

Procesos y API del SO

Rodolfo Baader¹

¹Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Sistemas Operativos, segundo cuatrimestre de 2024


(2) La clase anterior...

- Vimos
 - Qué es un SO.
 - Un administrador de recursos.
 - Una interfaz de programación.
 - Un poco de su evolución histórica.
 - Su misión fundamental.
 - Hablamos de multiprogramación.
 - Qué cosas son parte del SO y cuáles no.

(3) La de hoy

- Vamos a ver qué cosas hay detrás del concepto de proceso.
- Y qué abstracciones nos presenta el SO para lidiar con ellas.
- Es decir, una parte de la API del SO.

(4) Los procesos

- Un programa es una secuencia de pasos escrita en algún lenguaje.
- Ese programa eventualmente se compila en código objeto, lo que también es un programa escrito en lenguaje de máquina.
- Cuando ese programa se pone a ejecutar, lo que tenemos es un *proceso*. 
- A cada proceso se le asigna un identificador numérico único, el *pid* o *process id*.

(5) Procesos

- Visto desde la memoria, un proceso está compuesto por:
 - El área de *texto*, que es el código de máquina del programa.
 - El área de datos, que es donde se almacena el heap.
 - El stack del proceso.
 - ¿Dónde se almacenan las variables locales?
- ¿Qué puede hacer un proceso?
 - Terminar.
 - Lanzar un proceso hijo (`system()`, `fork()`, `exec()`).
 - Ejecutar en la CPU.
 - Hacer un *system call*.
 - Realizar entrada/salida a los dispositivos (E/S).
- Analicemos cada una de las actividades del proceso.

(6) Actividades de un proceso: terminación

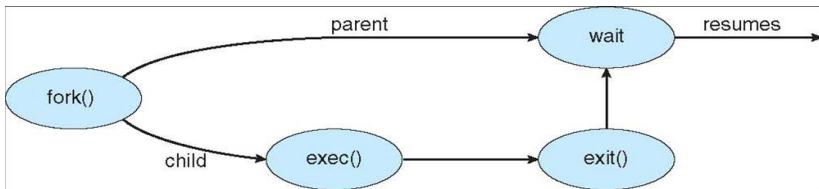
- El proceso indica al sistema operativo que ya puede liberar todos sus recursos (`exit()`).
- Además, indica su status de terminación (usualmente, un código numérico).
- Este código de status le es reportado al padre.
- ¿Qué padre?

(7) Árbol de procesos

- En realidad, todos los procesos están organizados jerárquicamente, como un árbol.
- Cuando el SO comienza, lanza un proceso que se suele llamar *init* o *systemd*.
- Por eso es importante la capacidad de lanzar un proceso hijo:
 - `fork()` es una llamada al sistema que crea un proceso exactamente igual al actual.
 - El resultado es el pid del proceso hijo, que es una copia exacta del padre.
 - El padre puede decidir suspenderse hasta que termine el hijo, llamando a `wait()`.
 - Cuando el hijo termina, el padre obtiene el código de status del hijo.
 - El proceso hijo puede hacer lo mismo que el padre, o algo distinto. En ese caso puede reemplazar su código binario por otro (`exec()`).

(8) Árbol de procesos (cont.)

- Cuando lanzamos un programa desde el shell, ¿qué sucede?
- El shell hace un `fork()`.
- El hijo hace un `exec()`.




Árbol de procesos: pstree

```
$pstree -u user
-+= 00001 root /sbin/launchd
...
\-= 00708 user /Applications/.../Terminal
  \-= 00711 root login -pf \user
    \-= 00712 user -bash
      \-= 00789 user pstree -u user
        \--- 00790 root ps -axww user,pid,ppid,pgid,...
```

(9) Actividades de un proceso: ejecutar en la CPU

- Una vez que el proceso está ejecutándose, se dedica a:
 - hacer operaciones entre registros y direcciones de memoria,
 - E/S,
 - llamadas al sistema (Syscalls).
- Imaginemos el programa más elemental, que sólo hace lo primero.


(10) Ejecutando en la CPU

- ¿Por cuánto tiempo lo dejamos ejecutar? (recordemos: sólo un proceso a la vez puede estar en la CPU).
 - Hasta que termina: Es lo mejor para el proceso, pero no para el sistema en su conjunto. Además, podría no terminar.
 - Un “ratito”. Ese “ratito” se llama *quantum*. 
- En general los SO modernos hacen *preemption*: cuando se acaba el quantum, le toca el turno al siguiente proceso.
- Surgen dos preguntas:
 - Quién y cómo decide a quién le toca.
 - Qué significa hacer que se ejecute otro proceso.

