Introducción a Sistemas Operativos: Comunicación entre Procesos

Clips xxx
Francisco J Ballesteros

1. Pipefroms

Otra función de utilidad realiza el trabajo inverso, permite leer la salida de un comando externo en un programa escrito en C. Este podría ser el código de ficha función.

```
int
pipefrom(char* cmd)
{
    int fd[2];

    if (pipe(fd) < 0) {
        return -1;
    }
    switch(fork()){
    case -1:
        return -1;
    case 0:
        close(fd[0]);
        dup2(fd[1], 1);
        close(fd[1]);
        execl("/bin/sh", "sh", "-c", cmd, NULL);
        err(1, "execl");
    default:
        close(fd[1]);
        return fd[0];
    }
}
```

En este caso redirigimos la salida estándar del nuevo proceso al pipe (en el proceso hijo) y retornamos el descriptor para leer del pipe.

Podemos utilizar esta función en un programa para leer la salida de un comando. Por ejemplo, este programa ejecuta el comando `who` que muestra qué usuarios están utilizando el sistema y lee su salida. En nuestro caso nos limitamos a escribir toda la salida del comando en nuestra salida estándar. En un caso real podríamos procesar la salida de `who` para hacer con ella algo más interesante.

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

int
pipefrom(char* cmd)
{
    // como se muestra arriba
}

static int
readall(int fd, char buf[], int nbuf)
{
    int tot, nr;

    for (tot = 0; tot < nbuf; tot += nr) {
        nr = read(fd, buf+tot, nbuf-tot);
        if (nr < 0) {
            return -1;
        }
        if (nr == 0) {
            break;
        }
    }
    return tot;
}

int
main(int argc, char* argv[])
{
    int    fd, nr;
    char    buf[1024];

    fd = pipefrom("who");
    if (fd < 0)
        err(1, "pipefrom");
    nr = readall(fd, buf, sizeof buf - 1);
    close(fd);
    if (nr < 0) {
        err(1, "read");
    }
    buf[nr] = 0;
    printf("command output:\n%s--\n", buf);
    exit(0);
}
```

El programa incluye una función auxiliar, `readall`, que lee todo el contenido desde el descriptor indicado dejándolo en un buffer. Esta podría ser una ejecuciæon del programa:

```
unix$ pwho
nemo     console  Jul 13 07:30
nemo     ttys000  Jul 13 07:31
nemo     ttys001  Aug 18 15:59
nemo     ttys002  Aug 20 18:48
unix$
```

2. Sustitución de comandos

El shell incluye sintaxis que realiza un trabajo similar al código que acabamos de ver. Se trata de la la llamada *sustitución de comandos* o *interpolación de salida* de comandos. La idea es utilizar un comando *dentro* de una línea de comandos para que dicho comando escriba el texto que debiéramos escribir nosotros en caso contrario.

Por ejemplo, podríamos querer guardar en una variable de entorno la fecha actual. Para ello deberíamos ejecutar `date` y luego escribir una línea de comandos que asigne a una variable de entorno lo que `date` ha escrito. Pero, por un lado, esto no es práctico y, por otro, si queremos ejecutar estos comandos como parte de un script, no queremos que un humano tenga que editar el script cada vez que lo ejecutamos.

Por ejemplo, tal vez queremos hacer un script llamado `mkvers` que genere un fichero fuente en C que declare un array que defina la versión del programa con la fecha actual.

Veamos cómo hacerlo. Queremos tener un fichero que contenga, por ejemplo,

```
char vers[] = "vers: Fri Aug 26 17:05:14 CEST 2016";
```

para utilizarlo junto con otros ficheros al construir nuestro ejecutable. El plan es que cada vez que hagamos un cambio significativo, ejecutaremos

```
unix$ mkvers
```

y luego compilaremos el programa. De ese modo el programa podría imprimir (si así se le pide) la fecha que corresponde a la versión del programa.

Este podría ser el script:

```
#!/bin/sh
DATE=‘date‘

(
    echo -n ’char vers[] = "vers: ’
    echo -n "$DATE"
    echo  ’";’
) > vers.c
```

La linea que nos interesa es

```
DATE=‘date‘
```

que es similar a haber escrito

```
DATE="Fri Aug 26 17:12:10 CEST 2016"
```

si es que esa era la fecha.

Lo que hace el shell cuando una línea de comandos contiene "`‘...‘`" es ejecutar el comando que hay entre "`‘...‘`" y sustituir ese texto de la línea de comandos por la salida de dicho comando. También se dice que el

shell interpola la salida de dicho comando en la línea de comandos. Para conseguir eso, el shell ejecuta código similar a `pipefrom()` y lee la salida del comando. Una vez la ha leído por completo, la utiliza como parte de la línea de comandos y continúa ejecutándola.

Por ejemplo, el comando *seq(1)* se utiliza para contar. Resulta muy útil para numerar cosas. Por ejemplo,

```
unix$ seq 4
1
2
3
4
unix$
```

Como puedes ver en la salida de este comando

```
unix$ echo x ‘seq 4‘ y
x 1 2 3 4 y
unix$
```

el shell ha ejecutado `echo` con "`1 2 3 4`" en la línea de comandos: ha reemplazado el comando entre comillas invertidas por la salida de dicho comando. ¡Y ha reemplazado los "`\n`" en dicha salida por espacios en blanco! Piensa que es una *línea* de comandos y no queremos fines de línea en ella.

Es *muy* habitual utilizar la sustitución de comandos, sobre todo a la hora de programar scripts. Por ejemplo, este libro tenía inicialmente ficheros llamados `ch1.w` para el primer capítulo, `ch2.w` para el segundo, etc.

Podemos crear estos ficheros utilizando un bucle `for` en el shell y la sustitución de comandos. Lo primero que haremos será guardar en una variable los números para los 6 capítulos que pensábamos escribir inicialmente.

```
unix$ caps=‘seq 6‘
unix$ echo $caps
1 2 3 4 5 6
unix$
```

Y ahora podemos ejecutar el siguiente comando que crea cada uno de los ficheros:

```
unix$ for c in $caps
> do
>     echo creating chap $c
>     touch ch$c.w
> done
creating chap 1
creating chap 2
creating chap 3
creating chap 4
creating chap 5
creating chap 6
unix$ echo ch*.w
ch1.w ch2.w ch3.w ch4.w ch5.w ch6.w
unix$
```

Como puedes ver, el comando `for` es de nuevo parte de la sintaxis del shell, similar al comando `if` que vimos anteriormente. En este caso, este comando ejecuta las líneas de comandos entre `do` y `done` (el cuerpo del bucle) para cada uno de los valores que siguen a `in`, haciendo que la variable de entorno cuyo

nombre precede a `in` contenga el valor correspondiente en cada iteración. Si no has entendido este párrafo, mira la salida del comando anterior y fíjate en qué es `$c` en cada iteración.

Podríamos haber escrito este otro comando

```
unix$ for c in ‘seq 6‘ ; do
>     echo creating chap $c
>     touch ch$c.w
> done
```

y el efecto habría sido el mismo.

Poco a poco vemos que podemos utilizar el shell para combinar programas existentes para hacer nuestro trabajo. ¡Eso es UNIX!. Recuerda...

- La primera opción es utilizar el manual y encontrar un programa ya hecho que puede hacer lo que deseamos hacer
- La segunda mejor opción es combinar programas existentes utilizando el shell para ello
- La última opción es escribir un programa en C para hacer el trabajo.