Introducción a Sistemas Operativos: Procesos

Clips xxx
Francisco J Ballesteros

1. Variables de entorno

Otra forma de darle información a un proceso (además de usando sus argumentos) es utilizar las llamadas **variables de entorno**. Cada proceso tiene un array de strings de tal forma que cada string tiene el aspecto

"*nombre*=*valor*"

y define una variable de entorno con nombre "*nombre*" y valor "*valor*".

Cuando UNIX inicializa la memoria para un nuevo programa se ocupa de que dicho array esté inicializado con las variables de entorno que se han indicado en la llamada al sistema utilizada para ejecutar un nuevo programa. Además de `argv`, ahora tienes que considerar que tienes un array de variables de entorno.

El propósito de estas variables es definir elementos que se desea que estén disponibles para los programas que ejecutas sin tener que pasar dichos elementos como argumento todas las veces. Naturalmente, tanto el nombre de las variables como su valor son strings, dado que esta abstracción (variables de entorno) consiste en el array de strings que hemos descrito.

Podemos definir variables de entorno utilizando el shell. Por ejemplo:

```
unix$ v=33
```

Cuando el shell ve que una palabra en una línea de comandos comienza por un "$", entiende que se desea utilizar el valor de la variable de entorno cuyo nombre sigue y cambia dicha palabra por el valor de la variable (¡que siempre es un string!).

Por ejemplo, ya conoces *echo(1)* y sabes que simplemente escribe sus argumentos tal cual los recibe en `argv`, sin ningún tipo de magia. Pues bien, mira lo que escribe *echo*:

```
unix$ v=33
unix$ echo $v
33
unix$
```

Te resultará útil definir variables para nombres que utilices a menudo. Por ejemplo, si sueles cambiar de directorio a un directorio dado, puedes utilizar una variable de entorno para escribir menos:

```
unix$ d=/un/path/muy/largo
unix$ cd $d
unix$
```

Es tan habitual utilizar el directorio *home* de un usuario, que cuando haces un *login* se define una variable de entorno `HOME` que contiene el path de tu directorio *home*. Así pues:

```
unix$ cd $HOME
unix$ pwd
/home/nemo
unix$ cd
unix$ pwd
/home/nemo
```

El shell expande variables de entorno (reemplaza `$x` por el valor de la variable `x`) en cualquier sitio de la línea de comandos. Fíjate en esto:

```
unix$ p=pwd
unix$ $p
/home/nemo
unix$
```

Pero, si intentamos utilizar este truco para ejecutar "`ls -l`" nos llevamos una sorpresa...

```
unix$ l=ls -l
sh: -l: command not found
```

Necesitamos escribir *una sóla palabra* tras el operador "=". Esta es la sintaxis del shell para definir variables. Como hay un espacio entre "`ls`" y "`-l`", el shell ha intentado ejecutar "`-l`" como uno comando.

¿Y si probamos...?

```
unix$ l=ls-l
unix$
```

¡Ha funcionado!, ¿O no?

```
unix$ $l
sh: ls-l: command not found
unix$
```

¿Comprendes lo que ha sucedido? Seguro que sí, y, si no es así, piensa... ¿Cuál es el nombre del comando que has ejecutado?

La solución a nuestro problema es utilizar sintaxis de shell para indicarle que cierto texto que escribimos en la línea de comandos ha de entenderse como una sóla palabra. Veámoslo:

```
unix$ l='ls -l'
unix$
```

Y ahora podemos...

```
unix$ $l
total 240
drwxr-xr-x   2 nemo  staff   1156 Jul 25 12:36 zxbib
drwxr-xr-x   5 nemo  staff   1734 Jul 25 13:01 zxdoc
...
```

Vamos a utilizar nuestro programa `eco2` para ver qué argumentos pasa el shell cuando utilizamos las comillas.

```
unix$ eco2 -v 'ls -l' ls -l
[ls -l] [ls] [-l]
```

Lo escrito entre comillas simples está en un sólo string en el vector `argv` de `eco2`. Esto es, el shell ha

tomado literalmente lo que hay entre comillas simples como una sóla palabra. Ya lo sabías si recuerdas el primer tema de este curso.

También podemos utilizar comillas dobles para hacer que el shell tome su contenido como una sóla palabra. La diferencia entre estas y las simples radica en que el shell, en el caso de las comillas dobles, expande variables de entorno dentro de ellas. Por ejemplo:

```
unix$ cmd=ls
unix$ arg=-l
unix$ line="$cmd $arg"
unix$ $line
total 240
drwxr-xr-x   2 nemo  staff   1156 Jul 25 12:36 zxbib
drwxr-xr-x   5 nemo  staff   1734 Jul 25 13:01 zxdoc
...
```

En cambio...

```
unix$ cmd=ls
unix$ arg=-l
unix$ line='$cmd $arg'
unix$ $line
sh: $cmd: command not found
unix$ echo $line
$cmd $arg
```

En un programa en C podemos consultar variables de entorno llamando a *getenv(3)*. Por ejemplo, este programa cambia su directorio actual al directorio casa e imprime el path de dicho directorio antes de continuar con su trabajo.

