El Sistema Operativo XV6 - Capitulo 1

Carlos Santiago Sandoval Casallas

csandovalc@unal.edu.co

Departamento de Ingeniería de Sistemas e Industrial
Universidad Nacional de Colombia

7 de mayo de 2025

UNIVERSIDAD NACIONAL DE COLOMBIA

Agenda

1. Interfaces del SO
2. Procesos y memoria
3. E/S y descriptores de archivo
4. Tuberías
5. Sistemas de archivos
   - Mundo real

UNIVERSIDAD NACIONAL DE COLOMBIA

¿Cuál es la finalidad de un sistema operativo?

Los objetivos principales de los sistemas operativos son:

- Gestionar los recursos físicos y simplificar su administración para otros programas
  Un procesador de textos no necesita preocuparse por el tipo de disco empleado
- Soportar la ejecución de múltiples programas en paralelo
  Un editor de texto y un navegador ejecutándose al mismo tiempo
- Proporcionar métodos controlados para la interacción entre programas
  Tuberías, paso de mensajes

UNIVERSIDAD NACIONAL DE COLOMBIA

Diseño de interfaces

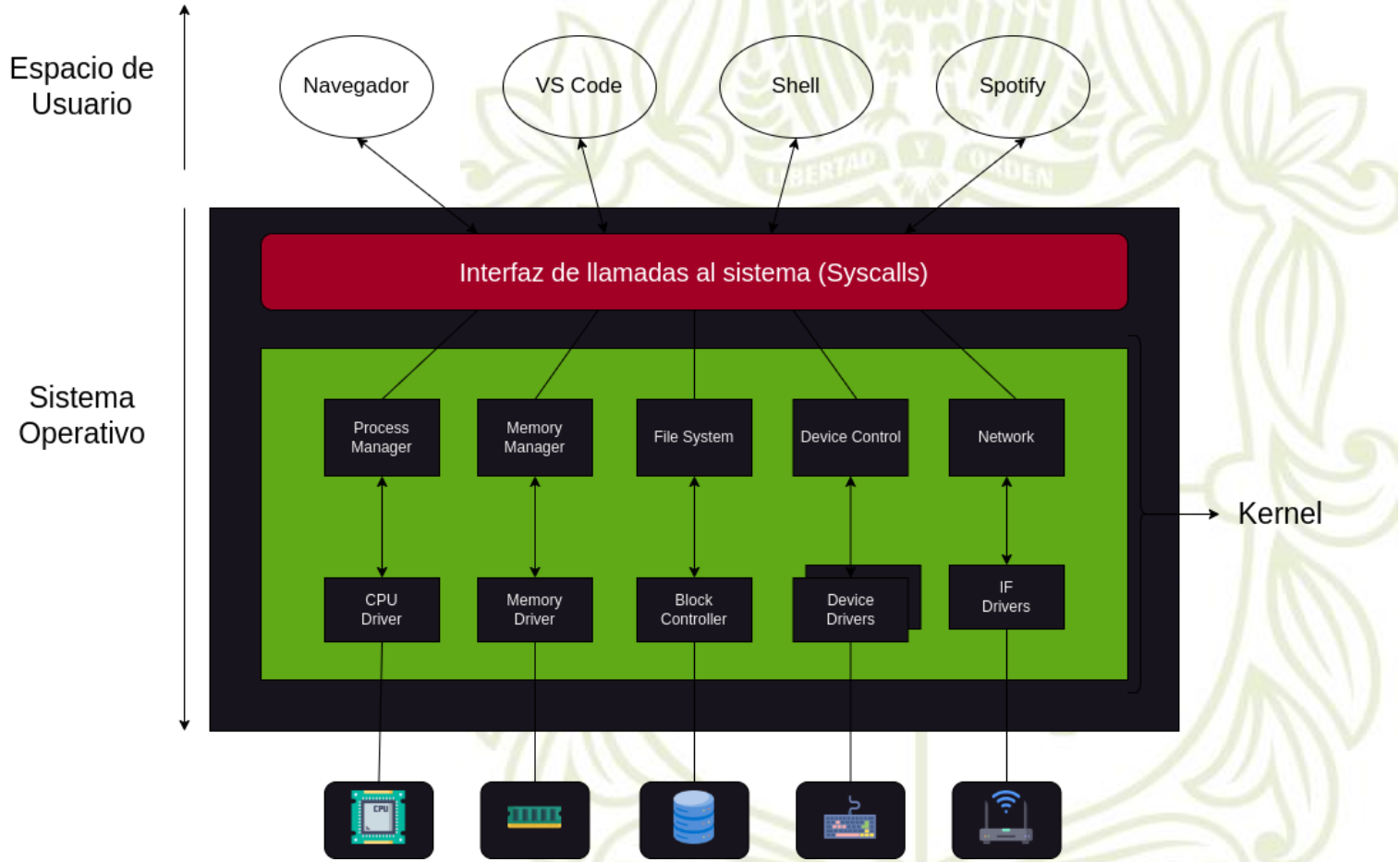
El concepto de interfaz es empleado para hacer referencia a una conexión funcional entre sistemas, programas, dispositivos, etc. [1]