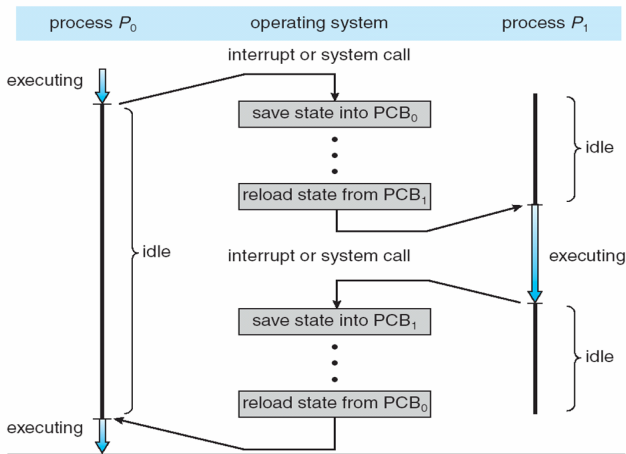
(11) Scheduler

- Aparece un componente esencial del SO, el *scheduler* o *planificador*. ⚠
- Es una parte fundamental del kernel.
- Su función es decidir a qué proceso le corresponde ejecutar en cada momento.
- Hay varias diferentes formas de decidir esto, que veremos la clase que viene.
- Pocas cosas tienen mayor impacto en el rendimiento de un SO que su política de scheduling. ⚠


(12) Ejecutando en la CPU (cont.)

- Para cambiar el programa que se ejecuta en la CPU, debemos:
 - Guardar los registros.
 - Guardar el IP.
 - Si se trata de un programa nuevo, cargarlo en memoria.
 - Cargar los registros del nuevo.
 - Poner el valor del IP del nuevo.
 - Otras cosas que veremos más adelante.
- A esto se lo llama *cambio de contexto* o *context switch*. 
- El IP y demás registros se guardan en una estructura de datos llamada *PCB* (Process Control Block).
- Notemos: el tiempo utilizado en cambios de contexto es tiempo muerto, no se está haciendo nada productivo. Dos consecuencias de esto:
 - Impacto en la arquitectura del HW: procesadores RISC.
 - Fundamental determinar un quantum apropiado para minimizar los cambios de contexto.
- Implementación: colgarse de la interrupción del clock.

(13) Cambio de contexto



(14) Actividades de un proceso: llamadas al sistema

- Un proceso también puede hacer llamadas al sistema.
- Algunas ya las vimos: `fork()`, `exec()`, etc.
- También hay llamadas al sistema en actividades mucho más comunes: imprimir en pantalla a la larga termina llamando a `write()`.
- En todas ellas se debe llamar al kernel. A diferencia de una llamada a subrutina común y corriente, las llamadas al sistema requieren cambiar el nivel de privilegio, un cambio de contexto, **a veces una interrupción**, etc. 
- Eso toma tiempo.

(15) Syscalls

- Las *syscalls* proveen una **interfaz** a los servicios brindados por el sistema operativo: la API (Application Programming Interface) del SO.
- La mayoría de los programas hacen un uso intensivo de ellas.
- Implementación: en general, se usa una interrupción para pasar a modo *kernel*, y los parámetros se pasan usando registros o una tabla en memoria. En Linux: interrupción **0x80** (en 32 bits); el **número de syscall** va por EAX (o RAX).
- Normalmente se las utiliza a través de *wrapper functions* en C. ¿Por qué no directamente? Veamos un ejemplo.

Un primer ejemplo

tinyhello.asm

```
section .data
hello: db 'Hola S0!', 10
hello_len: equ $-hello

section .text
global _start
_start:
    mov eax, 4 ; syscall write
    mov ebx, 1 ; stdout
    mov ecx, hello ; mensaje
    mov edx, hello_len
    int 0x80

    mov eax, 1 ; syscall exit
    mov ebx, 0 ;
    int 0x80
```

Usando *wrapper functions* en C

- Claramente, el código anterior no es *portable*.
- Además, realizar una *syscall* de esta forma requiere programar en lenguaje ensamblador.
- Las *wrapper functions* permiten interactuar con el sistema con mayor **portabilidad** y **sencillez**.

