Introducción a Sistemas Operativos: Procesos

Clips xxx
Francisco J Ballesteros

1. Nacimiento

A los subprogramas se los llama y retornan, pero si consideramos un programa que queremos ejecutar en un proceso, no hay *llamadas* a programas ni dichos programas *retornan*. Un proceso simplemente termina cuando le pide al sistema terminar o cuando se comporta mal (por ej., intenta leer una dirección de memoria que no está en ningún segmento) y el sistema lo mata. No obstante, como sabes, cuando ejecutamos comandos en el shell podemos indicar argumentos para que sus programas los procesen y para controlar lo que hacen.

Cuando el shell el pide al sistema que ejecute un programa, una vez que dicho programa está cargado en memoria, el sistema suministra un flujo de control para que ejecute. En realidad sabes que lo habitual es que *no* se cargue todo el programa en memoria y, en cambio, se pagina en demanda aquellas páginas de memoria a las que accede el programa conforme ejecuta. Pues bien, el flujo de control supone valores para los registros del procesador, inicializados para que comiencen la ejecución del programa que se desea ejecutar e incluyendo un contador de programa y un puntero de pila. La pila inicialmente estará prácticamente vacía salvo por los argumentos del programa principal. Cuando compilamos un programa en C, el enlazador se ocupa de indicar en el ejecutable que la dirección de `main` es la dirección en la que hay que empezar a ejecutar código. Es por eso que los programas en C comienzan ejecutando `main`. Los argumentos de `main` son un array de strings (en `argv`) y el número de strings que hay en dicho array (en `argc`).

Teniendo esto en cuenta, el siguiente programa escribe sus argumentos indicando antes de cada uno el índice en `argv`:

```
#include <stdlib.h>
#include <stdio.h>
int
main(int argc, char* argv[])
{
    int i;

    for(i = 0; i < argc; i++) {
        printf("%d\t%s\n", i, argv[i]);
    }
    exit(0);
}
```

Si lo guardamos en el fichero `eco.c` y lo compilamos y lo ejecutamos podemos ver qué argumentos recibe el programa para una línea de comandos dada:

```
unix$ cc eco.c
unix$ ./eco un programa sencillo
0:    ./eco
1:    un
2:    programa
3:    sencillo
unix$
```

Como verás, ¡el primer "argumento" en `argv` es en realidad el nombre del programa! Concretamente, es el nombre del programa tal y como lo hemos escrito en la línea de comandos del shell. Recuerda que "`./eco`" es un camino o path relativo y que "`.`" es el directorio actual. Así pues, `argv[0]` contiene el nombre del programa. Los siguientes strings en `argv` son los argumentos que hemos dado a `eco` en la línea de comandos. Es muy útil emplear `argv[0]` en mensajes de error para identificar ante el usuario al programa que tiene problemas.

Lo que ha sucedido es que el shell ha leído la línea de comandos que hemos escrito, y ha utilizado la primera palabra para localizar el fichero que contiene el programa que queremos ejecutar. Después, le ha pedido a UNIX que lo ejecute y que haga que `main` en dicho programa reciba una copia del array de strings que corresponde a la línea de comandos.

Ya conocemos *echo(1)*. Recuerda que dicho comando acepta la opción "`-n`" para suprimir la impresión del fin de línea tras imprimir sus argumentos. Podemos cambiar el programa anterior para que sea como *echo(1)*, y podemos hacer que la opción "`-v`" imprima cada argumento entre corchetes, para distinguirlos mejor. Este es el programa:

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

static char *argv0;

static void
usage(void)
{
    fprintf(stderr, "usage: %s [-nv] [args...]\n", argv0);
    exit(1);
}
```

```
int
main(int argc, char* argv[])
{
    int i, nflag, vflag;
    char *arg, *sep;

    nflag = vflag = 0;
    argv0 = argv[0];
    argv++;
    argc--;
    while(argc > 0 && argv[0][0] == '-') {
        if (strcmp(argv[0], "--") == 0) {
            argv++;
            argc--;
            break;
        }
        for(arg = argv[0]+1; *arg != 0; arg++) {
            switch(*arg) {
            case 'n':
                nflag = 1;
                break;
            case 'v':
                vflag = 1;
                break;
            default:
                usage();
            }
        }
        argv++;
        argc--;
    }
    sep = "";
    for(i = 0; i < argc; i++){
        if (vflag) {
            printf("%s[%s]", sep, argv[i]);
        } else {
            printf("%s%s", sep, argv[i]);
        }
        sep = " ";
    }
    if (!nflag) {
        printf("\n");
    }
    exit(0);
}
```

Primero guardamos en la global `argv0` el nombre del programa. Esto lo usamos en la función `usage` para imprimir un recordatorio del uso del programa empleando el nombre tal y como lo hemos recibido en `argv[0]`. Una vez hecho esto, quitamos del vector del argumentos el primer string haciendo que `argv` apunte al siguiente elemento (y actualizando `argc` para que refleje el número de strings en `argv`).

El bucle `while` procesa las opciones (argumentos que empiezan por "`-`") para detectar si se ha indicado "`-n`" o "`-v`". Como verás, el programa también entiende dichas opciones si se suministra "`-nv`" o "`-vn`", como suelen hacer los programas en UNIX. Además, un argumento "`--`" indica el fin de las opciones, para

hacer que el resto de argumentos no se procesen como opciones. Esto es lo habitual en UNIX. Por ejemplo, podemos ejecutar el programa como sigue, si lo tenemos compilado y enlazado en el ejecutable "`eco2`".