El diseño de estas interfaces se basa en crear pocos mecanismos que sean combinables entre sí para cumplir necesidades más complejas, proporcionando generalidad y facilitando la implementación.

En este caso se trabajará sobre XV6, un sistema operativo educativo desarrollado y mantenido por el MIT. Este sistema proporciona las interfaces básicas basadas en el sistema operativo Unix V6 [2], además de imitar su diseño.

UNIVERSIDAD NACIONAL DE COLOMBIA

Figura: Estructura de sistema operativo con su espacio de usuario



Fuente: Elaboración propia. Iconos obtenidos del sitio Flaticon.com

UNIVERSIDAD NACIONAL DE COLOMBIA

Procesos y el kernel

Cada programa en ejecución tiene el nombre de proceso, el kernel o núcleo, es un proceso especial que brinda servicios a los programas en ejecución, una computadora típicamente tiene muchos proceso pero solo un kernel.

Cuando un proceso necesita emplear uno de los servicios proporcionados por el kernel, invoca una llamada al sistema o *syscall*, la llamada ingresa al espacio de kernel, el kernel realiza el servicio solicitado y regresa al programa desde donde ha sido invocado.

Un proceso alterna entre ejecutarse en el espacio de usuario y el espacio del kernel

El kernel emplea algunos mecanismos de protección proporcionados por el hardware, para garantizar que un proceso que se ejecuta en el espacio de usuario pueda acceder solo a su propia memoria. Mientras que el kernel ejecuta instrucciones con estos privilegios sobre el hardware para implementar dichas protecciones.

Cuando un programa desde el espacio de usuario invoca una syscall, el hardware aumenta el nivel de privilegio para ejecutar la función **ya preestablecida en el kernel**.

User mode, supervisor mode, and system calls

Esto se explorará con mayor profundidad en el capítulo 2.2

Figura: Llamadas al sistema disponibles en XV6. Si no se indica lo contrario retornan -1 si hay error y 0 en otro caso.

System call	Description
int fork()	Create a process, return child's PID.
int exit(int status)	Terminate the current process; status reported to wait(). No return.
int wait(int *status)	Wait for a child to exit; exit status in *status; returns child PID.
int kill(int pid)	Terminate process PID. Returns 0, or -1 for error.
int getpid()	Return the current process's PID.
int sleep(int n)	Pause for n clock ticks.
int exec(char *file, char *argv[])	Load a file and execute it with arguments; only returns if error.
char *sbrk(int n)	Grow process's memory by n bytes. Returns start of new memory.
int open(char *file, int flags)	Open a file; flags indicate read/write; returns an fd (file descriptor).
int write(int fd, char *buf, int n)	Write n bytes from buf to file descriptor fd; returns n.
int read(int fd, char *buf, int n)	Read n bytes into buf; returns number read; or 0 if end of file.
int close(int fd)	Release open file fd.
int dup(int fd)	Return a new file descriptor referring to the same file as fd.
int pipe(int p[])	Create a pipe, put read/write file descriptors in p[0] and p[1].
int chdir(char *dir)	Change the current directory.
int mkdir(char *dir)	Create a new directory.
int mknod(char *file, int, int)	Create a device file.
int fstat(int fd, struct stat *st)	Place info about an open file into *st.
int stat(char *file, struct stat *st)	Place info about a named file into *st.
int link(char *file1, char *file2)	Create another name (file2) for the file file1.
int unlink(char *file)	Remove a file.

