



Capítulo 2

Estructuras del Sistema Operativo



Anuncios

- Hoy publicaré en SidWeb:
 - Lección 1 → la pueden tomar hasta el próximo martes
 - Laboratorio 1 → lo pueden completar hasta el próximo jueves

Contenido

- Servicios del sistema operativo
- Interfaz de usuario
- Llamadas al sistema
- Tipos de llamadas al sistema
- Programas del sistema
- Diseño e implementación de S.O.

Objetivos

- Describir los servicios que un sistema operativo proporciona a los usuarios, procesos y otros sistemas
- Discutir las diferentes maneras de estructurar un sistema operativo

ACM Curriculum Guidelines: Core Areas

- <https://www.acm.org/education/CS2013-final-report.pdf>
- Core Areas:
 - Systems Fundamentals (SF)
 - Operating Systems (OS)

Servicios del Sistema Operativo: Clasificación

Funciones útiles para

el usuario

los desarrolladores

el sistema

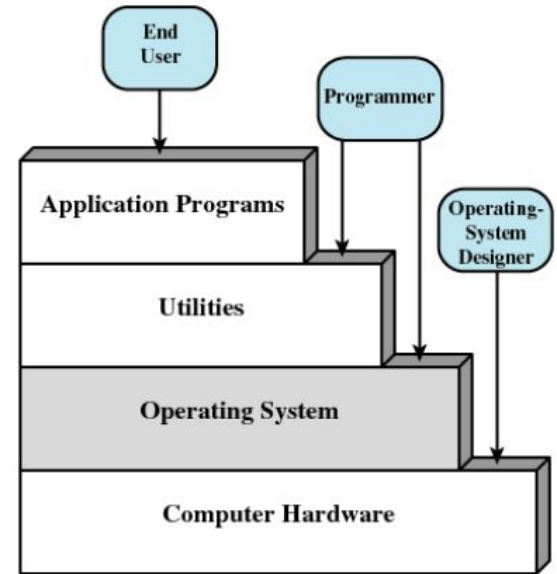


Figure 2.1 Layers and Views of a Computer System

Servicios para el usuario

- Interfaz de usuario
- Ejecución de programas
- Operaciones de E/S
- Manipulación del sistema de archivos
- Comunicaciones (en la misma máquina o en la red)
- Detección de errores
 - Incluye mecanismos reporte de errores

Servicios para desarrolladores

- Editores
- Debugging
- Llamadas al sistema ← Interfaz de desarrollo

Servicios para el sistema

- Asignación de recursos
- Contabilidad
- Protección y seguridad
 - **Protección:** asegurar que el acceso a los recursos del sistema esté controlado
 - **Seguridad:** autenticación, etc.




Servicio: Interfaces de usuario



Interfaces de usuario

- Históricamente, UIs muy diferentes
- CLI (Command-line interface)
 - En el kernel o programas del sistema
 - **shell**: cuando hay múltiples versiones
 - ¿Ejemplos?
- GUI (Graphical User Interface)
 - Metáfora de ventanas
 - Inventado en el Xerox PARC
 - ¿Ejemplos?




```
DISK OPERATING SYSTEM II VERSION 2.5
COPYRIGHT 1984 ATARI CORP.


A. DISK DIRECTORY      I. FORMAT DISK
B. RUN CARTRIDGE       J. DUPLICATE DISK
C. COPY FILE           K. BINARY SAVE
D. DELETE FILE(S)      L. BINARY LOAD
E. RENAME FILE         M. RUN AT ADDRESS
F. LOCK FILE           N. CREATE MEM.SAV
G. UNLOCK FILE         O. DUPLICATE FILE
H. WRITE DOS FILES     P. FORMAT SINGLE

SELECT ITEM OR RETURN FOR MENU
A
DIRECTORY--SEARCH SPEC,LIST FILE?
D1:
    DOS      SYS 037
    DUP      SYS 042
628 FREE SECTORS

SELECT ITEM OR RETURN FOR MENU
█
```



Servicio: Interfaz de desarrollo (llamadas al sistema)



Llamadas al sistema

- Interfaz de programación para los servicios proporcionados por el S.O.
- Generalmente escritas en un lenguaje de alto nivel (C/C++) + assembler
- Principalmente accesadas por programas via un **API** de alto-nivel (en lugar de ser invocadas directamente)
 - ¿Por qué?
- Tres APIs más comunes:
 - WinAPI (Windows, ej: Win32, Win64, WinCE)
 - POSIX (UNIX, Linux, Mac OS X)
 - Java API (JVM)
- Ejemplo:
 - `open(filename)`

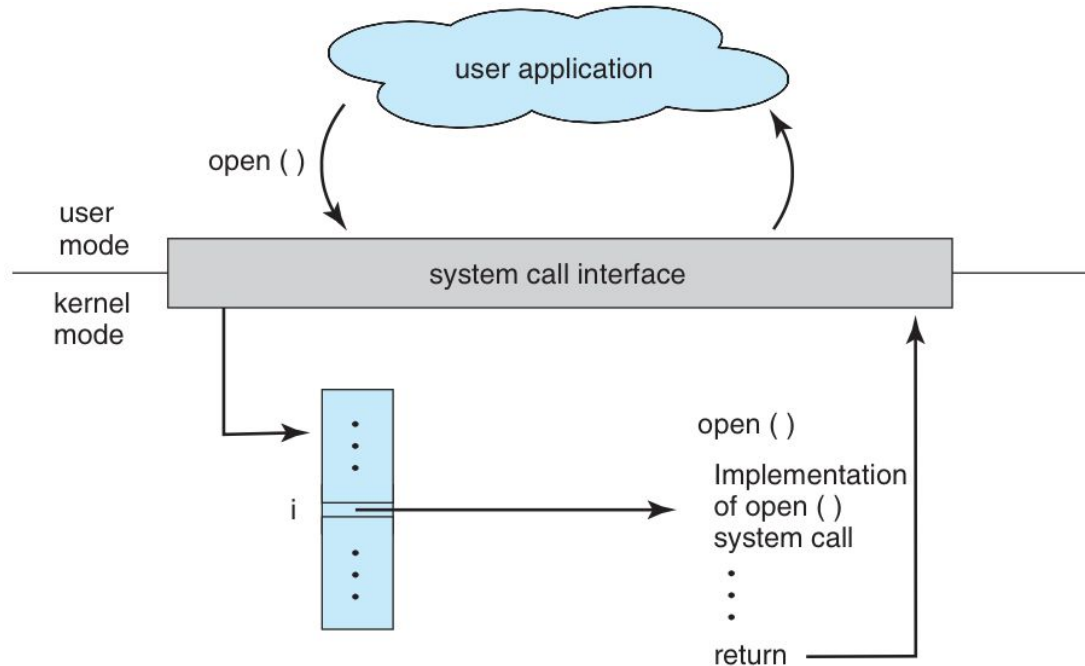
¿Qué ocurre en una llamada al sistema?

