Programación C: errores, parámetros, funciones y entorno

C Prácticas

Tabla de contenido

El entorno de desarrollo
- Ejemplo c001.c: Programa Hola Mundo
- Ejemplo c002.c: Línea de comandos

Llamadas al sistema

Resultado de las llamadas al sistema
- Ejemplo c003.c: Tratamiento del posible error en la ejecución de la llamada a fork().

La variable global errno
- Ejemplo c004.c: Mostar errores con errno, perror() y strerror(errno)
- Ejemplo c008.c: Uso de las constantes de error

Variables de entorno
Ejemplo c009.c: Variables de entorno vía parámetros de main
Ejemplo c010.c: Variables de entorno vía variable extern environ
Ejemplo c011.c: Consulta de una variable de entorno
Ejemplo c012.c: Consulta de las rutas de la variable PATH
Ejemplo c013.c: Uso de getenv() para obtener una variable de entorno

El entorno de desarrollo

Utilizaremos el compilador GNU C Compiler. http://gcc.gnu.org/

Un tutorial en castellano muy sencillo se puede encontrar en http://iie.fing.edu.uy/~vagonbar/gcc-make/gcc.htm

Brevemente describiremos los pasos básicos para poder compilar los ejercicios y ejemplos.

El comando gcc es el interfaz de usuario (front-end) del compilador de C GNU.

El siguiente comando compila un programa llamado "**uno.c**". El fichero ejecutable será "**uno**"

```
$gcc uno.c -o uno
```

La opción -o especifica el nombre del fichero ejecutable. Si no se especifica, el nombre del fichero ejecutable generado es "**a.out**".

Si tenemos varios ficheros fuente que forman parte de un único programa ejecutable podemos compilarlos de uno en uno de forma separada (opción -c del compilador) y linkarlos todos al final con el comando "**ld**" (linkador).

Otra opción es hacerlo todo de golpe especificando la lista de ficheros en el compilador. Por ejemplo, si tenemos tres ficheros llamados: **uno.c**, **dos.c** y **tres.c**, y queremos obtener un ejecutable denominado "**prog**", ejecutaríamos el siguiente comando:

```
$gcc uno.c dos.c tres.c -o prog
```

Ejemplo c001.c: Programa Hola Mundo

```c
// ============================
// Programa #1: Hola Mundo !!
// Archivo: holamundo.c
// ============================
#include <stdio.h>
int main(void) {
printf("Hola mundo\n");
return 0
}
```

Lo compilaríamos con:

```
$ gcc holamundo.c -o holamundo
```

Ejemplo c002.c: Línea de comandos

Muestra cómo se gestionan los parámetros pasados al programa desde la línea de comandos.

```c
// ======================================================
// Programa #2: Paso de parámetros desde la línea de órdenes
// Archivo: ejemplo2.c
// ======================================================
#include <stdio.h>
int main (int argn,char *argv[])
{
int i;
printf("Número de argumentos: %d\n",argn);
printf("Nombre del programa : %s\n",argv[0]);
// Mostramos el resto de parámetros
for(i=1;i<argn;++i) {
   printf("Argumento %d = %s\n",i,argv[i] );
   }
return 0;
}
```

Llamadas al sistema

Un programa de usuario nunca podrá tomar el control del sistema en modo supervisor; de no ser así, la ejecución de un programa podría afectar a otros. ¿Cómo puede un proceso realizar entonces una operación E/S?. Con las “llamadas al sistema”.

Una “llamada al sistema” es el método que utiliza un proceso para solicitar al sistema operativo que realice una acción. Los procesos realizan llamadas a las funciones de la librería, las cuales generan una interrupción software.

Una “llamada al sistema” es el equivalente software de las interrupciones hardware. Lo que sucede al realizar una llamada al sistema (petición de servicio) es:

1. Se genera una interrupción, para avisar al sistema operativo
2. El control pasa entonces a una rutina asociada del sistema operativo y se pasa a modo supervisor (se cambia el bit de modo).
3. El núcleo del sistema operativo comprueba que los parámetros son correctos.
4. Si es así, ejecuta la rutina y cuando termine se pasa de nuevo a modo usuario y se retorna el control a la siguiente instrucción del programa de usuario detrás de la llamada al sistema.
5. Si son incorrectos, el sistema toma las acciones oportunas.

Existe una librería de funciones (API) para poder realizar las llamadas desde C.

Estas forman el interfaz **POSIX**. Tendremos que incluir el fichero “**unistd.h**”, donde están definidos sus prototipos. Puesto que es una librería del sistema, utilizaremos los símbolos “<“, “>”.