```
unix$ eco2 -v -- -n hola
[-n] [hola]
unix$
```

Hemos optado por procesar los argumentos de tal forma que una vez procesadas todas las opciones, `argc` y `argv` contienen el resto de argumentos. Esto es cómodo en general.

Si llamamos al programa de forma incorrecta, obtenemos un recordatorio de cómo hay que usarlo:

```
unix$ eco2 -vx hola
usage: eco2 [-nv] [args...]
unix$
```

2. Muerte

Habrás notado que "`main`" termina llamando a `exit`. Esta llamada al sistema pide a UNIX que termine la ejecución del proceso (y por tanto de su programa). El entero que pasamos como argumento se espera que sea cero si el programa ha conseguido hacer su trabajo y distinto de cero en caso contrario. A este valor se le suele llamar **exit status** (en inglés). Todo esto es muy importante. Habitualmente es un programa el que ejecuta otros programas (por ejemplo el shell). Si dichos programas no informan correctamente respecto a si pudieron hacer su trabajo o no, el programa que los ejecuta podría hacer cosas que no esperamos.

Si un programa no comprueba si sus argumentos son correctos y/o no suministra el estatus de salida adecuado (cero o distinto de cero), no está correctamente programado y es posible que no pueda usarse en la práctica.

Habrás visto que `main` retorna un entero. Dado que *exit(3)* termina la ejecución, no retornamos nada en nuestro programa. No obstante, retornar de `main` hace que el valor retornado se utilice para llamar a *exit(3)* con dicho valor. Esto es, podríamos haber terminado nuestro programa escribiendo

```
int
main(int argc, char* argv[])
{
    ...
    return 0;
}
```

No hay magia en esto. En realidad el programa principal es una función que hace algo equivalente a

```
exit(main(argc, argv));
```

lo que explica que podamos retornar de `main`.

Podemos utilizar el shell para ver que tal le fue a un comando que ejecutamos anteriormente:

```
unix$ ls /blah
ls: /blah: No such file or directory
unix$ echo $?
1
unix$ echo $?
0
```

Aquí, "`$?`" es una *variable de entorno* (más adelante veremos lo que es esto) que contiene como valor un string correspondiente al estatus de salida del último comando ejecutado. ¿Puedes decir por qué la segunda vez el estatus es 0?

3. En la salida

En ocasiones resulta útil ejecutar código cuando un programa termina. Aunque no se debe abusar de este mecanismo, puede resultar útil para, por ejemplo, borrar ficheros temporales o realizar alguna otra tarea incluso si una función decide llamar a `exit`.

La función *atexit(3)* permite instalar punteros a función de tal forma que dichas funciones ejecuten durante la llamada a *exit(3)*. En realidad, `exit` llama las funciones que ha instalado `atexit` y luego llama a `_exit`, que realmente termina la ejecución del proceso.

Recuerda que el programa principal es el realidad

```
exit(main(argc, argv));
```

La función de librería `exit` se parece a...

```
void
exit(int sts)
{
    int i;
    for (i = 0; i < numexitfns; i++) {
        exitfns[i]();
    }
    _exit(sts);
}
```

Para ejecutar algo cuando el programa termine llamando a *exit*, podemos llamar a *atexit* como en el siguiente programa:

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

static void
exitfn1(void)
{
    puts("dentro de exitfn1");
}

static void
exitfn2(void)
{
    puts("dentro de exitfn2");
}
```

```
int
main(int argc, char* argv[])
{
    atexit(exitfn1);
    atexit(exitfn2);
    puts("a punto de salir...");
    return 0;
}
```

Cuando lo ejecutemos, suponiendo que el ejecutable es `atexit`, veremos algo como...

```
unix$ atexit
a punto de salir...
dentro de exitfn2
dentro de exitfn1
```

Como verás, el kernel de UNIX no sabe nada de nada de tus *atexit*. Simplemente la función de librería *exit(3)* se ocupa de hacer los honores. ¿Qué pasaría si llamas a *exit* desde una función instalada con *atexit*? ¡Pruébalo! ¿Entiendes por qué?