Programación de sistemas UNIX (POSIX)

Autor: Stefan Beyer

UNIX con todas sus variantes es probablemente el sistema operativo con más éxito. Aunque sus conceptos básicos ya tienen más de 30 años, siguen siendo la base para muchos sistemas modernos. Últimamente, Linux y los sistemas operativos basados en BSD (Mac OS X, FreeBSD, etc.) se han presentado como alternativa real para casi todos los tipos de sistemas informáticos. Ahora, existen variantes UNIX para sistemas embebidos, ordenadores de sobremesa, servidores, clusters y grandes mainframes. Eso tiene la ventaja que todos estos sistemas (desde un microcontrolador para una lavadora hasta un mainframe) se pueden programar utilizando el mismo interfaz (o por lo menos un interfaz bastante similar).

Todos los sistemas UNIX vienen con una librería muy potente para programar el sistema. Para los programadores de UNIX (eso incluye los programadores de Linux y Mac OS X) es fundamental aprender esa librería, porque es la base para muchas aplicaciones en estos sistemas. Los lenguajes de alto nivel como Java no son adecuados para muchas tareas, por ejemplo para interaccionar con las interfaces de dispositivos reales que proporciona el sistema operativo. Desafortunadamente, como hay una gran variedad de sistemas UNIX, hay también una gran variedad de librerías. Sin embargo, casi todos los sistemas tienen en común los siguientes aspectos:

- Se usa el lenguaje C para la programación al nivel del sistema.
- Las librerías soportan el estándar ISO para librarías de C como un subconjunto del interfaz que proporcionan.
- Hay un interfaz para las llamadas al sistema. Estas llamadas pueden variar, pero las llamadas más importantes no cambian mucho entre sistemas.

La causa de la gran variedad de sistemas UNIX es una historia complicada con conflictos entre vendedores distintos. Durante muchos años la diversidad de sistemas UNIX produjo muchos problemas de portabilidad. Era una suerte si un programa escrito para un sistema funcionaba también en el sistema de otro vendedor.

Afortunadamente, después de varios intentos de estandarización se introdujeron los estándares POSIX (Portable Operating Systems Interface). POSIX es un conjunto de estándares que definen un conjunto de servicios e interfaces con que una aplicación puede contar en un sistema operativo. Además los estándares POSIX incluyen herramientas básicas y un intérprete de órdenes estándar. Este artículo solamente trata de los interfaces de un sistema UNIX y no de las herramientas incluidas al nivel de las aplicaciones. En los últimos años POSIX ha sido ampliado y después de la unificación con otros estándares se conoce al estándar como "Single UNIX Specification (SUS), versión 3" [1] o como "IEEE Std. 1003.1-2001, POSIX".

Este artículo sirve como introducción a las posibilidades de la programación de sistemas UNIX usando los interfaces POSIX. No pretende ser un manual de programación, sino una introducción a los interfaces más útiles. Para manuales más completos sobre la programación en UNIX se puede recomendar los libros [2] y [3]. Además todas las llamadas están documentadas en las páginas del manual de UNIX. El manual de una llamada se puede acceder con la herramienta **man** (**man man** permite ver la documentación de la propia herramienta).

Interfaces de POSIX

Para que se pueda decir que un sistema cumple el estándar POSIX, el sistema tiene que implementar por lo menos el estándar base de POSIX. Sin embargo, muchas de las interfaces más útiles están definidas en extensiones. La implementación de esas extensiones no es obligatoria, pero casi todos los sistemas modernos soportan las extensiones más importantes.

Las más utilizadas son las interfaces para:

- la creación y la gestión de procesos,
- entrada-salida,
- comunicación sobre redes (sockets),
- la comunicación entre procesos (IPC),
- señales.

En el resto del artículo se introduce cada una de estas interfaces.

Gestión de procesos

Una definición simplificada de un proceso es un programa en ejecución. Cada proceso tiene su propio espacio de memoria. UNIX utiliza un algoritmo de planificación para dar intervalos de tiempo de ejecución a los distintos procesos activos en el sistema.

El sistema identifica a cada proceso por medio de un número identificador de proceso. Para crear un proceso hay que hacer una copia de otro. El nuevo proceso se llama "proceso hijo" y el antiguo "proceso padre". Después de crear el hijo se puede substituir el programa en ejecución en el hijo por otro programa. Todos los procesos en el sistema han sido creados de esta manera y forman una jerarquía con un origen común: el primer proceso creado durante la inicialización del sistema (normalmente llamando **init,** con identificador 1). Si el padre de un proceso muere otro proceso adopta a este hijo (normalmente **init** adopta a los procesos sin padre, aunque POSIX no lo exige).