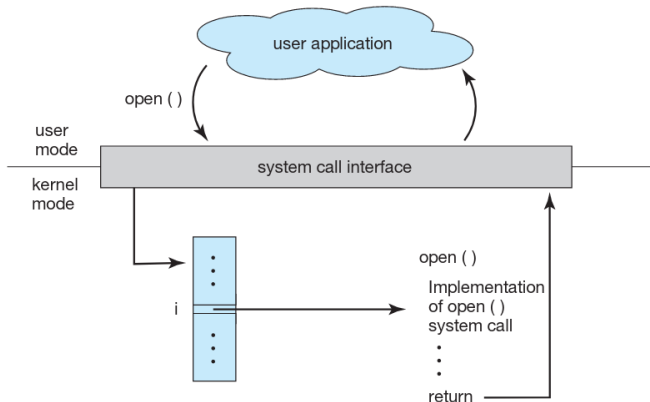
El ejemplo anterior, pero ahora en C:

```
hello.c
```

```
#include <unistd.h>

int main(int argc, char* argv[]) {
    write(1, "Hola S0!\n", 9);
    return 0;
}
```

Ejemplo de invocación a *syscall*

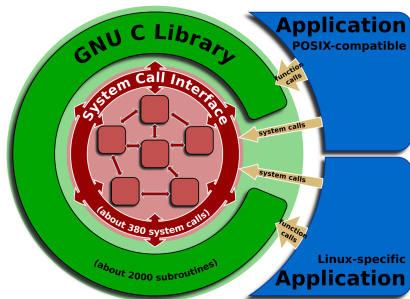


Invocación de la *syscall* `open()` desde una aplicación de usuario.

Imagen extraída de *Operating System Concepts* (Abraham Silberschatz et al.)

- La biblioteca estándar de C incluye funciones que no son *syscalls*, pero las utilizan para funcionar. Por ejemplo, `printf()` invoca a la *syscall* `write()`.
- Están definidas en el archivo `unistd.h` de la biblioteca estándar de C. Puede verse una lista de todas ellas usando `man syscalls`.

Syscalls en Linux



Basado en una ilustración de Shmuel Csaba Otto Traian (Wikimedia Commons).

El espacio correspondiente a la librería de C como al de aplicaciones se encuentran dentro del modo usuario. Por otro lado, aquello en color rojo corresponde al nivel kernel. Las aplicaciones pueden tener llamadas a funciones de la librería C o directamente a syscalls del sistema.

(21) Overhead system call

En MacOS 10.3.3 con Intel Core i7 (2012) 2.9 Ghz.

Ciclos de reloj de 10.000.000 iteraciones:

system call:	1.319.374.911
llamada a función:	993.981.671
asignación en arreglos:	584.898.656

Llamada a función es 1.699408 veces más cara que asignación.


System call es 1.327363 veces más cara que llamada a función.

(22) Ejemplos de llamadas al sistema

	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() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

- POSIX: Portable Operating System Interface; X: UNIX.
- IEEE 1003.1/2008 (<http://goo.gl/k7WGnP>)
- Core Services:
 - Creación y control de procesos
 - Pipes
 - Señales
 - Operaciones de archivos y directorios
 - Excepciones
 - Errores del bus.
 - Biblioteca C
 - Instrucciones de E/S y de control de dispositivo (ioctl).

(24) Actividades de un proceso: E/S

- La E/S es leentaaa, muuuy leentaaa.
- Quedarse bloqueado esperando es un desperdicio de tiempo...
- ...porque involucra hacer *busy waiting*, es decir, gastar ciclos de CPU.
- Hay una serie de alternativas más interesantes: 
 - Polling.
 - Interrupciones.
 - Otras que no vamos a ver por ahora.

(25) Comparación de los modos de E/S

- Busy waiting: el proceso no libera la CPU. Un único proceso en ejecución a la vez.
- Polling: El proceso libera la CPU, pero todavía recibe un quantum que desperdicia hasta que la E/S esté terminada.
- Interrupciones: Esto permite la multiprogramación.
 - El SO no le otorga más quanta al proceso hasta que su E/S esté lista.
 - El HW comunica que la E/S terminó mediante una interrupción.
 - La interrupción es atendida por el SO, que en ese momento “despierta al proceso”.

(26) Multi...

- Multiprocesador: un equipo con más de un procesador.
- Multiprogramación: la capacidad de un SO de tener varios programas (procesos en realidad) en ejecución.
- Multiprocesamiento: Aunque a veces se lo usa como sinónimo de lo anterior, se refiere al tipo de procesamiento que sucede en los multiprocesadores.
- Multitarea: es una forma especial de multiprogramación, donde la conmutación entre procesos se hace de manera tan rápido que da la sensación de que varios programas están corriendo en simultáneo.
- Multithreaded: son programas (procesos) en los cuales hay varios “mini procesos” corriendo en paralelo (de manera real o ficticia como en la multiprogramación).
- Multiusuario:

