Sistemas Operativos

Llamadas al sistema

Departamento de Ingeniería en Sistemas y Computación
Universidad Católica del Norte, Antofagasta.

Llamadas al sistema

- Son interfaces de programación que permiten el acceso a los servicios del SO.
- Frecuentemente son accesadas por los programas a través de API de alto nivel, en vez de invocaciones directas a ellas.

Llamadas al sistema

Las API más comunes son:

- Win32 para Windows
- POSIX para distribuciones de Linux, Mac OS X
- Java API para la JVM (Java Virtual Machine)

Llamadas al sistema

Las API más comunes son:

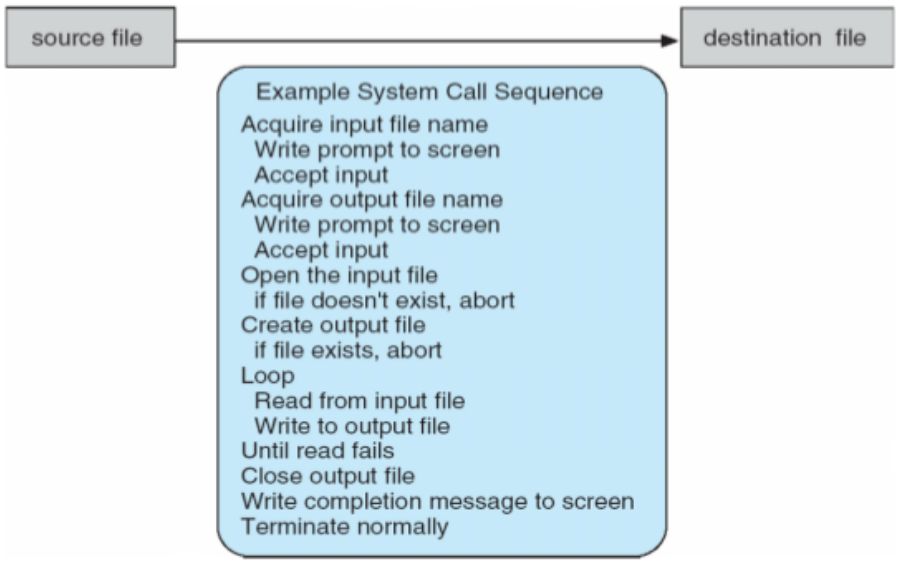
- Win32 para Windows
- POSIX para distribuciones de Linux, Mac OS X
- Java API para la JVM (Java Virtual Machine)

Llamadas al sistema

Las API más comunes son:

- Win32 para Windows
- POSIX para distribuciones de Linux, Mac OS X
- Java API para la JVM (Java Virtual Machine)

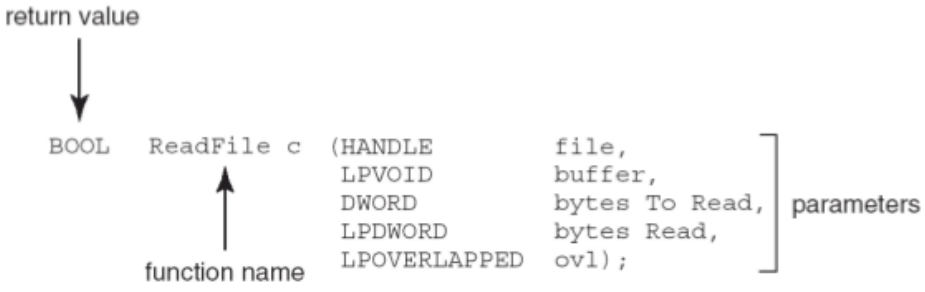
Llamadas al sistema



- Secuencia de llamadas al sistema para copiar el contenido de un archivo en otro.