- Se invocan rutinas en una librería (C)
- Pequeña rutina en assembler
 - Mueve parámetros a un espacio predefinido
 - Almacena estado de quien hace la llamada (*caller*) para restaurarlo luego
 - Genera una *excepción* para ir al sistema operativo
 - Coloca el resultado en el espacio predefinido
 - Restaura estado almacenado

Implementación de las llamadas al sistema

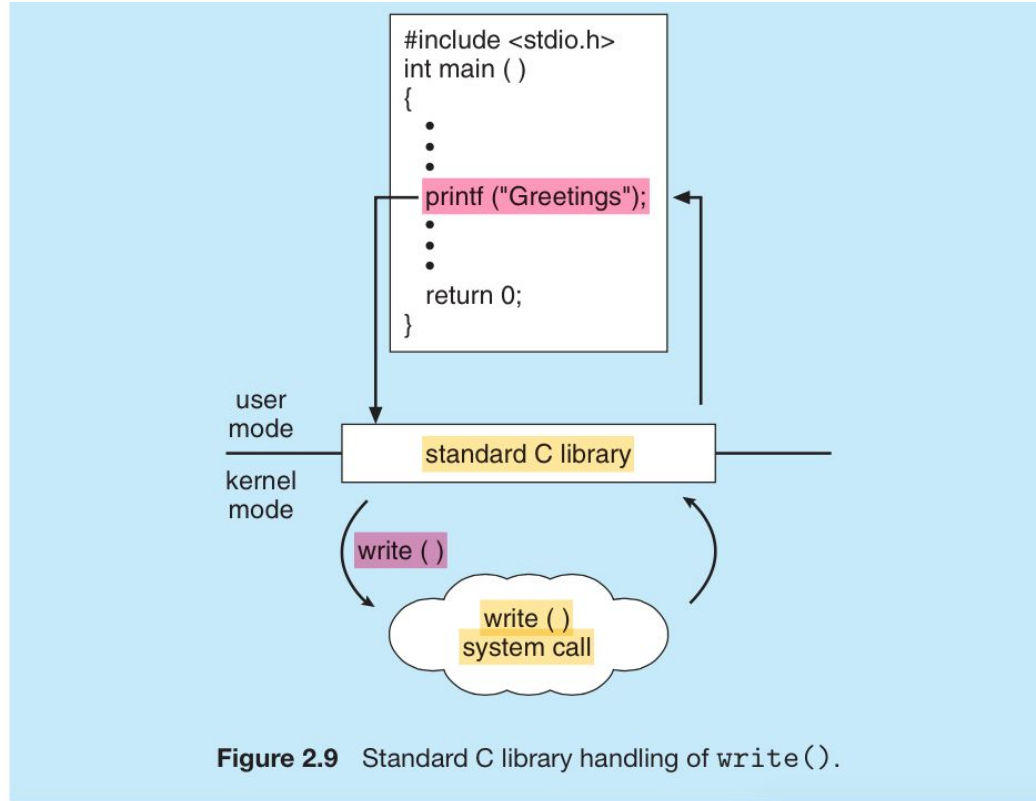
- Típicamente, un # asociado con cada llamada
 - Interfaz de llamadas al sistema mantiene una tabla indexada de acuerdo a estos #s
- Interfaz de llamadas al sistema invoca la llamada en kernel y retorna el estado de la llamada, y sus valores de retorno
- Casi todos los detalles de la interfaz del S.O. se esconden al programador a través del API
 - Administrado por librerías run-time (conjunto de funciones incluidas en librerías y proporcionadas por el compilador)

Relación API-llamadas al sistema-S.O.



The handling of a user application invoking the `open()` system call.

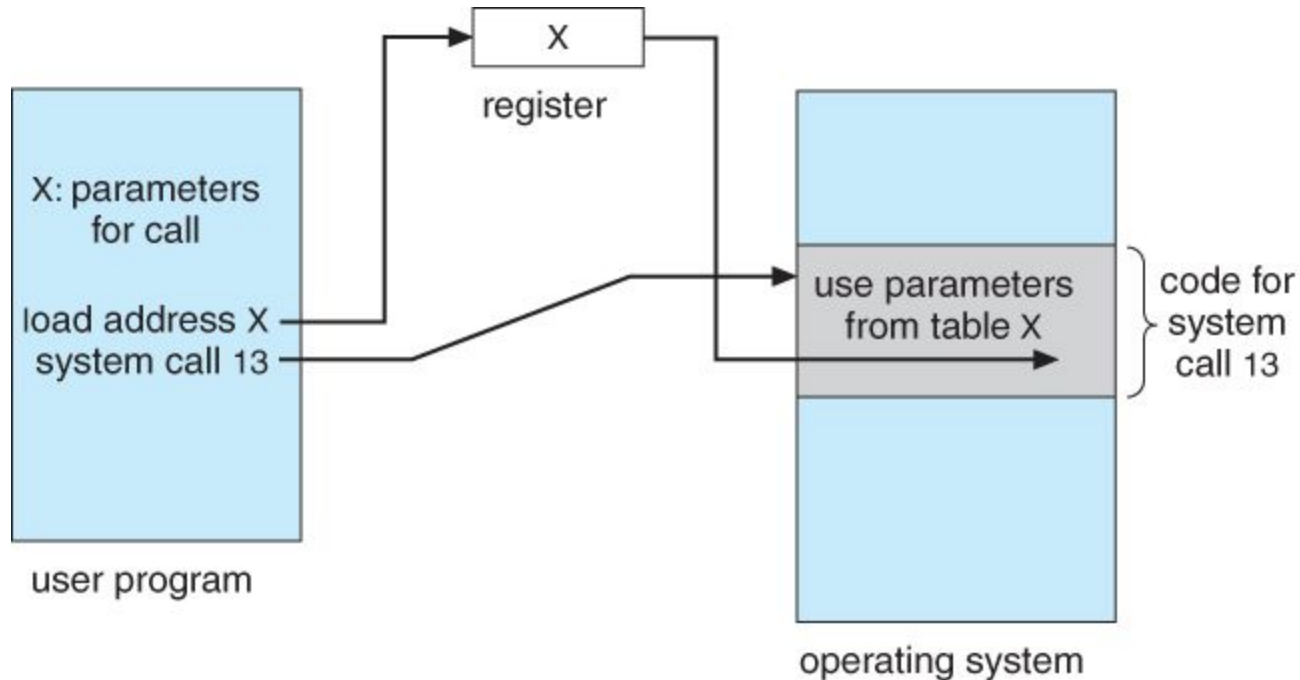
Ejemplo de C: printf (función) y write (syscall)