La creación de un nuevo proceso se realiza con la llamada **fork()**:

```
#include <unistd.h>
pid_t fork (void);
```

La llamada devuelve el identificador del hijo al padre y 0 al hijo. De esta manera se pueden distinguir los dos procesos durante el resto del código. En caso de no poder crear una copia del proceso, la llamada devuelve -1 y modifica el valor de la variable global **errno** para indicar el tipo de error. El siguiente ejemplo demuestra el uso de **fork()**:

```
pid_t hijo_pid;
hijo_pid = fork();
if (hijo_pid == -1) {
  perror("No puede crear proceso");
  return 1;
}
if (hijo_pid == 0) /* hijo */
  printf("hijo con pid: %ld\n", (long)getpid());
else /* padre */
  printf("padre con pid: %ld\n", (long)getpid());
```

Este trozo de código crea una copia del proceso actual. Dado que los dos procesos comparten el mismo código, es necesario comprobar el valor devuelto por **fork()** para distinguir padre e hijo. Ambos procesos continúan con la ejecución del mismo código después de la llamada a **fork()**, pero cada uno de los procesos tiene otro valor para la variable **hijo_pid**. Para demostrar que los procesos son realmente diferentes, los procesos utilizan la llamada **getpid()** para imprimir su identificador. Se puede utilizar también la llamada **getppid()** para obtener el identificador del padre de un proceso.

Normalmente, se crea un proceso nuevo para ejecutar otro programa en vez de compartir el código con el padre. Por esta razón existe un grupo de llamadas para ejecutar un programa:

```
#include <unistd.h>

int execl (const char *camino, const char *arg, ...);
int execlp (const char *fichero,
            const char *arg, ...);
int execle (const  char  *camino,
            const char *arg , ..., char * const
            envp[]);
int execv (const char *camino, char *const argv[]);
int execvp(const char *fichero, char *const argv[]);
```

Todas las funciones sirven para ejecutar un programa. La diferencia está en la forma de especificar la ruta donde se encuentra el programa, la forma de pasar los argumentos al programa y el entorno en que el programa ejecuta. Las funciones están documentadas en detalle en las páginas del manual.

El siguiente ejemplo demuestra como utilizar la combinación de **fork()** y la familia de funciones **exec**:

```
...
hijo_pid = fork();
if (hijo_pid == 0)
        execl("/bin/ls", "ls", "-l", NULL);
...
```

Después de comprobar el valor que ha devuelto **fork()** para asegurar que la próxima sentencia se ejecute solamente en el hijo, el código utiliza **execl()** para lazar el programa **ls** con el argumento **–l**. El nombre del programa también se ha de pasar como primer argumento, porque en C el primer argumento (**argv[0]**) de la función **main()** es el nombre del ejecutable.

Cuando un proceso acaba se supone que hay un proceso que le "espera". Normalmente el padre del proceso toma este papel. Si un proceso acaba y no hay ningún otro proceso que le espera, el proceso pasa a ser un *zombie*. Los zombies quedan en el sistema. Periódicamente el proceso **init** espera a hijos para quitar los zombies del sistema.

Para esperar a un proceso hay dos funciones:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int *status)
pid_t waitpid (pid_t pid, int *status, int options);
```

La función **wait()** suspende el proceso hasta que un hijo termina o hasta que se produce una señal. **waitpid()** suspende el proceso hasta que el hijo especificado por el argumento pid termina o hasta que se produce una señal. También se pueden especificar algunas opciones extra en el tercer argumento. Las opciones están documentadas en las páginas del manual. Ambas funciones guardan el estado del proceso a esperar en el argumento "status" si dicho argumento no es NULL.

La sentencia

```
hijo_pid = wait (NULL);
```

es las forma más básica de esperar a que un hijo termine. Para evitar *zombies* en caso de la entrega de una señal al proceso actual, se puede encapsular la llamada en un bucle de esta manera:

```
while ((hijo_pid = wait(NULL)) == -1)
&& errno == EINTR));
```

Naturalmente, la interfaz de gestión de procesos tiene muchas más funciones y usos, pero lo que se ha presentado arriba sirve para introducir el funcionamiento básico de muchos programas para UNIX. Por ejemplo hasta hace poco las aplicaciones tipo servidor de redes creaban un proceso nuevo para servir varias conexiones a la vez. Ahora existe la extensión de hilos para POSIX (pthreads) que permite servir varias peticiones en paralelo sin crear nuevos procesos. Eso resulta mucho más eficiente, pero como es solamente una extensión al estándar no hay garantía que cada sistema POSIX incluya soporte para hilos.

Entrada-Salida

Otra interfaz con gran importancia es la de entrada-salida. En UNIX casi todo es un fichero. Eso significa que los recursos para entrada y salida aparecen como un archivo en el sistema de archivos. Por eso las llamadas al sistema para abrir y cerrar archivos y para escritura y lectura son de las más importantes de UNIX. Un archivo abierto al nivel del sistema está identificado con un número entero positivo llamado *descriptor de archivos*.

Para abrir un archivo, se utiliza la función **open()**:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *camino, int flags);
int open(const char *camino, int flags, mode_t modo);
```