Fuente: Tomado de xv6: a simple, Unix-like teaching operating system [2]

UNIVERSIDAD NACIONAL DE COLOMBIA

Aplicación de las llamadas al sistema

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  int pid = fork();
  if (pid > 0) {
    printf("parent: child=%d\n", pid);
    pid = wait((int *) 0);
    printf("child %d is done\n", pid);
  } else if (pid == 0) {
    printf("child: exiting\n");
    exit(0);
  } else {
    printf("fork error\n");
  }
}
```

code/syscall_example.c

Las llamadas al sistema empleadas son:

- **fork:** Crea un nuevo proceso, este es un duplicado del proceso que emplea la llamada del sistema, el nuevo proceso es referido como *proceso hijo*. El proceso que empleo la syscall es referido como *proceso padre*.
- **wait:** Retorna el PID del proceso hijo que ha sufrido de un cambio de estado, estos pueden ser que el hijo: ha salido (exit), ha sido eliminado (killed), ha sido detenido o reanudado por una señal, este copia el estado de salida del proceso hijo.
- **exit:** Terminar la ejecución del proceso que lo llama, liberando recursos como memoria y archivos abiertos. Este toma un argumento de tipo entero, este es el *status*, al momento de invocar la syscall, se retorna el byte menos significativo a su proceso padre.

Toda la información puede ser encontrada en las man pages de Linux

UNIVERSIDAD NACIONAL DE COLOMBIA

Fork

Fork crea una réplica del proceso, donde tanto el proceso padre como el hijo poseen el mismo contenido en memoria, esta se encuentra en lugares diferentes de la memoria y emplean registros separados, por lo que cambiar una variable en un proceso, no afecta al otro.

Shell

El Shell es un programa ordinario que lee los comandos del usuario y los ejecuta. Este se ejecuta en el espacio de usuario, su funcionamiento es apoyado principalmente por las syscall de *fork* y *exec*.

- **exec:** Reemplaza la memoria del proceso que llama con una nueva imagen de memoria cargada desde un archivo almacenado en el sistema de archivos. El archivo debe tener un formato particular, XV6 emplea el formato ELF.

Formato ELF

Este se verá con mayor detalle en el capítulo 3.8

UNIVERSIDAD NACIONAL DE COLOMBIA

Ejemplo de Exec

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

int main() {
  char *argv[3];
  argv[0] = "echo";
  argv[1] = "hello";
  argv[2] = 0;

  execv("/bin/echo", argv);
  printf("Error de ejecucion\n");
  return 0;
}
```

code/exec_example.c

En el código anterior, se reemplaza el programa con una instancia del binario en */bin/echo* que se ejecuta con la lista de argumentos *argv*.

El bucle principal de la Shell ***(user/sh.c:146)*** [3] lee el input del usuario con el método *getcmd*. Se emplea un fork, el padre espera, mientras que el proceso hijo ejecuta el comando. Si exec tiene éxito, el proceso hijo ejecutará las instrucciones del binario ejecutado, cuando estos finalicen, el padre finaliza la espera y continua con la ejecución.

El programa del Shell aprovecha la separación en su implementación de la redirección de E/S. Esto para evitar el desperdicio de crear un proceso duplicado y luego reemplazarlo de inmediato.

Asignación de memoria del espacio de usuario

XV6 asigna la mayor parte de la memoria del espacio de usuario implícitamente; fork asigna la memoria requerida para la copia del proceso, mientras exec asigna suficiente memoria para contener el ejecutable. En caso de requerir más memoria en tiempo de ejecución, se puede emplear *sbrk(n)* para aumentar la memoria en *n* bytes.

sbrk

Este se verá con mayor detalle en el capítulo 3.7

UNIVERSIDAD NACIONAL DE COLOMBIA

Descriptor de Archivo

Un descriptor es un entero que representa un objeto administrado por el kernel, este puede ser empleado para administrar: un archivo, directorio o dispositivo. La interfaz del descriptor de archivos, el objeto para que sea manipulado como un flujo de entrada o salida de bytes.

El kernel de XV6 asigna el valor del descriptor a un índice de una tabla *por proceso*, permitiendo que cada proceso tenga un espacio privado de descriptores que inician en cero.

Existe una convención respecto a ciertos descriptores, estos son:

- **0:** Entrada estándar
- **1:** Salida estándar
- **2:** Error estándar

UNIVERSIDAD NACIONAL DE COLOMBIA

Aplicación de los descriptores

El Shell, se encarga de emplear los descriptores estándar, de esta manera puede usarlos en la canalización (uso de tuberías) o redirección de E/S. Para esto el Shell verifica tener estos descriptores abiertos ***(user/sh:152)***. [3]

Como emplear estos descriptores mediante las interfaces del sistema:

- *read(fd, buff, n):* ***fd*** es el descriptor del objeto sobre el cual se quiere leer, ***buff*** la referencia en donde se almacenaran los ***n*** bytes leídos.
- *write(fd, buff, n):* ***fd*** es el descriptor del objeto sobre el cual se quiere escribir, ***buff*** la referencia de donde salen los ***n*** bytes a escribir.