Paso de parámetros

- Una llamada al sistema requiere identificar: tipo (un #) y parámetros
- ¿Cómo se pueden especificar los parámetros?
 - Pasar parámetros en registros
 - Usado por Linux (6 registros: %eax (syscall #), %ebx, %ecx, %esi, %edi, %ebp)
 - ¿Limitaciones?
 - Push/pop de una pila
 - Parámetros almacenados en un bloque en memoria
 - Se pasa la dir. del bloque como parámetro en un registro
 - Enfoque utilizado por Linux (6 ó más parámetros) y Solaris
- En Linux, tres opciones:
 - int 80 → lento porque debe averiguar si es necesario pasar a modo de kernel
 - syscall → AMD
 - sysenter → Intel

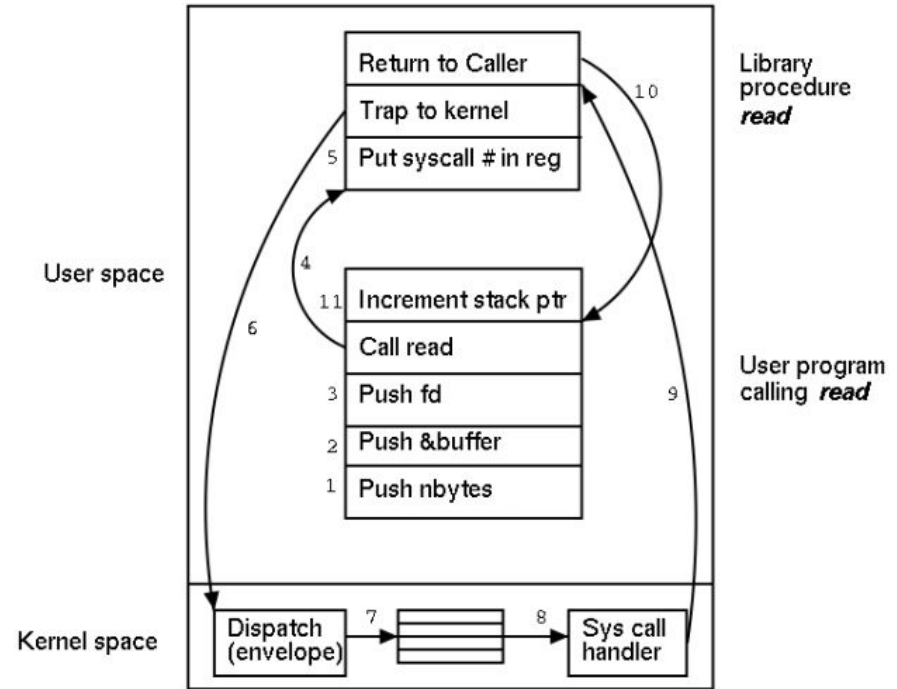
Ejemplo: Usando un bloque en memoria



Ejemplo: read()

cuenta = read (file, buffer, nbytes)

- cuenta = # bytes leídos
- Se colocan los parámetros
- Se llama a la librería
- Excepción al S.O.
- Usando el # de llamada al sistema, se ubica tabla con direcciones al “handler”
- Ejecución de la rutina de lectura
- Se vuelve al programa de usuario (siguiente instrucción)
- La pila se lee (*pop*)



Demo: Rastreo de operaciones

- pwd: Print working directory
- ltrace pwd
 - Library calls
 - setlocale, getcwd, puts
- strace pwd
 - System calls
 - execve, open, fstat, mmap, brk → ?
 - getcwd, write

Ejemplos de llamadas al sistema

Process Management		
Posix	Win32	Description
Fork	CreateProcess	Clone current process
exec(ve)		Replace current process
wait(pid)	WaitForSingleObject	Wait for a child to terminate.
exit	ExitProcess	Terminate current process & return status
File Management		
Posix	Win32	Description
open	CreateFile	Open a file & return descriptor
close	CloseHandle	Close an open file
read	ReadFile	Read from file to buffer
write	WriteFile	Write from buffer to file
lseek	SetFilePointer	Move file pointer
stat	GetFileAttributesEx	Get status info

Ejemplos de llamadas al sistema (cont.)

Directory and File System Management		
Posix	Win32	Description
mkdir	CreateDirectory	Create new directory
rmdir	RemoveDirectory	Remove <i>empty</i> directory
link	(none)	Create a directory entry
unlink	DeleteFile	Remove a directory entry
mount	(none)	Mount a file system
umount	(none)	Unmount a file system
Miscellaneous		
Posix	Win32	Description
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Change permissions on a file
kill	(none)	Send a signal to a process
time	GetLocalTime	Elapsed time since 1 jan 1970

Tipos de llamadas al sistema

- Control de procesos
- Comunicación
- Manejo de archivos
- Manejo de dispositivos (devices)
- Mantenimiento de información

Llamadas para Control de Procesos (POSIX)