La función asocia el archivo definido por el argumento "camino" con un descriptor de archivos. El segundo argumento "flags" identifica los permisos de acceso (por ejemplo escritura, lectura o ambos) y el "modo" de acceder al archivo (por ejemplo sobrescribir un archivo existente o añadir datos al final del archivo). Si se trata de crear un nuevo archivo, se tiene que incluir un tercer argumento "modo" para definir los permisos de acceso al archivo dentro del sistema de archivos. Las opciones para los argumentos flags y modo están definidas en la página del manual.

Para leer de un archivo abierto se utiliza la función **read()**:

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t nbytes);
```

El primer argumento es el descriptor del archivo en que se quiere escribir. El segundo argumento es un puntero al buffer donde se encuentran los datos que escribir. El último argumento es el número de bytes a escribir.

La función equivalente para escribir es **write()**:

```
#include <unistd.h>
ssize_t write (int fd, const void *buf, size_t num);
```

Los argumentos son los equivalentes a los argumentos de **read()**.

Para mover el punto de escritura dentro de un fichero se usa la función **lseek()**, que también está documentada en las páginas del manual.

Lo interesante de todas estas llamadas es que no solo sirven para trabajar con archivos. Las funciones también forman parte integral del interfaz de entrada / salida y sirven para enviar y recibir datos y órdenes a dispositivos que aparecen en el sistema de archivos del sistema como archivos especiales. Eso funciona muy bien, pero por desgracia hay órdenes para configurar dispositivos que no se pueden mandar de esta forma en UNIX. (En contraste, el sistema operativo Inferno [4] es un sistema muy interesante que lleva la idea de UNIX de representar todos los recursos como archivos al extremo. En este sistema es posible trabajar solamente con llamadas de lectura y escritura, pero eso resulta en una multitud de archivos representando cada aspecto de control sobre un archivo.) Por esa razón existe la función **ioctl()**, cuya sintaxis y uso son demasiados complicados para este artículo. El uso de la función es tan complicado y variado que para algunos dispositivos (por ejemplo el terminal) se han introducido funciones especializadas.

Comunicación sobre redes

La interfaz de comunicación sobre redes tiene sus orígenes en BSD. Los "sockets" de BSD al principio no estaban incluidos en POSIX, pero el comité responsable para el estándar se dio cuenta de este error muy pronto y ahora la interfaz socket forma parte obligatoria de POSIX.
Un socket es un extremo de un canal de comunicación. Este canal de comunicación no se refiere necesariamente a una comunicación sobre una red, aunque en este artículo se habla solamente sobre este tipo de sockets. Al nivel de programación un socket es solamente un descriptor de archivo. Así, es posible utilizar las llamadas de entrada / salida introducidas anteriormente sobre un socket. Sin embargo hay una interfaz especializada más adecuada para sockets.

Para empezar es importante distinguir dos tipos de comunicación: la comunicación conectada y desconectada. Estos dos tipos de comunicación corresponden a los dos tipos de protocolos de transporte en la pila de protocolos TCP/IP. Si se elige comunicación conectada, el sistema utiliza internamente TCP para establecer un canal de comunicación fiable entre los dos extremos de la comunicación. La aplicación puede escribir datos en un socket y esos datos se pueden leer al otro extremo. El protocolo se encarga de entregar los datos de forma fiable y en el orden correcto (en realidad TCP no ofrece comunicación cien por cien fiable según varias definiciones de fiabilidad, pero para la mayoría de las aplicaciones resulta adecuado). Si se elige comunicación desconectada, no se establece ningún canal de comunicación. Se pueden enviar datagramas de UDP a otros nodos en la red, pero el sistema no da ninguna garantía de que los datos se estén entregado correctamente ni en el orden correcto.

Se puede leer cómo establecer y utilizar sockets de cada tipo en las páginas del manual. Las funciones de interés son **socket()**, **bind()**, **accept()**, **listen()**, **connect()**, **send()**, **sendto()**, **recv()** y **recvfrom()**.

Comunicación entre procesos

El interfaz para comunicación entre procesos tiene sus orígenes en System VR4, un sistema de AT&T con bastante éxito. Este interfaz (llamado POSIX IPC) ha sido estandardizado en una extensión para POSIX (POSIX:XSI).

POSIX IPC permite compartir datos entre procesos en el mismo sistema. Hay tres mecanismos de comunicación: *colas de mensajes*, *semáforos* y *memoria compartida*. Una cola de mensajes es una forma de comunicación que permite mandar mensajes de un proceso a otro. En contraste, la memoria compartida es una forma de compartir datos en un segmento de memoria al que varios procesos pueden acceder. El concepto de semáforos es un invento del famoso informático Dijkstra [5]. Para este artículo es suficiente decir que un semáforo es una herramienta de concurrencia que permite sincronizar varios procesos.

Los objetos de comunicación entre procesos pueden existir cuando acaban los procesos y por eso POSIX incluye un conjunto de herramientas para gestionar esos objetos desde el intérprete de órdenes.

Cada objeto tiene un identificador y está asociado a una clave. Un proceso puede descubrir el identificador a partir de esa clave.

La siguiente tabla contiene una descripción de la interfaz de los tres mecanismos. Los detalles de esas funciones se pueden encontrar en las páginas del manual.

mecanismo	función POSIX	explicación
cola de mensajes	msgctl	control
	msgget	crear / acceder
	msgrcv	recibir mensaje
	msgsnd	mandar mensaje
semáforos	semctl	control
	semget	crear / acceder
	semop	ejecutar operación
memoria compartida	shmat	vincular memoria a proc.
	shmctl	control
	shmdt	separar memoria de proc.
	shmget	crear / iniciar / acceder

Señales

Una señal es la notificación de un evento a un proceso en software. Después de que una señal ha sido generada puede pasar un rato hasta que sea entregada porque el proceso de destino tiene que estar en ejecución en un procesador. Los procesos pueden registrar manejadores que sean llamados cuando se entregue una señal.

Las diferentes señales están definidas en el fichero <signal.h>, que se tiene que incluir en cada programa de C que usa señales. Si no hay un manejador definido por la aplicación un manejador predeterminado se ejecuta en caso que se entregue una señal.

Se puede manipular la máscara de señales para bloquear señales. Una señal bloqueada no es ignorada (lo que también sería posible), sino entregada más tarde, cuando sea desbloqueada. La única señal que no se puede ignorar y para la cual no se puede registrar un manejador es la señal `SIGKILL`, que causa la terminación anormal del programa.

Un proceso puede generar señales para otros procesos del mismo usario con la función **kill()**:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

El nombre de la función puede llevar a confusión, porque no solamente sirve para generar la señal `SIGKILL`, sino para generar todo tipo de señales.

Para registrar un manejador para una señal se utiliza la función **sigaction()**:

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
        struct sigaction *oldact);
```