Ejemplo de Win32

- ReadFile() de la API Win32, para leer desde un archivo



HANDLE file—the file to be read
LPVOID buffer—a buffer where the data will be read into and written from
DWORD bytesToRead—the number of bytes to be read into the buffer
LPDWORD bytesRead—the number of bytes read during the last read
LPOVERLAPPED ovl—indicates if overlapped I/O is being used

Implementación de llamadas al sistema

- Se asocia un número a cada llamada al sistema. La interfaz de llamadas al sistema mantiene una tabla indexada por estos números.
- La interfaz invoca a la llamada al sistema correspondiente dentro del kernel del SO y retorna el status de la llamada y un valor si corresponde.
- El llamante, no necesita saber como la llamada al sistema está implementada.

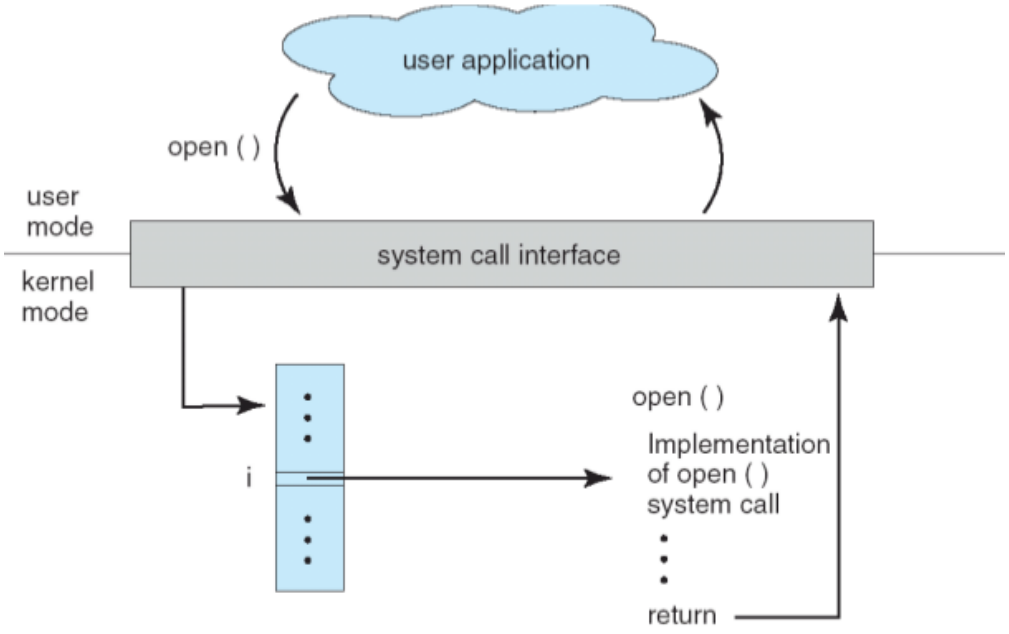
Implementación de llamadas al sistema

- Se asocia un número a cada llamada al sistema. La interfaz de llamadas al sistema mantiene una tabla indexada por estos números.
- La interfaz invoca a la llamada al sistema correspondiente dentro del kernel del SO y retorna el status de la llamada y un valor si corresponde.
- El llamante, no necesita saber como la llamada al sistema está implementada.