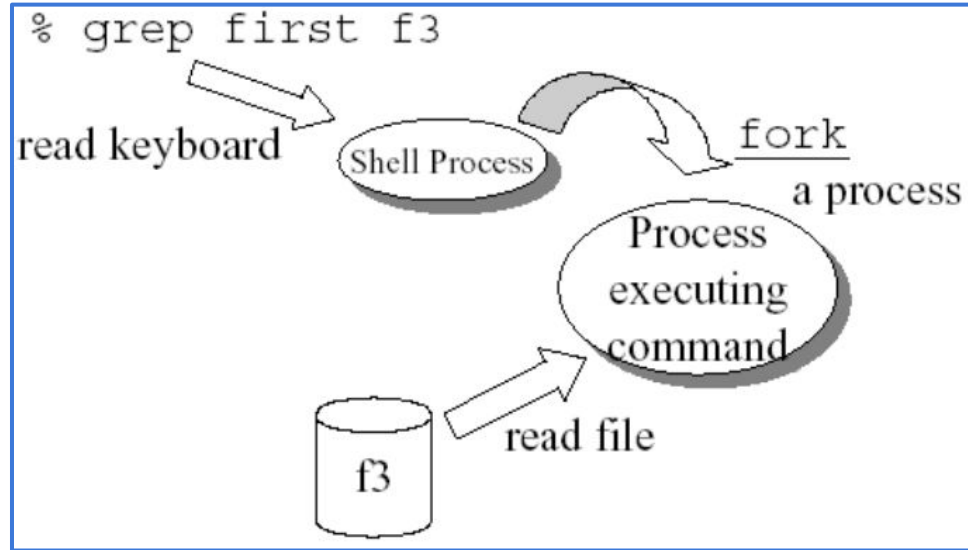
- `pid = fork()`
 - Permite crear nuevos procesos
 - Duplicado exacto de proceso original
 - Después de la copia, los espacios de direcciones no son compartidos (sólo se comparte el código del programa)
 - Devuelve un entero:
 - Cero en el proceso hijo, pid del proceso hijo (en el proceso padre)
- `wait(pid)`
 - Esperar hasta que proceso hijo termine

¿Qué hace este programa?

```
while (1) {  
    printf("$");  
    readcommand(command, args); // función definida por  
    usuario  
    if ((pid = fork ()) == 0) {  
        execv(command, args);  
    } else if (pid > 0) {  
        wait(pid);  
    } else {  
        printf("Unable to fork\n");  
    }  
}
```

¿Qué hace este programa? Es un shell

```
while (1) {  
    printf("$");           // muestra el prompt  
    readcommand(command, args); // lee comando  
    if ((pid = fork ()) == 0) {  
        // ejecuta comando  
        execv(command, args);  
    } else if (pid > 0) {  
        // espera que hijo termine  
        wait(pid);  
    } else {  
        printf("Unable to fork\n");  
    }  
}
```



execv()

- Ejecuta el programa indicado como parámetro
- Dos argumentos
 - Nombre del archivo
 - Puntero al arreglo de argumentos
- Diversas rutinas
 - Especificar parámetros de diversas maneras
 - Ej: execl, execv, execl, execve

execv()

*int execv(const char *path, char *const argv[]);*

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

main() {
    char command[100];
    char *argv[10];
    strcpy(command, "/usr/bin/ls");
    argv[0] = command;
    argv[1] = "/";
    argv[2] = NULL;
    execv(command, argv);
    cout << "finished\n";
}
```

Usando el proceso actual

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

runcommand(char *command, char *argv[])
{
    if (fork() == 0) {
        // child process
        execv(command, argv);
    }

    // parent process
    // continue
}
```

Usando el proceso hijo

Llamadas al sistema: Señales

- ¿Cómo interrumpir un comando después que éste ha sido iniciado?
- Cuando se envían señales a procesos que no han especificado cómo manejarlas, el proceso termina
 - Un manejador de señales especifica cómo manejar una señal
- `sigaction (SIGINT, SIG_IGN, null)`
 - Primer argumento: nombre de una señal
 - Definidas en `signal.h`
 - Segundo argumento:
 - `SIG_DFL` manejo default
 - `SIG_IGN` ignorar la señal
 - Puntero a una rutina
 - Tercer argumento: para guardar manejador anterior (opcional)

Demo: Manejo de señales

```
#include <signal.h>
#include <stdio.h>

void handler(int sig) {
    printf("OUCH\n");
    signal(SIGINT, &handler);
}
```

```
main() {
    signal(SIGINT, &handler); // Registra el manejador
    for (;;) pause();
}
```

```
gcc signal_example.c -o
ejemplo
```

¿Para qué sirven las señales?

- Son un tipo (primitivo) de programación basada en eventos
- Son un mecanismo (primitivo) de comunicación entre procesos
- Permiten a usuario comunicarse con procesos en el background

Demo: Procesos en el background

- ¿Cómo se envía un proceso al background en Unix?
 - comando &
 - Ejemplo: vi &
- ¿Cómo se puede detener un proceso que está en el background?
 - Podemos utilizar la llamada al sistema KILL para enviar una señal al proceso
 - La señal SIGKILL (9) no puede atraparse ni ignorarse
 - kill -9 pid
- ¿kill sirve solamente para matar procesos?
 - No, sirve para enviar señales a procesos
 - Ejemplo: enviar SIGINT al proceso del Demo anterior
 - ¿Qué número es SIGINT?
 - https://en.wikipedia.org/wiki/Unix_signal

Llamadas para manejo de archivos

- El sistema operativo tiene 3 clases de estructuras para archivos
 - La tabla de archivos
 - Una por proceso
 - Contiene punteros a los archivos abiertos por el proceso (seek pointer)
 - Entradas por defecto:
 - 0 → standard input
 - 1 → standard output
 - 2 → standard error
 - Vnode: Uno por cada archivo abierto (ref a inode, size,etc)
 - Inode: Uno por cada archivo en el disco
- Inodes se escriben en el disco
- Vnodes se ubican en el espacio de memoria del sistema operativo

Manejo de archivos: open()

```
main() {  
    int fd1, fd2;  
    fd1 = open("file1", O_WRONLY | O_CREAT | O_TRUNC, 0644);  
    fd2 = open("file1", O_WRONLY);  
}
```

- ¿Qué ocurre con esta llamada?
- ¿Dónde apunta cada vnode?
- ¿Tienen diferentes seek pointers?

Demo: write(): ¿Cuál es la salida de este programa?

