APIs de Sistemas Operativos

Un sistema operativo ofrece a las aplicaciones servicios que permiten acceder a recursos del sistema y facilidades como mecanismos de comunicación y sincronización entre procesos y otros.

El sistema operativo UNIX, creado por Ken Thompson y Dennis Ritchie provee interfaces simples y coherentes. La mayoría de los SO modernos se basan en estas APIs.

La API de UNIX se describe como un conjunto de *funciones C* que implementan *syscalls*. A continuación se muestra una lista parcial de llamadas al sistema correspondientes a operaciones sobre procesos y archivos.

Procesos

- `fork()` : Crea un nuevo proceso (hijo) el cual es una copia del proceso invocante.
- `exit(exit_code)` : Finaliza el proceso con código de salida `exit_code` .
- `pid wait(&exit_code)` : Espera (se bloquea) hasta que finalice un proceso hijo. Retorna el *pid* del hijo y en `exit_code` se recibe el código de salida.
- `getpid()` : Retorna el identificador del proceso corriente.
- `getppid()` : Retorna el identificador del proceso padre.
- `kill(pid)` : Envía una señal (terminación) al proceso con id `pid` .
- `sleep(n)` : Espera (se bloquea) por `n` segundos.
- `exec(filename, args)` : Carga el programa `filename` y lo ejecuta. Reemplaza la imagen de código y datos de la memoria del proceso corriente.
- `sbrk(n)` : Asigna `n` bytes de memoria adicionales al proceso corriente.

Archivos

- `open(filename, mode)` : Abre un archivo en el modo (*read*, *write*, ...) dado.
- `read(fd, buffer, count)` : Lee `count` bytes del archivo en `buffer` .
- `write(fd, buffer, count)` : Escribe `count` bytes de `buffer` al archivo.
- `close(fd)` : Cierra el archivo (libera recurso).
- `dup(fd)` : Dumplica (copia) `fd` en el primer descriptor no usado.
- `pipe()` : Crea un *pipe* para comunicación entre procesos.
- `chdir(dirname)` : Cambia de directorio.
- `mkdir(dirname)` : Crea un directorio.
- `mknod(name, major, minor)` : Crea un archivo *especial* (device).
- `fstat(fd)` : Retorna el *estado* y metadatos (size, ...) del archivo.
- `link(f1, f2)` : Crea un *alias* (`f2`) para el archivo `f1` .

- `unlink(filename)` : Borra el archivo.

En algunas versiones modernas de SO tipo UNIX algunas de estas *llamadas al sistema* se han extendido ampliamente, como por ejemplo en GNU-Linux.

Todas las llamadas al sistema se presentan al programador como funciones de biblioteca. Su implementación dispara un `syscall` , es decir una entrada a un punto del kernel. A partir de allí se ejecuta código del kernel hasta su retorno.

Todas las llamadas al sistema retornan un entero. Generalmente en caso de falla retornan un valor negativo (comúnmente -1).

Interfaz del usuario

Un SO presenta algún tipo de interfaz al usuario. Generalmente el programa que se encarga de interactuar con el usuario es un *shell*, el cual puede ser basado en *línea de comandos* (como *bash*, *zsh*, *csh* de UNIX o *cmd* en MS-Windows).

Un *shell* de línea de comandos permite al usuario ingresar comandos para ejecutar programas y soportan operadores sobre el control, sincronización y comunicación entre los procesos creados.

El shell se dispara al inicio del sistema, generalmente luego que el usuario inicia su sesión (*login*) asociada a la *consola* o *terminal*.

Típicamente un *shell* presenta al usuario un *prompt* que puede incluir su identificador de usuario, finalizado comúnmente por el caracter `#` . Cabe aclarar que el prompt de un shell es configurable por el usuario (Ej: `.bash_profile` en sistemas tipo UNIX).

El sistema presenta al usuario un *sistema de archivos*. Hay archivos de diferentes tipos:

1. *Programas*: archivos ejecutables (o *comandos*).
2. *Datos*: Contienen datos en algún formato (binario o texto).