Cada archivo referenciado por un descriptor, tiene un desplazamiento asociado

UNIVERSIDAD NACIONAL DE COLOMBIA

Ejemplo sobre el uso de descriptores estándar

Una implementación simple que sobre la lectura y escritura estándar:

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

int main() {
  char buf[512];
  int n;

  for (;;) {
    n = read(0, buf, sizeof buf);
    if (n == 0)
      break;
    if (n < 0) {
      fprintf(2, "read error\n");
      exit(1);
    }
    if (write(1, buf, n) != n) {
      fprintf(2, "write error\n");
      exit(1);
    }
  }
  return 0;
}
```

code/descriptors_example.c

UNIVERSIDAD NACIONAL DE COLOMBIA

Close

La llamada al sistema *close*, libera un descriptor de archivo, lo que permite su reutilización para otra llamada al sistema, una canalización (pipe) o duplicación (dup). Un descriptor de archivo recién asignado siempre será, el número más bajo disponible **(no utilizado)** en el proceso actual.

Los descriptores tienen ciertas interacciones con *fork* y *exec*. *Fork* copiará la tabla de descriptores, por lo tanto, ambos procesos tendrán acceso a los mismos archivos. *Exec* por su parte, reemplaza la memoria del proceso con la nueva imagen solicitada, **pero conservará la tabla de archivos**.

Esto permite que el Shell implemente la redirección de E/S usando fork y posteriormente empleando exec para ejecutar el nuevo programa.

UNIVERSIDAD NACIONAL DE COLOMBIA

Redirección mediante syscalls

El siguiente programa ejemplifica el comando *cat < input.txt*

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>


int main() {
  char *argv[2];
  argv[0] = "cat";
  argv[1] = 0;

  if (fork() == 0) {
    close(0); // Se libera el descriptor 0 (entrada estandar)

    // Se abre el archivo del cual leeremos el mensaje este tendra asignado
    // el descriptor mas bajo posible, en este caso 0.
    open("input.txt", O_RDONLY);
    execv("/bin/cat", argv);
  }
}
```

code/cat_example.c

Explicación

En el código anterior, el proceso hijo cierra el descriptor 0, de esta manera garantizamos que al momento de abrir el archivo *input.txt* este se le asigne el descriptor más pequeño posible, posteriormente *cat* se ejecuta, con el descriptor 0, pero en este caso hará referencia al archivo *input.txt* y no a la entrada estándar. **Este proceso no afectará los descriptores del proceso padre**

Esto se puede ver en profundidad en ***(user/sh.c:83)*** [3], al momento de ejecutarse este código, ya se ha bifurcado el Shell y *runcmd* llamará a *exec* para cargar el nuevo programa.

En la función open, el segundo argumento consiste en un conjunto de banderas expresadas en bits, que varían el funcionamiento de *open*. Estas banderas se definen en ***(kernel/fcntl.h)***. [3]

UNIVERSIDAD NACIONAL DE COLOMBIA

Fork y Exec

Con la redirección queda claro porque *fork* y *exec* son llamadas separadas, y es que esto permite redirigir la E/S del proceso hijo sin alterar la configuración de E/S del proceso principal.

Si imaginamos una llamada combinada (por ejemplo, *forkexec*), el Shell podría modificar su propia configuración de E/S antes de llamar a *forkexec*, pero posteriormente tendría que deshacer estas modificaciones; o *forkexec* podría tomar instrucciones para la redirección como argumentos; o a cada programa emplea su propia redirección de E/S.

UNIVERSIDAD NACIONAL DE COLOMBIA

Duplicación

Es posible duplicar un descriptor mediante la llamada al sistema de ***dup***, este toma como argumento un descriptor existente y retorna uno nuevo que hace referencia al mismo objeto. Ambos descriptores comparten un desplazamiento.

Desplazamiento con procesos bifurcados

Aunque *fork* copia la tabla de descriptores de archivo, el desplazamiento del archivo **se comparte entre los procesos**.

Por lo tanto, los descriptores comparten desplazamiento si se derivan del mismo descriptor, ya sea por la bifurcación de un proceso, o la duplicación de un descriptor. De lo contrario, los descriptores no comparten dicho desplazamiento, incluso si son el resultado de la llamada *open* para el mismo archivo.