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
```

¿Y si intercambiamos el orden
de los "write"?

```
main()
{
    int fd1, fd2;
    fd1 = open("file1", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    fd2 = open("file1", O_WRONLY);
    write(fd1, "Pepe\n", strlen("Pepe\n"));
    write(fd2, "Maria\n", strlen("Maria\n"));
    close(fd1); close(fd2);
}
```

Ejercicio

¿Cuál sería una salida válida del siguiente programa?

```
int main() {  
    int fd1, fd2;  
    fd1 = open("foo.txt", O_RDONLY, 0644);  
    close(fd1);  
    fd2 = open("baz.txt", O_RDONLY, 0644);  
    printf("fd2 = %d\n", fd2);  
    exit(0);  
}
```

A. fd2 = 3

B. fd2 = foo.txt

C. fd2 = baz.txt

D. fd2 = 644

Manejo de archivos: dup()

- Duplica un descriptor de archivo
 - `fd2 = dup(fd1)`
 - Resultado: los dos descriptors de archivos apuntan a la misma entrada en la tabla de archivos

Demo: dup vs. open → ¿Salida?

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>

main()
{
    int fd1, fd2;
    fd1 = open("file1", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    fd2 = dup(fd1);
    write(fd1, "Pepe\n", strlen("Pepe\n"));
    write(fd2, "Maria\n", strlen("Maria\n"));
    close(fd1); close(fd2);
}
```


exec() → Nuevamente

- Cuando exec se ejecuta se carga el nuevo programa a ser ejecutado y el anterior termina
- Todas las instrucciones después de un exec no se ejecutarán en el programa original
- Hay solamente un elemento que permanece sin cambio:
 - La tabla de descriptores de archivo
- Se puede utilizar exec desde línea de comando

Demo: exec() desde la línea de comando

- ¿Qué pasará si ejecutamos lo siguiente?

```
$ exec ps
```

- ¿Y lo siguiente?

```
$ exec top
```

exec() → Nuevamente (cont.)

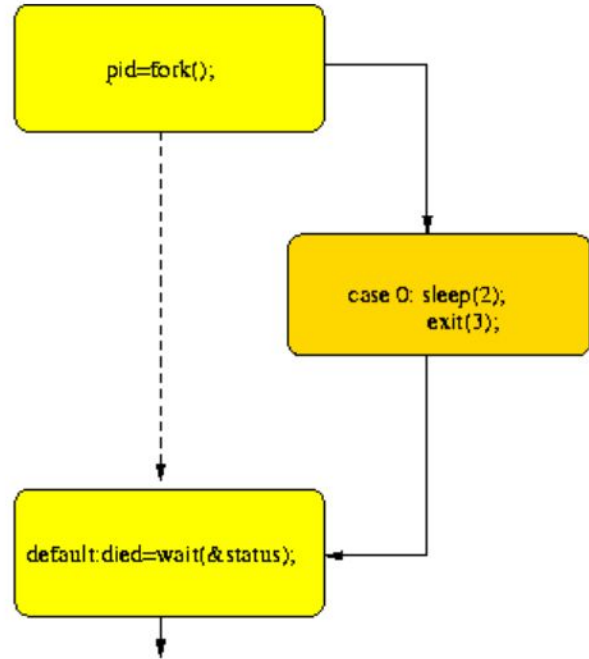
- OJO: Si un proceso padre no espera (wait) a que termine un proceso hijo y termina primero, el proceso hijo es adoptado por otro proceso (generalmente el que tiene PID 1)
- Si un hijo termina y nadie lo ha esperado (wait), se queda como un *zombie*, ocupando memoria
 - Ciertos S.O. liberan esta memoria después de un tiempo
- Otra mejor alternativa: crear procesos no ligados a un padre

Demo: zombie process

```
main ()
{
    if (fork() > 0)
    {
        sleep (60);
    }
}
```

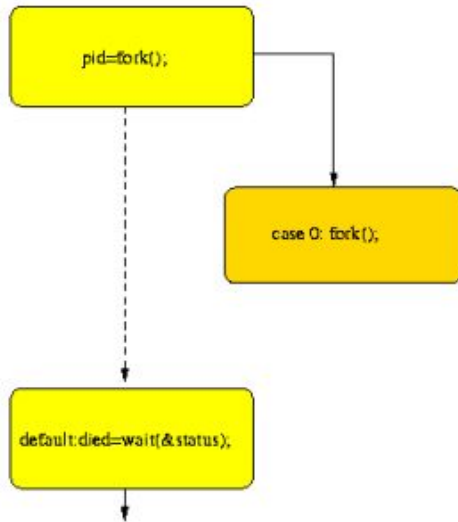
¿Cómo evitar crear procesos zombie?

- Dos alternativas:
 - Padre espera que hijo termine →
 - Crear hijos no ligados al padre

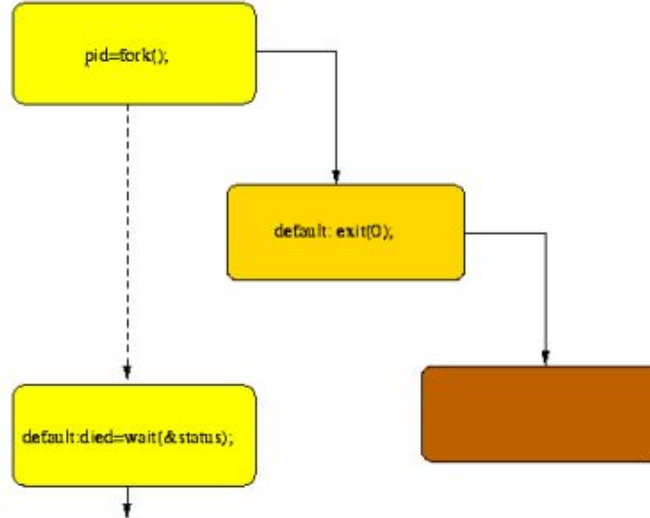


Creando un proceso no ligado al padre

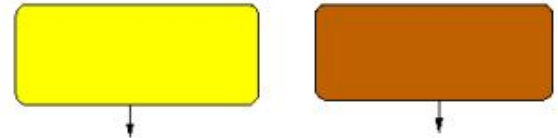
The initial process forks...