Un archivo de textos contiene una secuencia de códigos de caracteres (ej: ASCII, UTF-8, ...). Generalmente se estructuran en líneas, es decir, secuencias de caracteres finalizados por `\n` (*newline*) o `\n\r` (*newline*/*carriage-return*).

El conjunto de archivos del sistema se organizan en una estructura jerárquica (de árbol) con *directorios* (o *carpetas*), los cuales a su vez contienen otro subárbol de directorios y archivos.

Para hacer referencia a un archivo se puede hacer de dos formas:

1. *Full path*: Camino desde la raíz del sistema de archivos hasta el archivo.

ejemplo: `cat /home/user/docs/myfile.txt`

2. *Relativa*: En base al *directorio corriente*, es decir, el directorio de trabajo actual al que se accede mediante el comanddo `cd dir-path`).

   Ejemplo: `cd docs ; cat myfile.txt`

En los sistemas tipo UNIX la convención es que un proceso con *código de salida=0* significa *terminación normal*, mientras que otros valores representan algún código de error, dependiendo de la aplicación.

Un shell típico en sistemas tipo UNIX reconoce los siguientes operandos y operadores:

- Operandos: Pueden ser un *comando* o un *nombre de archivo*
- Operadores:
  - `cmd1 ; cmd2` : Ejecuta `cmd1`, al finalizar ejecuta `cmd2`.
  - `cmd1 && cmd2` : Ejecuta `cmd2` sólo si `cmd1` finaliza exitosamente.
  - `cmd1 || cmd2` : Ejecuta `cmd2` sólo si `cmd1` finaliza con error.
  - `cmd &` : Lanza el `cmd` de fondo (en *background*), el shell no espera a su terminación, devolviendo el *prompt* inmediatamente.
  - `cmd n> file` : Se *redirige* el *descriptor de archivo de salida* `n` al archivo `file`. Si `n` se omite, se asume 1 (salida estándar).
  - `cmd < file` : Redirige la entrada estándar (comúnmente el teclado) desde el archivo `file`.
  - `cmd1 | cmd2` : Ejecuta `cmd1` y `cmd2` concurrentemente, *redirige* la salida estándar de `cmd1` a un *pipe* y la entrada estándar de `cmd2` a la *salida del pipe*. Más abajo se dan mas detalles sobre *pipes*.

Procesos

En esta sección se analizan las llamadas al sistema para la creación, destrucción,sincronización y comunicaciones entre procesos (*Inter Process Communication - IPC*).

En una API tipo UNIX la única llamada al sistema para crear un nuevo proceso es `fork()`, la cual crea una copia del proceso corriente, conjuntamente con su estado.

El estado de un proceso (mantenido por el kernel) tiene generalmente los siguientes atributos:

- Su estado: `RUNNING`, `READY`, `SLEEPING`, `TERMINATED` ...
- Su *identificador* o *pid*: Un número > 0

- Su *imagen de mamoria*: Código, datos globales y stack.
- El conjunto (tabla) de *archivos/pipes* abiertos.
- Una referencia a su padre (*parent*)
- Un stack para ejecución en modo *kernel*
- Espacio para *salvar* su *contexto* (estado de la CPU) cuando cambie de estado.
- El código de salida o terminación.

La llamada al sistema `fork()`, ejecutada inicialmente por un proceso *padre* hace que el kernel cree una copia (el *hijo*) del proceso corriente, el cual *hereda* su estado: El conjunto de valores de los registros de la CPU y la *tabla de descriptores de archivos abiertos*, entre otros.

Esto hace que cuando en el futuro el *planificador de procesos* decida otorgarle la CPU al proceso hijo éste iniciará su ejecución en el punto del retorno del `fork()` ya que ese era el contenido del program counter.

En el proceso padre `fork()` retorna el `pid` (> 0) del nuevo proceso (*hijo*). En el hijo retorna 0 (cero).

El nuevo proceso (hijo), al heredar el estado del padre, inicia su ejecución en el retorno del `fork()`, aunque aquí retorna 0.