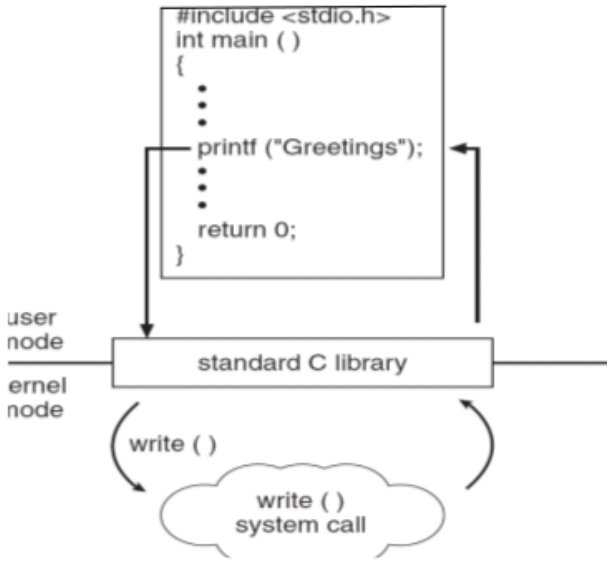
Implementación de llamadas al sistema

- Se asocia un número a cada llamada al sistema. La interfaz de llamadas al sistema mantiene una tabla indexada por estos números.
- La interfaz invoca a la llamada al sistema correspondiente dentro del kernel del SO y retorna el status de la llamada y un valor si corresponde.
- El llamante, no necesita saber como la llamada al sistema está implementada.

Esquema

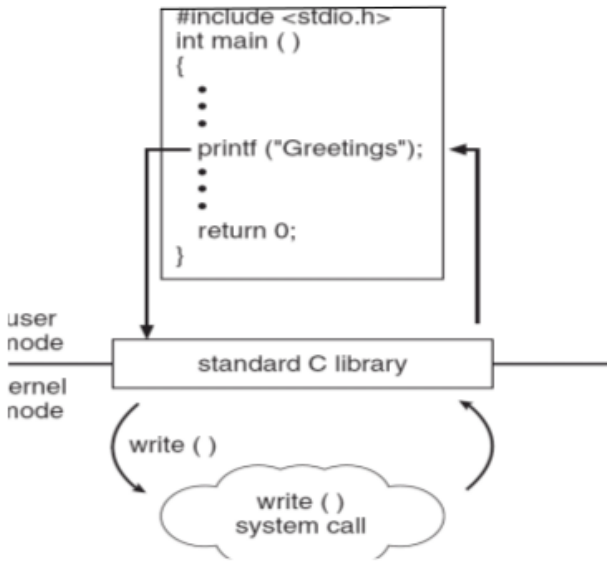

user application
open ( )
user mode
kernel mode
system call interface
i
open ( )
Implementation of open ( ) system call
return

Ejemplo biblioteca estándar C



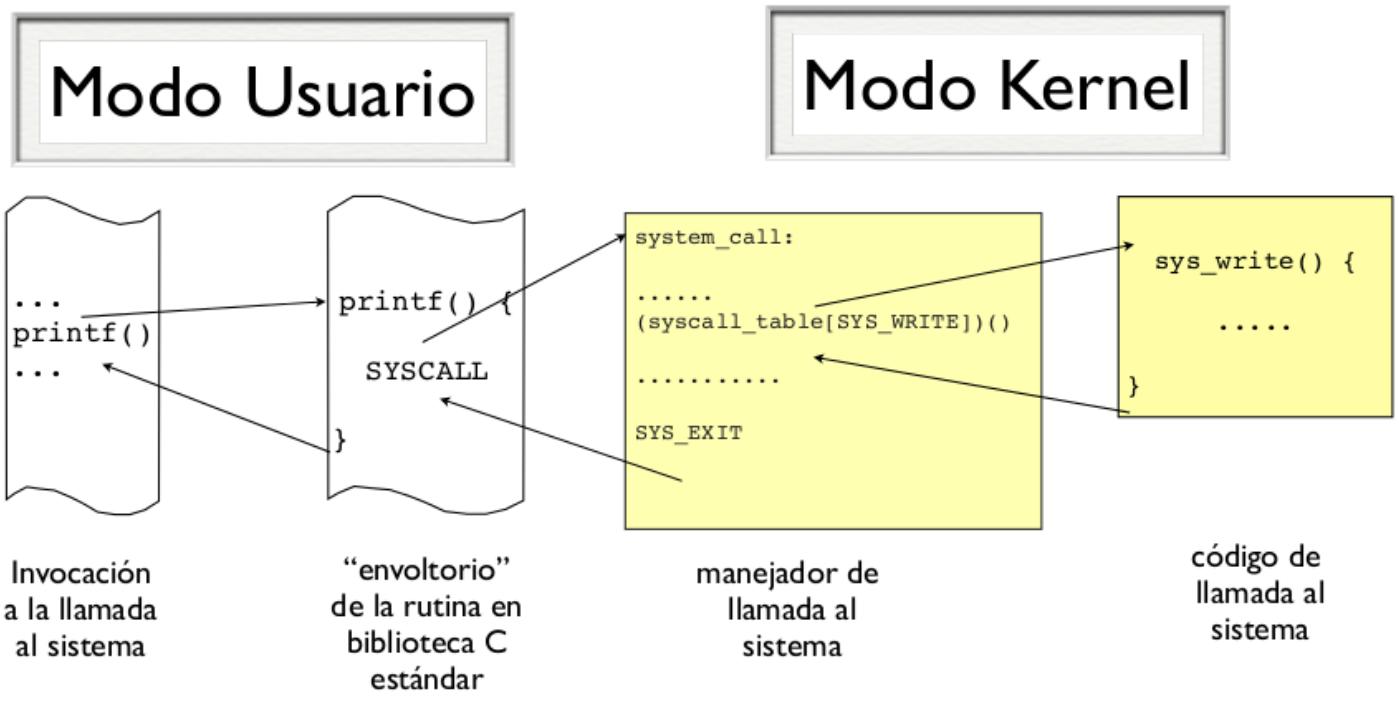
- El programa C invoca la llamada a biblioteca **printf()**.
- Esta llamada invoca la llamada al sistema **write()**

