lista `PATH`.
6. Ahora sí, cree el hijo y ejecute el comando con sus argumentos.

Esta parte debería ser un **reflejo exacto** del poder de las llamadas a sistema para manejo de procesos `fork()`, `execv()`, y `wait()`. Todo lo que resta es el parsing de la línea de comandos y la impresión del prompt.

Veamos un ejemplo de uso del `fork` y `wait` para crear toda una parentela de procesos. Notar el interleaving que se produce en las ejecuciones.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

#define N 8

int
main (int argc, char **argv)
{
        unsigned int i,j,h,s;
        int status;
        if (argc!=2) {
               fprintf(stderr,"Uso: %s nro_de_hijos\n",argv[0]);
               exit(1);
        }
        h = atoi(argv[1]);
        for (i=0;i<h;i++) {
                s = fork();
                if (s<0) {
                       perror("Creando el hijo");
                       exit(1);
                }
                else if (s==0) {
                       for(j=0;j<N;j++)
                               printf("HIJO pid=%5d: %3d\n",getpid(),j);
                        return 0;       /*no quiero nietos!*/
                }
                else
                        printf("PADRE: hijo pid=%5d, creado\n",s);
        }
        for (i=0;i<h;i++) {
                s = wait(&status);
                if (s<0) {
                        perror("Esperando el hijo");
                        exit(1);
                }
                printf("PADRE: hijo pid=%5d, terminado\n",s);
        }
        return 0;
}
```

La otra llamada al sistema a utilizar es `execv`, que reemplaza la imagen del proceso por una nueva imagen que se encuentra en un archivo. En este pequeño ejemplo, se reemplaza el proceso actual por el que ejecuta el programa `date`.

```
#include <stdio.h>
#include <unistd.h>

int
main (int argc, char **argv)
{
        execvp("date",argv);
        printf("Llega hasta aca?\n");
        return 0;
}
```

Aunque `execvp()`, parece que le soluciona todos los problemas relativos a la búsqueda del path del ejecutable, **no se permite su utilización** y deberán codificar o reutilizar algoritmos y estructuras de datos apropiadas.

Parte B

El proceso que ejecuta el programa baash deberá correr concurrentemente con el proceso hijo creado para ejecutar el comando, si es que hay un operador '&' a la derecha de todos los argumentos. Simplemente no hay que esperar!

Parte C

Cuando se crea un proceso, este hereda de los descriptores de archivos abiertos de su padre. Además como el proceso 1, denominado `init` y padre de los restantes procesos del sistema (`ps aux | grep init`, o bien `pstree` para obtener el árbol genealógico), tiene abierto los tres primeros identificadores de archivos `stdin`, `stdout` y `stderr` (0,1 y 2) a descriptores de archivos asociados a la terminal de entrada y a la terminal de salida, todos los hijos heredan estos fileid's asociados a los filedescriptors.

Para implementar los operadores de redirección bastaría con cambiar los filedescriptors asociados a los fileid's de `stdin`, `stdout` y `stderr`. Esto se puede hacer con la llamada al sistema `dup`, que **duplica** el filedescriptor indicado en su argumento, en el menor fileid libre. Veamos un ejemplo que redirige la salida a un archivo. Notar que las llamadas al sistema para manejo de archivos ~~que~~ no son las habituales de la libc (`fopen`, `fclose`, ...).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define STDOUT_FID 1

int
main (int argc, char **argv)
{
        int fid;
        int flags,perm;

        flags = O_WRONLY|O_CREAT|O_TRUNC;
        perm =  S_IWUSR|S_IRUSR;
        fid = open("redireccion.out", flags, perm);
        if (fid<0) { perror("open");exit(1); }
        close(STDOUT_FID); /* stdout def. en stdio.h es un FILE* */
        if (dup(fid)<0) { perror("dup"); exit(1); }

        printf("Hello World!\n");
        close(fid);

        return 0;
}
```