El siguiente ejemplo muestra el uso de `fork()`.

c

```c
#include <unistd.h> /* UNIX syscalls */
#include <stdio.h>  /* portable i/o functions */

int main(void) {
    int pid = fork();
    if (pid == 0) {
        printf("I'm the child. Pid: %d\n", getpid());
    } else {
        printf("I'm the parent. Pid: %d\n", getpid());
        printf("My child is: %d\n", pid);
    }
}
```

Un proceso *padre* debería esperar hasta que sus hijos terminen antes de finalizar.

A continuación se muestra un ejemplo del uso de `wait` y `exit`.

c

```c
#include <unistd.h> /* UNIX API (syscalls) */
#include <stdio.h>  /* portable i/o functions */
```

```c
int main(void) {
    if (fork() == 0) {
        printf("I'm the child. Pid: %d\n", getpid());
        exit(0);
    } else {
        int child, status;
        printf("I'm the parent. Pid: %d\n", getpid());
        child = wait(&status);
        printf("In parent: child %d finished\n", child);
    }
}
```

La llamada al sistema `exec(program, args)` reemplaza la imagen de memoria (código y datos) del proceso corriente con el código y datos de `program`. Cabe aclarar que `exec()` no crea un nuevo proceso. El proceso comienza su ejecución desde el `entry point` (*function address*) que está especificado en el archivo ejecutable `program`.

Es común que un programa *lance* otro programa para realizar alguna tarea para luego continuar otras operaciones. En este modelo se debe hacer siguiendo el patrón mostrado en el siguiente ejemplo, el cual ejecuta el comando `ls -l`:

c

```c
#include <unistd.h> /* UNIX syscalls */
#include <stdlib.h> /* wait() */
#include <stdio.h>  /* portable i/o functions */

int main(void) {
    char *args[] = {"ls", "-l", 0};

    if (fork() == 0) {
        /* in child */
        execve("/bin/ls", args, 0);
        /* on success, below is unreachable code */
        printf("Ooops, exec() failed!\n");
        exit(-1);
    } else {
        wait(0);
        printf("In parent: child finished\n");
    }
}
```

Shell: Control de procesos

Un comando de la forma `cmd &` lanza el programa `cmd` como un *proceso de fondo (background)*. El shell le asigna un *identificador de job* como se muestra abajo.

sh

```
$ ping google.com > output &
[1] 4287
$ _
```

El shell muestra el *job id* (ejecutando en background) 1 asociado al proceso con pid 4287.

El comando `jobs` listará los procesos en background y su estado.

sh

```
[1]+  Running                 ping -i 5 google.com &
```

Un proceso en *foreground* (el shell está esperando a que termine) puede *suspenderse* o pararse usando `Ctrl-Z` . El shell devuelve el *prompt* y puede usarse el comando `bg %job_id` para continuarlo de fondo.

Es posible *finalizar* (*matar*) un proceso de fondo mediante el comando `kill %job_id` .

sh

```
$ ping google.com
PING google.com (74.125.226.71) 56(84) bytes of data.
64 bytes from lga15s44-in-f7.1e100.net (74.125.226.71): icmp_seq=1 ttl=5
64 bytes from lga15s44-in-f7.1e100.net (74.125.226.71): icmp_seq=2 ttl=5
64 bytes from lga15s44-in-f7.1e100.net (74.125.226.71): icmp_seq=3 ttl=5
Ctl-Z
[1]+  Stopped               ping google.com
$ bg
[1] 4579
...
kill %1
[1]+  Finished              ping google.com
```

También es posible enviar al *foreground* un proceso corriendo en *background* con el comando `fg` .

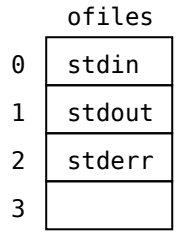
Redirección de entrada/salida

Cada proceso inicia comúnmente con tres archivos abiertos (heredados) desde `init`, el primer proceso del sistema, lanzado por el *kernel* luego del *boot*.

El *file descriptor 0* (*standard input*) está asociado a la entrada de la *consola*, comúnmente un *teclado*.

Los descriptores 1 y 2 (*standard output and error*, respectivamente) están asociados a la salida de la *consola*, típicamente la pantalla o *display*.