Ejemplo biblioteca estándar C



- El programa C invoca la llamada a biblioteca **printf()**.
- Esta llamada invoca la llamada al sistema **write()**

Llamadas al sistema y bibliotecas



Tipos de llamadas al sistema

- Control de procesos
- Gestión de archivos
- Gestión de dispositivos
- Mantención de la información del sistema
- Comunicaciones
- Protección

Ejemplos de Windows y UNIX

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Gestión de ficheros: creat, open, read, write, close.

- Un fichero en UNIX:
  - Es una secuencia de Byte sin formato, se accede a ellos de forma “directa”.
  - La interpretación del contenido depende de las variables que use para lectura y escritura (enteros, caracteres,..).
  - Existe un puntero que indica la posición del siguiente byte a leer o escribir, se actualiza tras cada operación.
  - Tiene asociados unos permisos: rwxrwxrwx (usuario, grupo, otros) que se representan como un número en octal
  - Ejemplo: rwxrw-r-- será: 0764

- **Creación de nuevos ficheros: creat.**
  - Sintaxis:

```
#include <fcntl.h>
int creat (nombre, mode_t permisos);
const char *nombre;
mode_t permisos;
```

  - Si existe el fichero :
    - No modifica los permisos.
    - Si tiene permiso de escritura borra el contenido.
    - Sino tiene permisos de escritura da un error.
  - Sino existe:
    - Lo crea y añade permisos de escritura.
  - Devuelve un entero:
    - Sino hay error: entre 0 y 19 (el descriptor del fichero). Devolverá el más bajo que este libre (no asociado).
    - Si hay error: un número negativo.
  - Ejemplos:

```
fd=creat("prueba",0666);
fd=creat("/usr/prácticas/hola/prueba.c",0600);
```

– **Escritura de datos en un fichero: write.**

- Sintaxis:

```
#include <unistd.h>
size_t write(desc_fich,dato,n_bytes);
int desc_fich;
const void *dato;
size_t n_bytes;
```

- Devuelve:
  - El número de caracteres que ha podido escribir (n_escritos).
  - Un valor negativo si ha ocurrido un error en la llamada.
- Ejemplos:

```
int  n_escritos, fprueba;
fprueba =creat("fprueba",0666);
n_escritos= write(fprueba,"Esto es el dato del fichero\0",28);
```

\0 es un carácter que delimita el final de una cadena

– **Cerrar un fichero: close.**

- Cuando no se va a acceder a los datos de un fichero se cierra.
- La utilidad es dejar libre un descriptor de ficheros.
- Sintaxis:

```
#include <unistd.h>
int close(descriptor_fichero);
int descriptor_fichero;
```

- Devuelve:

  0 si se ha cerrado el fichero ó -1 en caso contrario.