Para implementar el operador de tubería '|', debemos utilizar la syscall `pipe`, la cual crea en el espacio del filesystem un nuevo nodo de este tipo y devuelve dos fileid's que serán la salida y la entrada de la tubería. Veamos un sencillo ejemplo de IPC.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int
main (int argc, char **argv)
{
        int fork_id;
        int srv_client_pipe[2];
        int client_srv_pipe[2];

        if (pipe(srv_client_pipe)<0) { perror("srv_client_pipe");exit(1); }
        if (pipe(client_srv_pipe)<0) { perror("client_srv_pipe");exit(1); }
        if ((fork_id = fork())<0) { perror("fork");exit(1); };

        if (fork_id!=0) { /* soy padre, soy cliente */
                int status;
                int x,fx;
                x = 8;
                printf("CLIENTE: envia %d\n",x);
                write(client_srv_pipe[1], &x, sizeof(int));
                read(srv_client_pipe[0], &fx, sizeof(int));
                printf("CLIENTE: recibe %d\n",fx);
                wait(&status);
                close(srv_client_pipe[0]);
                close(srv_client_pipe[1]);
                close(client_srv_pipe[0]);
                close(client_srv_pipe[1]);
        }
        else { /* soy hijo, soy server */
                int in,out;
                read(client_srv_pipe[0], &in, sizeof(int));
                printf("SERVIDOR: recibe %d\n",in);
                out = in*in;
                printf("SERVIDOR: envia %d\n",out);
                write(srv_client_pipe[1], &out, sizeof(int));
        }

        return 0;
}
```

Notar que este programa, aunque es concurrente, presenta sólo un interleaving en su salida. Se dice que el programa está *fuertemente sincronizado* debido a las primitivas de comunicación.

Tips

El problema de la búsqueda en la secuencia PATH es un simple problema de búsqueda de secuencia, que con un buen tipo abstracto de dato (TAD) de secuencia implementado sobre listas se debería resolver el problema de manera

compacta.

Planteen el problema de parseo de forma general, ya saben todas las posibilidades que hay para parsear, asi que no debería ser difícil realizar una funcion `parse_command()` que devuelva una estructura command que contega toda la información necesaria para ejecutar el o los comandos, y aplicar los operadores.
Si es lo suficientemente general, los demás puntos saldrán sin esfuerzo adicional.

El shell refleja tan bien la estructura interna de las llamadas al sistema de los UNIX para creación de procesos, manejo de corrientes e IPC, como se refleja en "*The UNIX Time-Sharing System*" en su sección VI, que el programa principal debería ser **muy compacto**. Si no logran esto reorganicen sus ideas intentando ser lo más económicos posible. Se recomienda fuertemente leer este artículo.

El libro "Advanced Linux Programming", muestra le uso del `assert()`, y como manejar correctamente los errores que pueden devolver las llamadas a sistema. Estudien este material antes de comenzar la codificación. Los ejemplos de código contienen un poco de *programación defensiva*.

Para ver cuanto se sigue la programación defensiva en la práctica usual de desarrollo de software vayan a freshmeat.net, busquen proyectos cualquiera en C, con fuentes disponibles, bajen la última versión, y encuentren en los fuentes una syscall de la cual se ignora el resultado. Miren el código de OpenSSH. Qué notan? Este tip está tomando la forma de ejercicio obligatorio y debe comentarse en la documentación del laboratorio.

Material adicional:

- "Pipes, Redirection, and Filters" del libro "*The Art of Unix Programming*".
- Capítulos 3 (Process) y 5 (Interprocess Communication) del libro "*Advanced Linux Programming*".
- Capítulo "Implementing a Job Control Shell", de "*The GNU C Library Reference Manual*"

Tareas Adicionales

Si les sobra tiempo pueden hacer las siguientes mejoras:

- Posibilidad de mezclar los 4 operadores "&<>|", a lo más una vez cada uno, en la misma línea de comandos.
- Manejar bien los procesos lanzados en background para que no se generen *zombies*. Pueden consultar la sección 3.4.3 de "Advanced Linux Programming".
- Prompt configurable desde la variable de entorno.