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

Normalmente, cuando se produce algún error, las rutinas del API retornan el valor -1.

En total disponemos de unas 41 llamadas al sistema, que describimos a continuación agrupadas por sus funciones.

Las funciones del sistema trabajan sobre las abstracciones de proceso, fichero y tiempo.

Proceso:

1. Gestión de procesos.
2. Señales.

Fichero:

- Gestión de ficheros
- Gestión de directorio y el sistema de ficheros

Tiempo:

- Gestión de tiempo

Resultado de las llamadas al sistema

Las llamadas al sistema, normalmente retornan -1 en caso de fallo.

Es muy importante realizar siempre esa comprobación para conseguir programas robustos.

Ejemplo c003.c: Tratamiento del posible error en la ejecución de la llamada a fork().

¿Por qué sale dos veces el mensaje?

```
// ========================================
// Programa #3: Llamada al sistema con error
// Archivo: ejemplo3.
// ========================================
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argn, char *argv[]) {
if ( fork() == -1 ) {
   printf ("La llamada a fork ha fallado ...\n");
   exit (1); // o return 1;
   }
else {
   printf ("La llamada a fork se ha realizado con exito...\n");
   exit(0); // o return 0;
   }
}
```

En caso de error en una llamada al sistema, se actualiza automáticamente una variable global llamada "**errno**" que nos indica un código con el motivo de dicho error. Para más información sobre esta variable consultar el siguiente punto: "La variable

global: errno".

La variable global errno

Se define en el fichero **<sys/errno.h>**, el cual deberemos incluir si queremos acceder a ella.

```
#include <errno.h>
```

Aunque no es necesario conocer como se define, el sistema lo hace así:

```
extern int * __error();
#define errno (* __error())
```

La función __error() retorna un puntero a un campo de la estructura que define los hilos (threads) hijos del hilo inicial. Para el hilo inicial y procesos que no tienen hilos (procesos pesados), retorna un puntero a la variable **errno** global. Observando el #define anterior llegamos a la conclusión de que cuando nosotros escribimos "**errno**" en nuestro código, realmente estamos obteniendo un puntero a un entero, pero esto será transparente para nosotros.

Cuando una llamada al sistema genera un error, retornará un valor negativo (normalmente -1) y establece la variable errno con un valor numérico que indica el motivo.

Este valor permanecerá hasta que otra llamada al sistema provoque otro error, es decir, las llamadas al sistema que no provocan error nunca establecerán el valor de la variable **errno**. Por esto hay que llevar cuidado y consultar el valor de **errno justo después de haberse producido el error**.

La siguiente tabla nos muestra una lista con los códigos de error, los cuales están definidos en el fichero <…/errno.h>. Los puntos suspensivos indican que este fichero estará ubicado en un directorio diferente en cada instalación.

La función perror()

Aparte, existe la función **perror()**, que da un mensaje (en inglés) sobre el error producido.