...the new child forks...



...and dies, leaving the original parent and its detached grandchild



Descriptores de archivos después de un `fork()`

- Todos los descriptores de archivo son duplicados después de un `fork()`
- Incluyendo `stdin`, `stdout` y `stderr`

Demo: Descriptores de Archivos

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
main()
{
    char s[1000];
    int i, fd;
    fd = open("file3", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    i = fork();
    sprintf(s, "fork() = %d.\n", i);
    write(fd, s, strlen(s));
}
```


Demo (cont.)

```
$ fs3
$ cat file3
fork() = 0.
fork() = 22107.
$ fs3
$ cat file3
fork() = 0.
fork() = 22110.
$ fs3
$ cat file3
fork() = 22113.
fork() = 0.
$
```

Comunicación entre Procesos

Comunicación entre procesos

- Los procesos pueden querer colaborar entre sí
- S.O. debe proveer los mecanismos
 - Señales ← ver diapositivas anteriores
 - Memoria compartida
 - Paso de mensajes
 - Sockets
 - Pipes

Paso de mensajes

- Antes de que la comunicación exista debe abrirse una conexión
 - Directa o a través de un “mailbox”
- Se debe poder identificar a los procesos involucrados
 - IP o pid
 - Llamadas al sistema útiles incluyen (gethostId, getprocessId)
- La comunicación la inicia un cliente
- La comunicación es aceptada por el servidor
- Si algún servidor es usado frecuentemente es posible crear un programa del sistema
 - Daemon
 - ¿Ejemplos?

Paso de mensajes en POSIX

- Message queues
 - `<sys/msg.h>`
- Crear mailbox
 - `msgget()`
- Enviar mensaje a mailbox
 - `msgsnd()`
- Recibir mensaje de mailbox
 - `msgrcv()`
- Eliminar mailbox
 - `msgctl()`
- Más información
 - <https://www.cs.cf.ac.uk/Dave/C/node25.html>

Memoria compartida

- Dos procesos acuerdan que uno de ellos va a tener acceso a las regiones de memoria del otro
- Los procesos definen el formato de los datos y el control de acceso a la memoria
- ¿Ventajas?
- ¿Desventajas?

Ventaja:

Más rápido que el paso de mensajes. ¿Por qué?

Desventaja:

Requiere mecanismos de sincronización
(susceptible a condiciones de carrera)

Memoria compartida en POSIX

- Memoria compartida
 - `<sys/shm.h>`
- Crear segmento de memoria compartida
 - `shm_open()`
- Mapear el segmento de memoria compartida en el espacio de direcciones del usuario
 - `mmap()`
- Escribir/leer de la memoria compartida
 - Igual que de cualquier posición de memoria (punteros)
- Ejemplo del libro:
 - <http://www.cse.psu.edu/~deh25/cmpsc473/notes/OSC/Processes/shm-posix-producer-orig.c>

Pipes

- Llamada al sistema pipe()
- pipe() crea un **pipe** y retorna dos descriptores de archivos