Es posible ver los *archivos abiertos* de un proceso como una tabla que mantiene internamente el kernel de la forma:



La llamada al sistema `open(filename, mode)` retorna un nuevo descriptor de archivo, el cual puede verse como un *índice* de la tabla *ofiles* (*opened files*).

Este modelo de dos pasos `fork()/exec()` permite al proceso padre *modificar el ambiente de ejecución* del nuevo programa a lanzar. En particular puede *redireccionar* las entradas y salidas.

El siguiente ejemplo permite muestra cómo es posible lanzar un proceso hijo que en lugar de escribir en la entrada estándar lo hará en el archivo `/tmp/out.txt`.

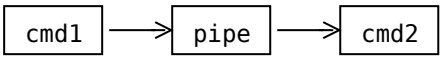
c

```c
#include <unistd.h> /* UNIX syscalls */
#include <fcntl.h>  /* file operations and flags */

int main(void) {
    if (fork() == 0) {
        /* in child: open/create file with permissions (user: rw, group:
        int fd = open("out.txt", O_WRONLY | O_CREAT, 0640);
        printf("File fd: %d\n", fd);    /* fd should have the value 3 */
        close(1);                       /* close stdout */
        dup(fd);                        /* duplicate fd: ofiles[1] == fc
        write(1, "Hello world\n", 12);
    }
}
```

Pipes

Un *pipe* es un mecanismo de comunicación (unidireccional) entre procesos. Un comando de shell de la forma `cmd1 | cmd2` hace que se lancen concurrentemente `cmd1` y `cmd2` previa redirección de la salida estándar al extremo de escritura del *pipe* y la entrada estándar de `cmd2` al extremo de lectura del pipe. Gráficamente:



La llamada al sistema `pipe(p)` retorna en el argumento de salida `p` dos *file descriptors* (enteros). El primer elemento del arreglo corresponde al extremo de lectura del pipe (que usará `cmd2`) y el segundo al de escritura (usado por `cmd1`). En el siguiente ejemplo, el proceso padre envía un mensaje al hijo por un *pipe*.

c

```c
#include <unistd.h> /* UNIX syscalls */
#include <stdio.h>

#define N 20

int main(void) {
    int p[2];
    pipe(p);    /* create pipe */

    if (fork() == 0) {
        /* in child: read from pipe */
        char buffer[N];
        close(p[1]);            /* close write pipe descriptor */
        read(p[0], buffer, N);
        printf("Child read: %s", buffer);
    } else {
        /* in parent */
        close(p[0]);            /* close read pipe descriptor */
        write(p[1], "Hello from parent\n", 18);
    }
}
```

Cabe aclarar que un *pipe* es un medio de comunicación *unidireccional*. Si el escritor también lee del pipe podría consumir su propia escritura.

Para lograr una comunicación *bidireccional* debemos usar dos pipes.

El *pipe* internamente es un *buffer acotado* y el kernel implementa el clásico patrón *productor/consumidor*. El proceso que escribe puede bloquearse porque el buffer se llenó y el lector puede bloquearse si el buffer está vacíout.txt

Un escritor *bloqueado* se desbloqueará cuando el lector consuma datos del pipe. Conversamente, el lector se desbloqueará cuando el escritor produzca datos.

Se debe notar que un *pipe* sólo puede utilizarse por procesos relacionados padre/hijo. Dos procesos independientes pueden comunicarse por un *named pipe* o *FIFO*. Un *FIFO* es un archivo especial asociado a un pipe gestionado por el kernel. Uno de los procesos lo abre para escritura y el otro para lectura. Las operaciones `read` y `write` leerán o escribirán en el pipe asociado y se sincronizan de la misma manera.

Señales

Las señales son un mecanismo básico de *comunicación entre procesos* para implementar la notificación de *eventos*. El kernel en algunos casos abstrae una *interrupción* o *excepción* en una *señal* enviada a un proceso. También un proceso puede enviar una *señal* a otro proceso. Este mecanismo fue desarrollado en las primeras versiones de UNIX en la década de 1970.