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

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

    home = getenv("HOME");
    if (home == NULL){
        fprintf(stderr, "we are homeless\n");
        exit(1);
    }
    if (chdir(home) < 0) {
        err(1, "can't cd to home");
    }
    printf("working at home: %s\n", home);

    // ... do some other things...

    exit(0);
}
```

Si lo tenemos compilado en `env` y lo ejecutamos, podemos ver lo que hace...

```
unix$ env
working at home: /home/nemo
unix$
```

Si la variable no está definida, `getenv` devuelve `NULL`. Esto no es un error, y `errno` no se actualizará en tal caso. Simplemente la variable puede no estar definida. O dicho de otro modo... puede que ningún string en tu vector de variables de entorno comience por "`HOME=`" (aunque no es lo que uno espera en UNIX para la variable `HOME`).

¿Qué haría el programa si lo cambiamos para que ejecute

```
home = getenv("$HOME");
```

en lugar de hacer que hacía antes? ¡Pruébalo! ¿Puedes ver por qué? ¿Cómo se llama la variable? Recuerda que "`$HOME`" es sintaxis del shell. ¿Ejecuta el shell en algún caso cuando tu programa en C ejecuta la línea anterior? ¡Desde luego que no!

¿Y qué haría si lo cambiamos para que utilice

```
if (chdir("$HOME") < 0) {
    err(1, "can't cd to home");
}
```

o para que utilice

```
if (chdir("HOME") < 0) {
    err(1, "can't cd to home");
}
```

en lugar de lo que teníamos antes?

Para definir una variable de entorno en el proceso en que ejecutamos un programa en C podemos utilizar la función *putenv(3)* o *setenv(3)*. Ambas están descritas en *getenv(3)*, así que si has seguido con la buena costumbre de leer la página de manual de cada llamada la primera vez que la usas, ya las conoces. Esto es un ejemplo en cualquier caso:

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

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

    putenv("tempdir=/tmp");
    temp = getenv("tempdir");
    if (temp == NULL){
        fprintf(stderr, "no temp dir\n");
        exit(1);
    }
    printf("temp dir is: %s\n", temp);

    exit(0);
}
```

Y esta es su ejecución:

```
unix$ penv
temp dir is: /tmp
```

Naturalmente, sería absurdo utilizar la variable de entorno `tempdir` como lo hemos hecho si no pensamos ejecutar nuevos procesos desde nuestro programa en C. Si tan sólo queremos definir una variable para guardar el path de un directorio temporal, C ya tiene variables.

Hasta ahora todo bien. Pero... ¿Puedes explicar esto?

```
unix$ penv
temp dir is: /tmp
unix$ echo $tempdir
unix$
```

Resulta que tras ejecutar nuestro programa en C que define una variable de entorno, ¡el shell no la tiene definida! Pero esto es normal. Piensa que la variable la define el proceso que ejecuta el programa en C, y que el shell es otro proceso. Cada proceso tiene sus propias variables de entorno. ¿Lo ves normal ahora?

Cuando no entiendas algo, piensa que no hay magia y piensa en qué programas intervienen en lo que estás haciendo y qué hace cada uno paso a paso. Si sigues sin poder explicar el resultado es que te estás perdiendo algo respecto a lo que hace cada programa. Recurre al manual y haz programas de prueba que intenten explicar tus hipótesis respecto a qué está pasando.

En relación con esto, resulta interesante mirar lo que sucede en este otro caso:

```
unix$ x=foo
unix$ echo $x
foo
unix$ sh
unix$ echo $x

unix$ exit
unix$ echo $x
foo
unix$
```

Hemos ejecutado un shell (ejecutando el comando `sh`) y dicho shell no tiene definida la variable `x` (`$x` no tiene nada). Cuando terminamos dicho shell con `exit` volvemos al shell original y ahí si que tenemos la variable de entorno definida.

Es como si las variables que defines fuesen locales al shell y los comandos que ejecutamos desde ese shell ya no las tienen. Vamos a comprobarlo con un programa en C:

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

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

    x = getenv("x");
    if (x == NULL){
        fprintf(stderr, "no x\n");
        exit(1);
    }
    printf("%s\n", x);

    exit(0);
}
```

Si lo compilamos y dejamos el ejecutable en `penvx` podemos ver lo que sucede:

```
unix$ x=foo
unix$ penvx
no x
unix$
```

Efectivamente, el shell no ha pedido a UNIX que ejecute `penvx` con la variable `x` definida en el entorno. Pero hay formas de hacer que lo haga:

```
unix$ export x
unix$ penvx
foo
unix$
```

El comando `export` es un comando construido dentro del shell. Es una *primitiva* o *builtin* del shell. Su propósito es indicarle al shell que *exporte* una o más variables de entorno a los procesos que ejecute dicho shell para ejecutar nuevos comandos.

Hoy día suele ser habitual hacer ambas cosas a la vez:

```
unix$ export x=foo
unix$ penvx
foo
unix$
```