- **Ejemplo 2:**

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main()
{   int fd, i, vector[10];
    fd=creat("prueba",0600);
    for (i=0;i<10;i++) vector[i]=i;
    write(fd,vector,sizeof(vector));
    close(fd);
    exit(0);
}
```

- **Ejemplo 3:** escribir en prueba de 0 a 9 y después leer el contenido de prueba e imprimir en pantalla.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
int main()
{ int  fd, i, vector[10], dato, leidos;
  fd= creat("prueba",0600);
  for (i=0;i<10;i++) vector[i];
  write(fd,vector,sizeof(vector));
  close(fd);
  fd= open("prueba",O_RDONLY);
  while ((leidos= read(fd,&dato,sizeof(int)))>0)
  { printf("Leido el número %d\n",dato); }
  close(fd);
  exit(0);
}
```

Gestión de ficheros y directorios: remove, rename, lseek, stat, mkdir, rmdir y chdir.

- **Borrar un fichero: remove.**
  - Sintaxis:

```
#include <stdio.h>
int  remove (Filename);
const char *Filename;
```

  - Borra el fichero llamado *Filename*
  - El fichero no se borrará si está abierto cuando se utiliza la llamada remove.
  - Si el fichero tiene varios links (enlaces duros) la cuenta de estos desciende en uno.
  - Devuelve:
    - cero en caso de que se haya podido borrar el fichero.
    - un valor no nulo en caso contrario.

Cambio el nombre de un directorio o fichero: rename.

- Sintaxis:

```
#include <stdio.h>
int  rename (FromPath, ToPath);
const char * FromPath, ToPath;
```

- *FromPath* identifica el fichero o directorio cuyo nombre se cambia.
- *ToPath* identifica el nuevo path.
- *FromPath*, *ToPath* deben ser ambos del mismo tipo (fich, o direc).
- Si *ToPath* es fichero o directorio vacío se reemplaza por *FromPath*.
- Si es directorio no vacío no cambia el nombre y da error.
- Devuelve:
  - -1 en caso de error.
  - 0 en caso contrario.
- Errores:
  - Si no se tienen permisos de ejecución en los directorios de los caminos (paths), permisos de escritura en los directorios o ficheros oportunos, etc.
  - Si algún parámetro está siendo usado al mismo tiempo por el sistema.
  - Si *ToPath* especifica un directorio no vacío, *FromPath* es directorio y *ToPath* no ó *FromPath* es fichero y *ToPath* no.

Mover puntero de lectura o escritura de un fichero abierto : lseek.

Sintaxis:

```
#include <unistd.h>
off_t lseek ( FileDescriptor, Offset, Whence)
int FileDescriptor, Whence;
off_t Offset;
```

- Posiciona el puntero de un fichero abierto cuyo descriptor sea *FileDescriptor*.
- ***FileDescriptor*** especifica el descriptor de un fichero abierto con la llamada open.
- ***Offset*** especifica el valor en bytes que se desplazará el puntero. Un valor negativo mueve en dirección inversa. El valor de *offset* está limitado por OFF_MAX.
- ***Whence*** especifica cómo interpretar *Offset* para mover el puntero del fichero especificado por *FileDescriptor*. Será uno de los siguientes valores que están definidos en el fichero /usr/include/unistd.h:

- **SEEK_SET** (ó 0) Mueve el puntero a la posición indicada por Offset.
- **SEEK_CUR** (ó 1) El Offset se usa como desplazamiento relativo desde la posición actual del puntero. La posición final del puntero será (actual + Offset).
- **SEEK_END** (ó 2) El Offset se usa como desplazamiento relativo desde el final del fichero. La posición final del puntero será (final de fichero + Offset).