La numeración de la tabla siguiente, puede variar de sistema en sistema, por lo que se recomienda se utilicen exclusivamente los nombres de las variables simbólicas y no la numeración de dicha tabla.

```
#ifndef _I386_ERRNO_H
#define _I386_ERRNO_H

#define EPERM            1      /* Operation not permitted */
#define ENOENT           2      /* No such file or directory */
#define ESRCH            3      /* No such process */
#define EINTR            4      /* Interrupted system call */
#define EIO              5      /* I/O error */
#define ENXIO            6      /* No such device or address */
#define E2BIG            7      /* Argument list too long */
#define ENOEXEC          8      /* Exec format error */
#define EBADF            9      /* Bad file number */
#define ECHILD          10      /* No child processes */
#define EAGAIN          11      /* Try again */
#define ENOMEM          12      /* Out of memory */
#define EACCES          13      /* Permission denied */
#define EFAULT          14      /* Bad address */
#define ENOTBLK         15      /* Block device required */
#define EBUSY           16      /* Device or resource busy */
#define EEXIST          17      /* File exists */
#define EXDEV           18      /* Cross-device link */
#define ENODEV          19      /* No such device */
#define ENOTDIR         20      /* Not a directory */
#define EISDIR          21      /* Is a directory */
#define EINVAL          22      /* Invalid argument */
#define ENFILE          23      /* File table overflow */
#define EMFILE          24      /* Too many open files */
#define ENOTTY          25      /* Not a typewriter */
#define ETXTBSY         26      /* Text file busy */
#define EFBIG           27      /* File too large */
#define ENOSPC          28      /* No space left on device */
#define ESPIPE          29      /* Illegal seek */
#define EROFS           30      /* Read-only file system */
#define EMLINK          31      /* Too many links */
#define EPIPE           32      /* Broken pipe */
#define EDOM            33      /* Math argument out of domain of func */
#define ERANGE          34      /* Math result not representable */
#define EDEADLK         35      /* Resource deadlock would occur */
#define ENAMETOOLONG    36      /* File name too long */
#define ENOLCK          37      /* No record locks available */
#define ENOSYS          38      /* Function not implemented */
#define ENOTEMPTY       39      /* Directory not empty */
#define ELOOP           40      /* Too many symbolic links encountered */
#define EWOULDBLOCK     EAGAIN  /* Operation would block */
#define ENOMSG          42      /* No message of desired type */
#define EIDRM           43      /* Identifier removed */
#define ECHRNG          44      /* Channel number out of range */
#define EL2NSYNC        45      /* Level 2 not synchronized */
```

```
#define EL3HLT          46      /* Level 3 halted */
#define EL3RST          47      /* Level 3 reset */
#define ELNRNG          48      /* Link number out of range */
#define EUNATCH         49      /* Protocol driver not attached */
#define ENOCSI          50      /* No CSI structure available */
#define EL2HLT          51      /* Level 2 halted */
#define EBADE           52      /* Invalid exchange */
#define EBADR           53      /* Invalid request descriptor */
#define EXFULL          54      /* Exchange full */
#define ENOANO          55      /* No anode */
#define EBADRQC         56      /* Invalid request code */
#define EBADSLT         57      /* Invalid slot */

#define EDEADLOCK       EDEADLK

#define EBFONT          59      /* Bad font file format */
#define ENOSTR          60      /* Device not a stream */
#define ENODATA         61      /* No data available */
#define ETIME           62      /* Timer expired */
#define ENOSR           63      /* Out of streams resources */
#define ENONET          64      /* Machine is not on the network */
#define ENOPKG          65      /* Package not installed */
#define EREMOTE         66      /* Object is remote */
#define ENOLINK         67      /* Link has been severed */
#define EADV            68      /* Advertise error */
#define ESRMNT          69      /* Srmount error */
#define ECOMM           70      /* Communication error on send */
#define EPROTO          71      /* Protocol error */
#define EMULTIHOP       72      /* Multihop attempted */
#define EDOTDOT         73      /* RFS specific error */
#define EBADMSG         74      /* Not a data message */
#define EOVERFLOW       75      /* Value too large for defined data type */
#define ENOTUNIQ        76      /* Name not unique on network */
#define EBADFD          77      /* File descriptor in bad state */
#define EREMCHG         78      /* Remote address changed */
#define ELIBACC         79      /* Can not access a needed shared library */
#define ELIBBAD         80      /* Accessing a corrupted shared library */
#define ELIBSCN         81      /* .lib section in a.out corrupted */
#define ELIBMAX         82      /* Attempting to link in too many shared libraries */
#define ELIBEXEC        83      /* Cannot exec a shared library directly */
#define EILSEQ          84      /* Illegal byte sequence */
#define ERESTART        85      /* Interrupted system call should be restarted */
#define ESTRPIPE        86      /* Streams pipe error */
#define EUSERS          87      /* Too many users */
#define ENOTSOCK        88      /* Socket operation on non-socket */
#define EDESTADDRREQ    89      /* Destination address required */
#define EMSGSIZE        90      /* Message too long */
#define EPROTOTYPE      91      /* Protocol wrong type for socket */
#define ENOPROTOOPT     92      /* Protocol not available */
```

```
#define EPROTONOSUPPORT 93      /* Protocol not supported */
#define ESOCKTNOSUPPORT 94      /* Socket type not supported */
#define EOPNOTSUPP      95      /* Operation not supported on transport endpoint */
#define EPFNOSUPPORT    96      /* Protocol family not supported */
#define EAFNOSUPPORT    97      /* Address family not supported by protocol */
#define EADDRINUSE      98      /* Address already in use */
#define EADDRNOTAVAIL   99      /* Cannot assign requested address */
#define ENETDOWN        100     /* Network is down */
#define ENETUNREACH     101     /* Network is unreachable */
#define ENETRESET       102     /* Network dropped connection because of reset */
#define ECONNABORTED    103     /* Software caused connection abort */
#define ECONNRESET      104     /* Connection reset by peer */
#define ENOBUFS         105     /* No buffer space available */
#define EISCONN         106     /* Transport endpoint is already connected */
#define ENOTCONN        107     /* Transport endpoint is not connected */
#define ESHUTDOWN       108     /* Cannot send after transport endpoint shutdown */
#define ETOOMANYREFS    109     /* Too many references: cannot splice */
#define ETIMEDOUT       110     /* Connection timed out */
#define ECONNREFUSED    111     /* Connection refused */
#define EHOSTDOWN       112     /* Host is down */
#define EHOSTUNREACH    113     /* No route to host */
#define EALREADY        114     /* Operation already in progress */
#define EINPROGRESS     115     /* Operation now in progress */
#define ESTALE          116     /* Stale NFS file handle */
#define EUCLEAN         117     /* Structure needs cleaning */
#define ENOTNAM         118     /* Not a XENIX named type file */
#define ENAVAIL         119     /* No XENIX semaphores available */
#define EISNAM          120     /* Is a named type file */
#define EREMOTEIO       121     /* Remote I/O error */
#define EDQUOT          122     /* Quota exceeded */

#define ENOMEDIUM       123     /* No medium found */
#define EMEDIUMTYPE     124     /* Wrong medium type */

#endif
```