 - El primero abierto para lectura
 - El segundo abierto para escritura
- Tanto proceso padre como hijo inicialmente tienen accesos a ambos extremos del pipe
- Demo:
 - `ps -ef | grep ssh | more`
 - Sintaxis BSD: `ps aux | grep ssh | more`



Otros servicios



Manejo de dispositivos

- Los recursos del SO pueden ser considerados como dispositivos
- El SO debe de garantizar acceso a un recurso antes de ser usado (RAM, disk, etc)
- Los pasos para tener acceso a dispositivos son muy similares a el acceso a archivos y la comunicación entre procesos
 - Solicitar acceso, esperar que este disponible, leer/escribir, cerrar (liberar)
- En algunos SO los dispositivos se implementan como archivos y se evita crear llamadas al sistema adicionales
 - ej.: stdin, stdout, stderr

Manejo de información

- Existe llamadas al sistema con el solo propósito de transferir información del SO al los programas
 - time
 - date
- Otros ejemplos:
 - Estado del sistema
 - free
 - df
 - Procesos en el sistema
 - ps
 - top

Programas del Sistema

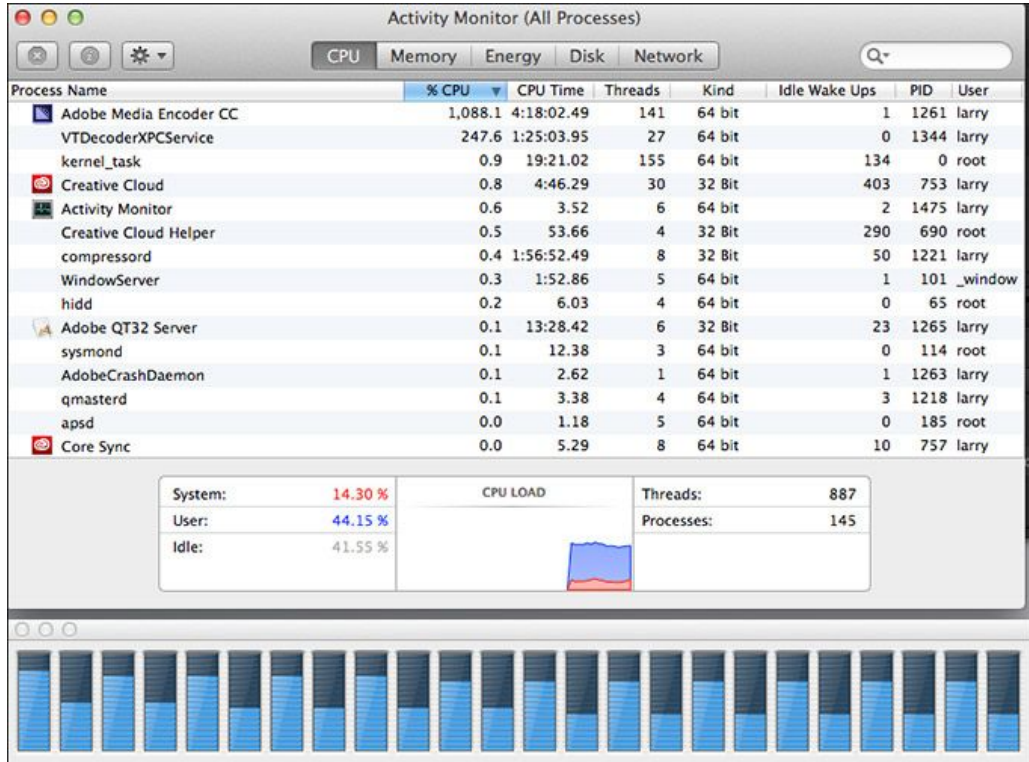
Programas del sistema

- Proporcionan un ambiente conveniente para el desarrollo y ejecución de programas
 - Pueden ser simples IU a las llamadas del sistema o considerablemente más complejos
- Tipos:
 - Manipulación de archivos: Explorador de archivos, Nautilus
 - Información de estado: top, ps, GUIs
 - Modificación de archivos: vim, pico, nano, emacs
 - Soporte de lenguajes de programación: gcc, gdb, ltrace, strace
 - Carga y ejecución de programas
 - Comunicaciones: telnet, ssh, ftp
 - Programas de aplicación: calc, navegadores

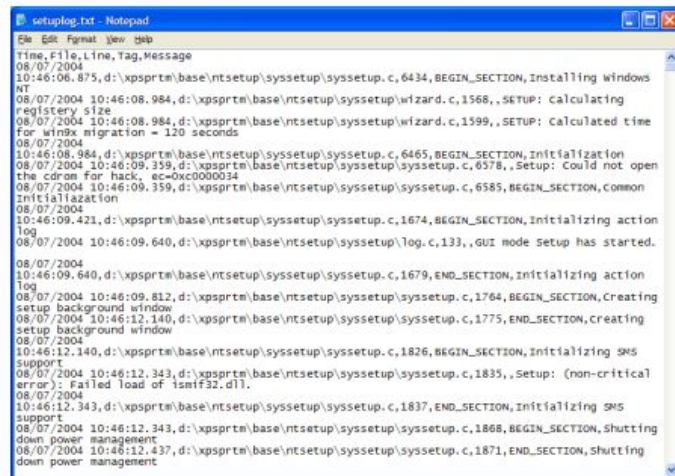
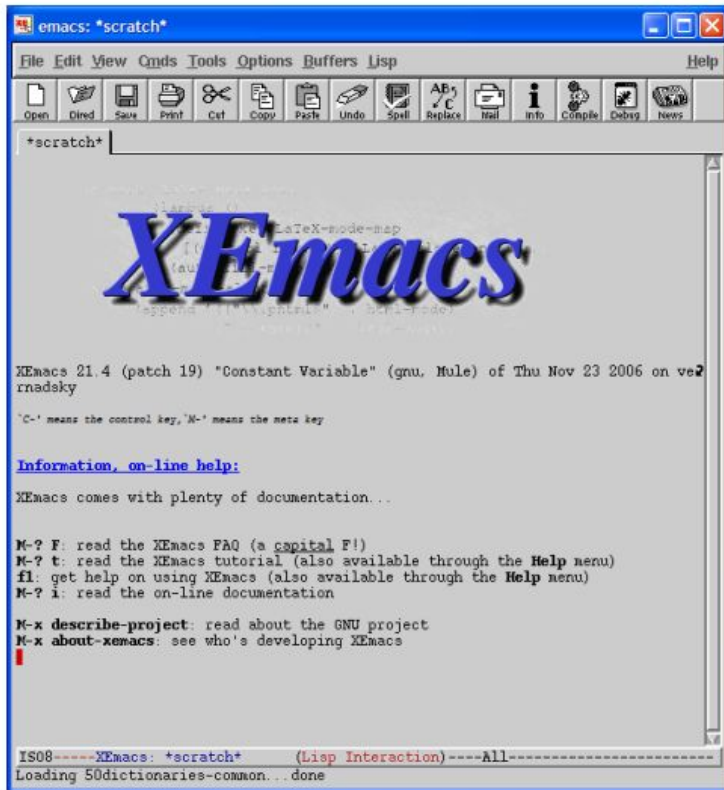
Manipulación de archivos



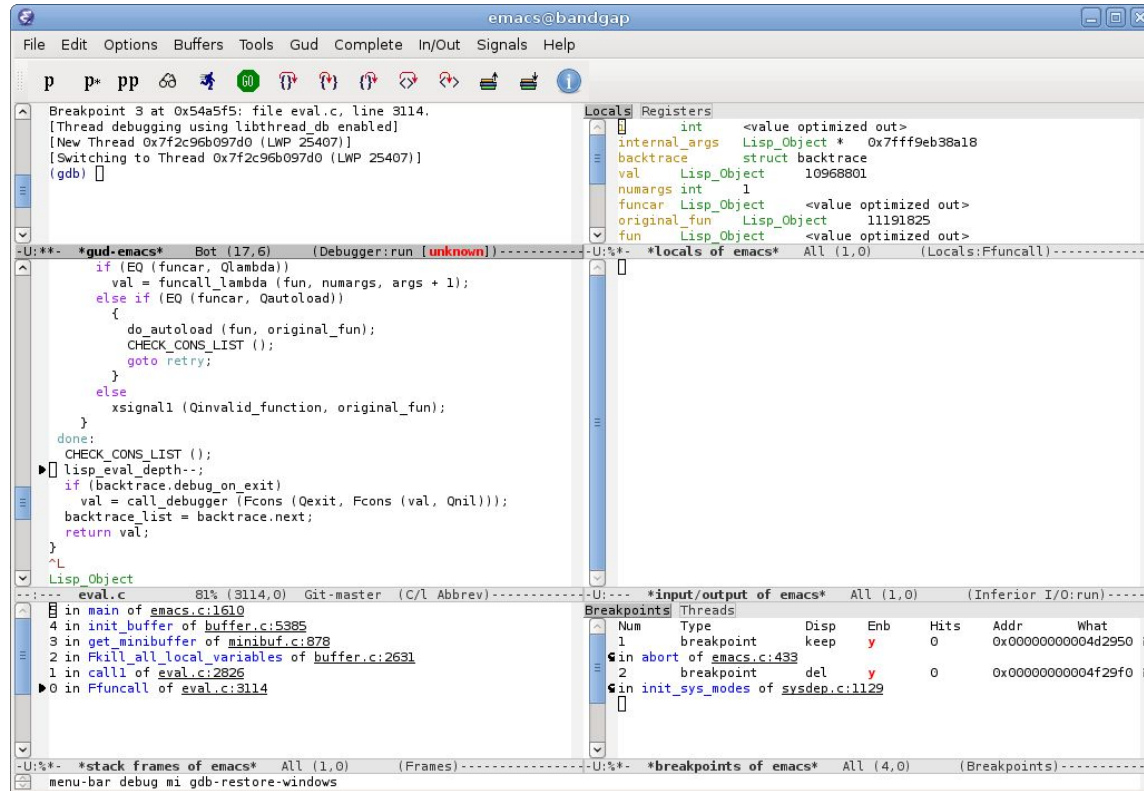
Información de estado



Modificación de archivos



Soporte de lenguajes de programación



The screenshot shows the Emacs editor window titled 'emacs@bandgap'. The menu bar includes File, Edit, Options, Buffers, Tools, Gud, Complete, In/Out, Signals, and Help. The toolbar contains various icons for file operations, editing, and debugging. The main window displays the source code of 'eval.c' with a breakpoint set at line 3114. The GDB console shows the execution state, including the current thread (Thread 0x7f2c96b097d0) and the current frame (main of eval.c). The 'Locals' panel shows the current frame's variables, including 'internal_args', 'backtrace', 'val', 'numargs', 'fun', and 'original_fun'. The 'Breakpoints' panel shows the current breakpoint at line 3114 of eval.c.