- Devuelve:
  - Devuelve la localización final del puntero en bytes medida desde el inicio del fichero.
  - Devuelve -1 en caso de error. .
- Errores:
  - Si *FileDescriptor* no corresponde a un fichero abierto.
  - Si *FileDescriptor* corresponde a una tubería abierta.
  - Si el offset sobrepasa el límite permitido definido en OFF_MAX.
  -
- En el fichero **/usr/include/unistd.h** están definidos los macros, tipos y subrutinas.

Obtiene información referente a un fichero : stat.

- Sintaxis:

```
#include <sys/stat.h>
int stat ( Path, Buffer )
const char  *Path;
struct stat  *Buffer;
```

- Obtiene información referente a un fichero del cual damos su path.
- *Path* especifica el nombre del fichero.
- *Buffer* es un puntero a la estructura **stat** en el que se devuelve la información.
- No son necesarios permisos de lectura, escritura o ejecución para el fichero.
- En la ruta del *Path* todos los directorios deben tener permisos de ejecución.
- Detallamos aquí sólo los campos de la estructura **stat** más importantes.

- Devuelve:
  - -1 en caso de error.
  - 0 en caso contrario.
- Errores:
  - Permiso denegado en alguno de los directorios del path.
  - Un componente del path no es un directorio.
  - No existe el fichero.

- La estructura stat está definida en el fichero /usr/include/sys/stat.h.

- Los valores de algunos campos de stat están definidos en anubis en el fichero */usr/include/sys/mode.h* y para **linux** ver: *man 2 stat*

- Los tipos de datos están definidos en el fichero /usr/include/sys/types.h.

Crea un directorio: mkdir.

- Sintaxis:

```
#include <stdio.h>
int  mkdir (Path, Mode);
const char * Path;  mode_t  Mode;
```

- Crea un nuevo directorio con los bits de permisos indicados en Mode.
- *Path* especifica el nombre del nuevo directorio.
- *Mode* especifica la máscara de lectura, escritura y ejecución para usuario, grupo y otros con la que se creará el nuevo directorio (en octal).
- Devuelve:
  - -1 en caso de error.
  - 0 en caso contrario.
- Errores:
  - No se tienen los permisos adecuados para crear el directorio.
  - El nombre dado existe como fichero.
  - La longitud del *Path* es demasiado larga.
  - Algún componente del *Path* no es un directorio o no existe.

Borra un directorio: rmdir.

- Sintaxis:

```
#include <stdio.h>
int  rmdir (Path);
const char * Path;
```

- Borra el directorio especificado en *Path*. Para ello hay que tener permiso de escritura en el directorio padre de *Path*.
- *Path* especifica un nombre de directorio.
- Devuelve:
  - -1 en caso de error.
  - 0 en caso contrario.
- Errores:
  - No se tiene permiso de escritura en el directorio padre de *Path*.
  - El directorio no está vacío.
  - La longitud de *Path* excede la permitida.
  - No existe el directorio.
  - Uno componente de *Path* no es directorio.
  - El directorio se refiere a un punto en el que hay montado un dispositivo.

Ejercicio

Implemente un programa en C que muestre por pantalla la extensión de todos los archivos presentes en la ubicación donde está almacenado el ejecutable.

Ejercicio

Implemente un programa en C que pida al usuario ingresar el nombre de cierta extensión, el programa debe crear una carpeta con ese nombre y todos los archivos de dicha extensión presentes en la ubicación donde está almacenado el ejecutable deben ser movidos a la nueva carpeta creada.