Ejemplo c004.c: Mostar errores con errno, perror() y strerror(errno)

Cómo se muestra el código de error generado en errno al fallar una llamada al sistema, en este caso el intento de cerrar un fichero cuyo identificador es 23 (fid = file id)

```
// ================================================================
// Programa #4: Consultar el valor numérico de la variable global "errno"
// Archivo: ejemplo4.c
// ================================================================
#include <stdio.h>
```

```c
#include <unistd.h>
#include <errno.h> // La variable "errno" se declara en este fichero
#include <string.h>
int main (int argn, char *argv[]) {
  // Llamada al sistema para cerrar un fichero que no está abierto
  if ( close(23) == -1) { // Lógicamente, se produce un error. ¿Cual?
     printf ("Motivo del error de la última llamada al sistema: %d \ncuya descripcion es: %s\n", errno ,strerror(errno));
     perror("[error c004 en main]");
     return 1;
     }
return 0;
}
```

Ejemplo c008.c: Uso de las constantes de error

Cómo podemos tratar de manera diferente en función del código (constante genérica) del código de error producido.

```c
// ========================================================================
// Programa #8: Uso de las constantes
// Archivo: ejemplo8.c
// ========================================================================
#include <stdio.h>
#include <unistd.h>
#include <errno.h> // La variable "errno" se declara en este fichero
#include <string.h>
int main (int argn, char *argv[]) {
  // Llamada al sistema para cerrar un fichero que no está abierto
  if ( close(-23) == -1){ // Lógicamente, se produce un error. ¿Cual?
      if (errno==EBADF) {
          // Bad file descriptor
          printf ( "Descriptor de fichero incorrecto\n" );
          }
      else if (errno==EIO) {
              // Input/Output Error
              printf ( "Error físico de E/S\n" );
              }
      else {
         printf ( "Error: %s\n", strerror(errno) );
         }
   }
return 0;
}
```

Variables de entorno

Ejemplo c009.c: Variables de entorno vía parámetros de main

Cómo un tercer parámetro con las variables de entorno es procesado en main. Recibe array de cadenas de variables de entorno definidas en el Shell que lanza al proceso.

```
// ========================================
// Programa #9: Tabla de variables de ambiente
// Archivo: ejemplo9.c
// ========================================
#include <stdio.h>
int main (int argn, char *argv[], char *env[]) {
  int i;
  for (i=0; env[i]!=NULL; i++) {
       printf ("%s\n",env[i]);
       }
return 0;
}
```

Para más detalles sobre la variable de **environ** ver http://man7.org/linux/man-pages/man7/environ.7.html

Ejemplo c010.c: Variables de entorno vía variable extern environ

Vemos cómo se puede crear una tabla de variables de entorno

```
// ========================================
// Programa #10: Tabla de variables de ambiente
// Archivo: ejemplo10.
// ========================================
#include <stdio.h>
#include <unistd.h>
extern char **environ; // Declaración externa de environ
int main (void) {
  int i;
  for (i=0; environ[i]!=NULL; i++)
      printf ("%s\n",environ[i]);
return 0;
}
```