```
Breakpoint 3 at 0x54a5f5: file eval.c, line 3114.
[Thread debugging using libthread_db enabled]
[New Thread 0x7f2c96b097d0 (LWP 25407)]
[Switching to Thread 0x7f2c96b097d0 (LWP 25407)]
(gdb)

-U:***- *gud-emacs* Bot (17,6) (Debugger:run [unknown])-----
if (EQ (funcar, Qlambda))
  val = funcall_lambda (fun, numargs, args + 1);
else if (EQ (funcar, Qautoload))
{
  do_autoload (fun, original_fun);
  CHECK_CONS_LIST ();
  goto retry;
}
else
  xsignal1 (Qinvalid_function, original_fun);
}
done:
CHECK_CONS_LIST ();
lisp_eval_depth--;
if (backtrace.debug_on_exit)
  val = call_debugger (Fcons (Qexit, Fcons (val, Qnil)));
backtrace_list = backtrace.next;
return val;
}
^L
Lisp_Object
----- eval.c 81% (3114,0) Git-master (C/l Abbrev)-----
in main of emacs.c:1610
4 in init_buffer of buffer.c:5385
3 in get_minibuffer of minibuf.c:878
2 in kill_all_local_variables of buffer.c:2631
1 in call1 of eval.c:2826
0 in Ffuncall of eval.c:3114

Locals Registers
internal_args Lisp_Object * 0x7fff9eb38a18
backtrace struct backtrace
val Lisp_Object 10968801
numargs int 1
fun Lisp_Object <value optimized out>
original_fun Lisp_Object 11191825
fun Lisp_Object <value optimized out>

-U:***- *locals of emacs* All (1,0) (Locals:Ffuncall)-----

-U:***- *input/output of emacs* All (1,0) (Inferior I/O:run)-----
Breakpoints Threads
Num Type Disposition Enb Hits Addr What
1 breakpoint keep y 0 0x000000000004d2950
in abort of emacs.c:433
2 breakpoint del y 0 0x000000000004f29f0
in init_sys_modes of sysdep.c:1129

-U:***- *stack frames of emacs* All (1,0) (Frames)-----
-U:***- *breakpoints of emacs* All (4,0) (Breakpoints)-----
menu-bar debug mi gdb-restore-windows
```

Diseño e Implementación del S.O.

Diseño e Implementación del S.O.

- Diseño e implementación de S.O. no es una ciencia exacta, pero ciertos enfoques son útiles
- Estructura interna de diferentes S.O. varía considerablemente
- Lo primero es especificar las metas
 - Depende de HW y tipo del sistema
 - **Metas de usuario** – S.O. debe ser conveniente, de fácil aprendizaje, confiable, seguro y rápido
 - **Metas del sistema** – S.O. debe ser de fácil diseño, implementación y mantenimiento, a la vez que es flexible, confiable, sin errores y eficiente

Principios de diseño: Políticas vs. Mecanismos

- Es importante separar
 - **Políticas:** ¿Qué debe hacerse?
 - **Mecanismos:** ¿Cómo lo hacemos?
- Los mecanismos determinan cómo hacer algo
- Las políticas deciden lo que se debe hacer
- Separarlos es un principio muy importante
 - Permite flexibilidad máxima
 - Permite que las políticas cambien en el futuro
- Esquema de microkernel lleva este principio a su máxima expresión
 - Microkernel no incluye casi ninguna política

Ejemplo

- UNIX original era de tiempo compartido
- Solaris carga las políticas de planificación de trabajos de unas tablas
 - Dependiendo de la tabla cargada, se aplican diferentes políticas
 - Puede ser:
 - Tiempo compartido
 - Tiempo real
 - Por lotes
 - Etc.

Principios de diseño: Seguridad

- “A chain is only as strong as its weakest link”

Microkernel

- Beneficios
 - Más fácil extender
 - Más fácil portar el S.O. a nuevas arquitecturas
 - Más confiable (menos código corre en modo de kernel)
 - Más seguro
- Desventajas
 - Rendimiento reducido debido a las comunicaciones entre espacio de usuario y de kernel

Diseño modular vs. diseño por capas

- La mayoría de S.O. modernos implementan módulos de kernel
 - Enfoque orientado a objetos
 - Cada componente central está separado
 - Cada componente habla con los otros a través de interfaces conocidas
 - Cada módulo puede ser cargado de manera independiente, de ser necesario
- Similar a las capas, pero más flexible

Diseño de máquinas virtuales

- Toman el enfoque por capas y lo llevan a su conclusión lógica
- Tratan el HW y kernel del S.O. como si todo fuera HW
- La MV proporciona una interfaz idéntica a la del hardware real
- S.O. crea la ilusión de múltiples procesos, c/u ejecutándose en su propio procesador, con su propia memoria (virtual)