La llamada al sistema `kill(pid, signal)` envía la señal `signal` al proceso identificado con `pid` . Cada señal tiene ide que sea desde ntificadores numéricos. El sistema envía señales a los procesos cuando el usuario en una terminal pulsa las combinaciones de teclas como por ejemplo `Ctrl-C` , la cual envía `SIGINT` (*interrupt*) o `Ctrl-Z` , que envía `SIGTSTP` .

Un proceso puede *manejar* una señal *s* instalando un *signal handler* por medio del syscall `signal(sig, handler)` , donde `sig` es la señal a manejar y `handler` es una función de la forma `void handler(int signal)` .

Por omisión cada señal tiene un *default handler* implementado en el kernel. Su comportamiento depende de cada señal. Por ejemplo, para la señal `SIGINT` el *default handler* finaliza el proceso, mientras que el *default handler* para `SIGTSTP` causa que el proceso se *suspenda* (queda en estado `SLEEPING`).

Cuando un proceso recibe una señal, se altera su flujo de ejecución normal y se dispara su *handler*. Luego del retorno, el proceso continúa con su ejecución en el estado que estaba antes de la recepción de la señal.

La única señal que no puede ser atrapada (manejada) es `SIGKILL` la cual se usa para *terminar* un proceso (si es que el usuario es el *usuario efectivo*, es decir, quien lo ejecutó).

Alarmas

Las llamadas al sistema `alarm(seconds)` y `setitimer(...)` permiten definir una alarma, es decir que luego de transcurrido ese tiempo, el kernel lanzará la señal `SIGALRM` al proceso.

Una alarma generalmente se usa cuando una aplicación realiza una operación (por ejemplo, enviar un mensaje) y si no tiene respuesta dentro de un intervalo de tiempo (*timeout*), requiere realizar alguna acción de recuperación o re-intento.

Trazado de procesos

Un proceso puede controlar la ejecución de otro proceso. Al primero se lo conoce como el *trazador* y al otro el *trazado*. Comúnmente un *debugger* se implementa usando este mecanismo, entre otros.

La llamada al sistema `ptrace(request, pid, addr, data)` . El proceso *trazado* se suspende cada vez que envíe una señal o haga una llamada al sistema. El *trazador* será notificado en un `wait(pid, &status)` o `waitpid(pid, &status, options)` . En `status` se indica la causa de la suspensión del proceso trazado.

Luego, el *trazador* puede realizar (mediante `ptrace()`) alguna de las siguientes acciones sobre el proceso *trazado*:

- Identificar qué llamada al sistema realizó.
- Leer o escribir en el área de código o datos. Así un debugger, puede por ejemplo, definir un *breakpoint* por software, reemplazando una instrucción por una llamada al sistema.
- Acceder al estado de la CPU (valores de los registros) del proceso bloqueado
- Hacer que continúe la ejecución
- Otras. Para más detalles, hacer `man ptrace` .

A continuación se muestra un ejemplo simple que *intercepta* la primera llamada al sistema hecha por el proceso hijo (en GNU-Linux arquitectura x86_64).

c

```c
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/user.h>
```

```c
#include <sys/reg.h>
#include <sys/syscall.h>
#include <stdio.h>

int main()
{   pid_t child;
    long rax;
    int status;
    child = fork();
    if(child == 0) {
      ptrace(PTRACE_TRACEME, 0, NULL, NULL);
      execl("/bin/ls", "ls", NULL);
    }
    else {
      while (1) {
        // wait for a child syscall
        wait(&status);
        // finish tracer if child did an exit syscall
        if (WIFEXITED(status))
          break;
        // get value of saved rax register of child process
        orig_rax = ptrace(PTRACE_PEEKUSER, child, 8 * ORIG_RAX, NULL);
        printf("Child syscall number %ld\n", orig_rax);
        // continue tracing next syscall
        ptrace(PTRACE_SYSCALL, child, NULL, NULL);
      }
   }
   return 0;
}
```

Se pueden ver los números de Linux syscalls en **https://filippo.io/linux-syscall-table/**

El comando `strace` permite *rastrear* un proceso.