Ejemplo c011.c: Consulta de una variable de entorno

de cómo podemos consultar el valor de una determinada variable de entorno, en este caso PATH

```c
// ======================================================
// Programa #11: Consulta de la variable de ambiente PATH
// Archivo: ejemplo11.c
// ======================================================
#include <stdio.h>
#include <string.h>
#include <unistd.h>
extern char **environ; // Declaración externa de environ
int main (void) {
  int i;
  for (i=0; environ[i]!=NULL; i++) {
      if (memcmp(environ[i],"PATH=",5) == 0) {
          printf ("La variable PATH vale: %s\n",environ[i]+5);
          break;
          }
      }
return 0;
}
```

Ejemplo c012.c: Consulta de las rutas de la variable PATH

De cómo podemos separar las distintas rutas de la variable PATH en distintas cadenas.

```c
// ======================================================
// Programa #12: Separación de las rutas de la variable PATH
// Archivo: ejemplo12.
// ======================================================
#include <stdio.h>
#include <string.h>
#include <unistd.h>
extern char **environ; // Declaración externa de environ

// Prototipos de las funciones utilizadas
// Consultar la variable de ambiente PATH: Devuelve la variable en si
static char *GetEnvPath(void);

// Consultar la variable de ambiente PATH: Devuelve el índice de la variable en el array de entorno
static int GetEnvPath2 (void);

// Extraer las distintas rutas desde la variable de ambiente PATH
static int GetPathDirs(char *path,char *patharray[],int arraysize);

// Mostrar las rutas ya separadas
static void PrintPaths(char *paths[],int arraysize);

//---------------------------------------
```

```
int main (void){
char *pathexp;
char pathbuf[1000];
char *paths[100];
int count;

//Una forma de obtener el PATH
pathexp = GetEnvPath();

//Otra forma de obtener el PATH
pathexp = environ[GetEnvPath2()];

strcpy (pathbuf,pathexp);
count = GetPathDirs(pathbuf,paths,100);
PrintPaths (paths,count);
return 0;
}

//----------------------------------------
static char *GetEnvPath (void) {
int i;
for ( i=0; environ[i]!=NULL; i++) {
    if (memcmp(environ[i],"PATH=",strlen("PATH=")) == 0) {
        return environ[i];
        }
    }
return "";
}

static int GetEnvPath2 (void) {
int i;
for ( i=0; environ[i]!=NULL; i++) {
    if (memcmp(environ[i],"PATH=",strlen("PATH=")) == 0) {
        return i;
        }
    }
return -1;
}

//----------------------------------------
static int GetPathDirs (char *path, char *array[], int size){
int i;
int pos;
path+=strlen("PATH=");
array[0]=path;
for ( i=0,pos=1; path[i]!=0; i++) {
    if ( path[i]==':' ) {
        path[i]=0;
        array[pos++] = path+i+1;
```

```
        if (pos == size)
            return size;
        }
    }
return pos;
}

//---------------------------------------
static void PrintPaths (char *paths[], int size) {
int i;
printf ("Total de rutas encontradas: %d\n", size);
for (i=0; i<size; i++)
     printf ("%d. : %s\n", i+1, paths[i] );
}
```

Ejemplo c013.c: Uso de getenv() para obtener una variable de entorno

Por ejemplo, para visualizar el valor de la variable PATH podríamos hacer:

En ambos casos obtendremos los mismos resultados.

Una forma más cómoda de obtener el valor de una variable a partir de su nombre es con la función "**getenv(nombre)**". Nos evita que realizar un bucle en busca de la entrada en la tabla de variables de ambiente. En el siguiente ejemplo pasamos desde la línea de órdenes como parámetro el nombre de la variable de ambiente que queremos visualizar. Algo parecido al comando echo.

```
$ echo $PATH
$ c01313 PATH
```

```
// ====================================================================
// Programa #13: Obtención del valor de una variable de ambiente con getenv()
// Archivo: ejemplo13.c
// ====================================================================
#include <stdio.h>
#include <stdlib.h>

int main (int argn, char *argv[]) {
int i;
char *valor;
// Si sólo tenemos un argumento (nombre del programa) salimos.
if (argn < 2) exit(1);

// Mostramos el valor de todas las variables especificadas (si existen)
```

```
for ( i=1; i<argn; i++) {
    valor = getenv ( argv[i] );
    if ( valor == NULL ) {
        printf ("Variable de ambiente '%s' no definida\n", argv[i]);
        }
    else {
        printf ("V13 alor de '%s' = %s\n",argv[i], valor);
        }
    }
return 0;
}
```

← Entrada anterior

Entrada siguiente →