UNIVERSIDAD NACIONAL DE COLOMBIA

Ejemplo de Dup

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

int main() {
  int fd = dup(1);
  write(fd, "Hello", 6);
  write(fd, " World\n", 8);

  return 0;
}
```

code/dup_example.c

Pipe

Una tubería es un pequeño búfer del kernel que es accesible para los procesos, esto mediante un par de descriptores de archivo, uno para leer y otro para escribir. La tubería se comporta como una lista tipo FIFO. Estas proporcionan un método de comunicación entre procesos.

UNIVERSIDAD NACIONAL DE COLOMBIA

Uso de la tubería

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

int main() {
  int p[2]; // Arreglo donde almacenaremos los descriptores
  char *argv[2];
  argv[0] = "wc";
  argv[1] = 0;

  pipe(p);
  if (fork() == 0) {
    // El descriptor 0 referenciara al descriptor de lectura de la tuberia
    close(0);
    dup(p[0]);

    // Cierra la tuberia
    close(p[0]);
    close(p[1]);

    execv("/bin/wc", argv);
  } else {
    close(p[0]); // Cierra el descriptor de lectura

    // Escribe sobre la tuberia y cierra el descriptor de escritura
    write(p[1], "Hello World\n", 12);
    close(p[1]);
  }
}
```

code/pipe_example.c

UNIVERSIDAD NACIONAL DE COLOMBIA

Lectura y escritura sobre la tubería

Además de la ya mencionada característica de flujo FIFO en los datos, la tubería presenta más particularidades; si la tubería está vacía, la operación de lectura se vuelve bloqueante, mientras que si la tubería está llena, la operación de escritura se vuelve bloqueante.

Operaciones bloqueantes

Esto hace referencia a que el proceso detiene su ejecución, hasta que el sistema operativo le indique mediante una señal que debe reanudarse.

Esta señal puede ser disparada porque la tubería ya cumple la condición para ejecutar la operación deseada o todos los descriptores de archivo que hacen referencia a la tubería han sido cerrados; en este último caso, la operación retornara 0.

UNIVERSIDAD NACIONAL DE COLOMBIA

Pipes en el Shell

Cuando se emplean tuberías en el Shell (por ejemplo: **ls | grep home**), en este caso el proceso secundario crea una tubería para conectar el extremo izquierdo con el derecho. Luego llama a *fork* y *runcmd* para ambos extremos y espera que ambos terminen.

En caso de que el extremo derecho pueda ser un comando que incluya una pipe (por ejemplo: **a | b | c**), este a su vez bifurcara dos nuevos procesos secundarios (uno para b y otro para c). De esta manera, el Shell puede crear un árbol de procesos; las hojas son comandos, los nodos interiores son procesos que esperan hasta que los hijos izquierdo y derecho se completen.

Tuberías vs archivos temporales

- Las tuberías se limpian automáticamente, en caso de emplear archivos, se debe tener cuidado con eliminar el archivo temporal cuando haya terminado.
- Las tuberías pueden pasar flujos de datos arbitrariamente largos, mientras que el uso de archivos requiere contar con el espacio en disco para almacenar todos los datos.
- Las tuberías permiten la ejecución paralela de las etapas de la tubería, el uso de archivos requiere que el primer programa finalice antes de que comience el segundo.

UNIVERSIDAD NACIONAL DE COLOMBIA

Estructura de directorios

El sistema de archivos de XV6 proporciona archivos de datos, estos contienen arreglos de bytes no interpretados, y directorios, estos contienen referencias nombradas hacia archivos y otros directorios.

Los directorios poseen una estructura en forma de árbol, este empieza en un directorio especial llamado raíz. Las rutas que **NO** comienzan con “/” se evalúan en relación con el directorio actual del proceso, este puede cambiar con la syscall *chdir*.

Uso de chdir

En caso de que exista la ruta “/a/b/c”, se presentan las dos posibilidades de acceder al archivo c.