El primer argumento identifica la señal, el segundo argumento es una estructura de datos que identifica la acción a tomar cuando se entrega una señal de este tipo. El último argumento es una estructura de datos del mismo tipo, en que la función guarda la acción registrada anteriormente para la señal.

La estructura **sigaction()** está definida en <signal.h> y contiene como campos por lo menos un puntero a la función que implementa el manejador, una máscara que especifica otras señales que ignorar durante la ejecución de la manejador, algunas opciones adicionales y un puntero para un manejador de tiempo real. La definición exacta de estos campos no importa para este artículo.

Para ignorar una señal se puede pasar `SIG_IGN` como manejador y para registrar la acción predeterminada se puede pasar `SIG_DFL`.

Conclusión

Este artículo sirve como introducción de las interfaces más útiles y no pretende ser un manual de programación. Las interfaces que han sido introducidas tienen muchos más usos. También hay interfaces que no han sido mencionadas. Por ejemplo, la extensión POSIX:THR define un interfaz para múltiples hilos de ejecución dentro de un proceso. De hecho la última versión de POSIX tiene 1742 funciones. Un programador de UNIX que tenga la intención de utilizar las potentes interfaces que proporciona su sistema, debería leer uno de los manuales incluidos como referencias en este artículo. ■

Referencias

[1] *The Single UNIX Specification Version 3*. The Open Group. **www.unix.org**
[2] Kay A. Robbins y Steven Robbins, UNIX Systems Programming, Prentice Hall. 2003.
[3] W. Stevens, Advanced Programming in the UNIX Environment, Addison-Wesley. 1992.
[4] El sistema operativo Inferno, **www.vitanuova.com/inferno**
[5] E. Dijkstra, "Co-operative sequential processes", Programming Languages, Academia Press, 1968m pg. 43-112