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
  chdir("/a");
  chdir("b");
  open("c", O_RDONLY);

  open("/a/b/c", O_RDONLY);
}
```

code/chdir_example.c

Creación de archivos y directorios

Existen llamadas al sistema para crear archivos y directorios: ***mkdir*** crea un nuevo directorio, ***open*** con el indicador de ***O_CREATE*** crea un nuevo archivo de datos y ***mknod*** crea un archivo de dispositivo.

mknod crea un archivo especial que referencia un dispositivo, asociados al archivo de dispositivo, están los números de dispositivo mayor y menor, **sus dos argumentos**, estos identifican de manera única un dispositivo del kernel.

Cuando un proceso abre posteriormente un archivo de dispositivo, el kernel *desvía* las syscall de lectura y escritura, a la implementación del dispositivo que maneja el kernel, en lugar de pasarlas al sistema de archivos.

UNIVERSIDAD NACIONAL DE COLOMBIA

Métodos de creación de archivos y directorios

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

int main() {
  // El segundo argumento es el modo (ver inode(7))
  mkdir("/dir", 0700);

  int fd = open("/dir/file", O_CREAT|O_WRONLY);
  close(fd);

  mknod("/console", 1, 1);
}
```

code/mkdir_example.c

Inode y nombres de archivos

El nombre de un archivo es diferente al archivo en sí, un archivo es representado a través de un ***inodo***, este es una estructura de datos que almacena metadatos del archivo, como tamaño, permisos, número de nombres, su tipo (archivo, directorio, dispositivo), su ubicación en disco y otras características dependiendo de la implementación. Estos inodos son identificados mediante un número unico.

El nombre de un archivo es denominado enlace, cada enlace consta de una entrada en un directorio, esta entrada contiene el nombre del archivo y referencia al inodo.

Recuperar información del inode

Es posible recuperar la información de un inodo al que hace referencia un descriptor de archivo, para esto se implementa la llamada al sistema ***fstat***. La syscall toma la referencia de una estructura de tipo ***stat*** y carga la información del archivo, la estructura se define en ***(kernel/stat.h:5)*** [3].

```
1 struct stat {
2   int dev;      // File system's disk device
3   uint ino;     // Inode number
4   short type;   // Type of file
5   short nlink; // Number of links to file
6   uint64 size; // Size of file in bytes
7 };
```

code/stat.h

Enlazar

La syscall ***link*** crea otro nombre de archivo que hace referencia al mismo inodo de un archivo existente. Esto incrementará en 1 el número de nombres almacenado en el inodo.

```
1 open("a", O_CREATE|O_WRONLY);
2 link("a", "b");
```

code/link.c

El leer o escribir sobre ***a*** es igual a escribir en ***b***.

Desenlazar

La syscall ***unlink*** eliminará un nombre del sistema de archivos, esto decrementará en 1 el número de nombres almacenado en el inodo, el inodo del archivo y el espacio en disco que almacena su contenido solo se liberara cuando el número de enlaces o nombres del archivo sea cero, adicionalmente no debe haber descriptores de archivo que hagan referencia a él.

```
1 unlink("a");
2
3 // Crear un inodo temporal que se limpie
4 // cuando se cierre el descriptor
5 // o finalice el proceso
6 fd = open("/tmp/xyz", O_CREATE|O_RDWR);
7 unlink("/tmp/xyz");
```

code/unlink.c

La combinación de descriptores de archivos “estándar”, tuberías y una sintaxis de Shell conveniente para operaciones proporciono un gran avance en la escritura de programas reutilizables de propósito general [2]. Gracias a esto, la interfaz de llamada del sistema Unix se estandarizó a través de **POSIX**, **XV6 no es compatible con POSIX**, esto dado que el objetivo del sistema operativo es la simplicidad y la claridad.

Los kernels modernos proporcionan muchas más syscalls y otros servicios, adicionalmente evolucionan continua y rápidamente, incluso proporcionan herramientas más allá de POSIX. Después de estudiar XV6, debería poder observas otros sistemas operativos más complejos y reconocer los conceptos presentes en XV6.

UNIVERSIDAD NACIONAL DE COLOMBIA

Referencias I

[1] Wikipedia. **Interfaz — Wikipedia, The Free Encyclopedia**. http://es.wikipedia.org/w/index.php?title=Interfaz&oldid=152027930. [Online; accessed 25-June-2023]. 2023.

[2] Cox, Kaashoek, Morris. **xv6: a simple, Unix-like teaching operating system.** 2022. URL: https://github.com/mit-pdos/xv6-riscv-book.

[3] MIT PDOS. **xv6-riscv**. 25 de ago. de 2022. URL: https://github.com/mit-pdos/xv6-riscv.

UNIVERSIDAD NACIONAL DE COLOMBIA

Gracias por la atención

Contacto:

csandovalc@unal.edu.co

UNIVERSIDAD NACIONAL DE COLOMBIA