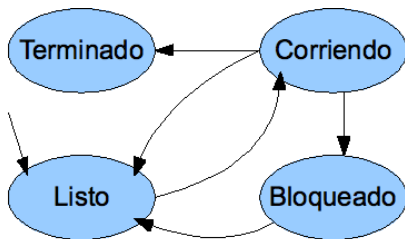


(27) Multiprogramación desde el código

- Hay dos formas de hacer esto desde el código:
 - Bloqueante: hago el system call, para cuando recibo el control la E/S ya terminó. Mientras, me bloqueo.
 - No bloqueante: hago el system call, que retorna en seguida. Puedo seguir haciendo otras cosas. Debo enterarme de alguna manera si mi E/S terminó.
- system call: `select()`
- Forma de uso: `select(..., *lectura, *escritura, *excepcion, timeout)`
- lectura, escritura y excepcion son conjuntos de “E/S pendientes” (los detalles en la práctica).
- Vuelve al pasar el timeout o cuando alguna de mis E/S está lista (o dio error).


(28) Estado de un proceso

Esto da origen al concepto de *estado de un proceso*. ⚠



- Corriendo: está usando la CPU.
- Bloqueado: no puede correr hasta que algo externo suceda (típicamente E/S lista).
- Listo: el proceso no está bloqueado, pero no tiene CPU disponible como para correr.

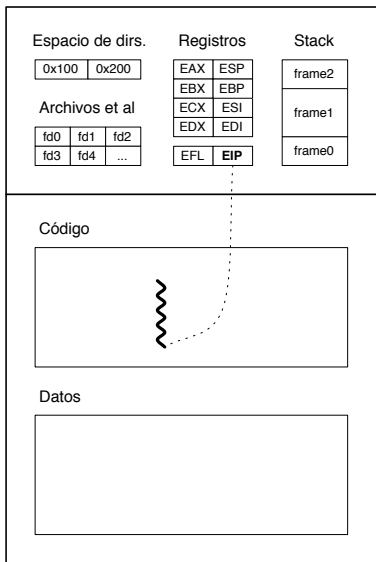
(29) Estado de un proceso

- Notar: carga del sistema = cantidad de procesos listos.
- Es responsabilidad del scheduler elegir entre los procesos listos cuál es el próximo a correr.
- Cuál elegir está determinado por la política de scheduling, que veremos más adelante.
- Sin embargo, queda claro que necesita tener una lista de procesos.
- En realidad, es una lista de PCBs y se llama *tabla de procesos*. 
- En cada PCB, además, se guarda la prioridad del proceso, su estado, y aquellos recursos por los que está esperando.
- Los PCBs suelen también formar una lista enlazada que comienza en cada recurso por el que están esperando.

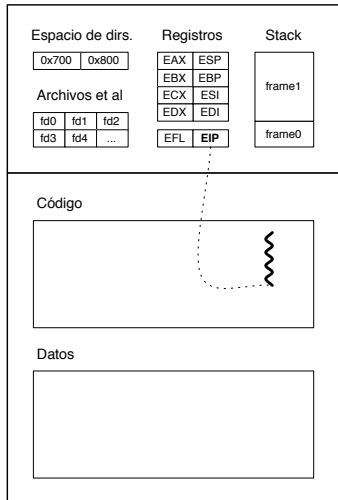
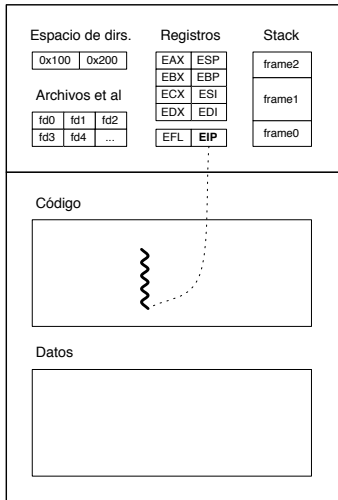
(30) Señales

- Las **señales** son un mecanismo que incorporan los sistemas operativos POSIX, y que permite notificar a un proceso la ocurrencia de un evento.
- Cuando un proceso recibe una señal, su ejecución se interrumpe y se ejecuta un *handler*.
- Cada tipo de señal tiene asociado un *handler* por defecto, que puede ser modificado mediante la *syscall* `signal()`.
- Toda señal tiene asociado un número que identifica su tipo. Estos números están definidos como constantes en el *header* `<signal.h>`. Por ejemplo: `SIGINT`, `SIGKILL`, `SIGSEGV`.
- Las señales `SIGKILL` y `SIGSTOP` no pueden ser bloqueadas, ni se pueden reemplazar sus *handlers*.
- Un usuario puede enviar desde la terminal una señal a un proceso con el *comando* `kill`. Un proceso puede enviar una señal a otro mediante la *syscall* `kill()`.

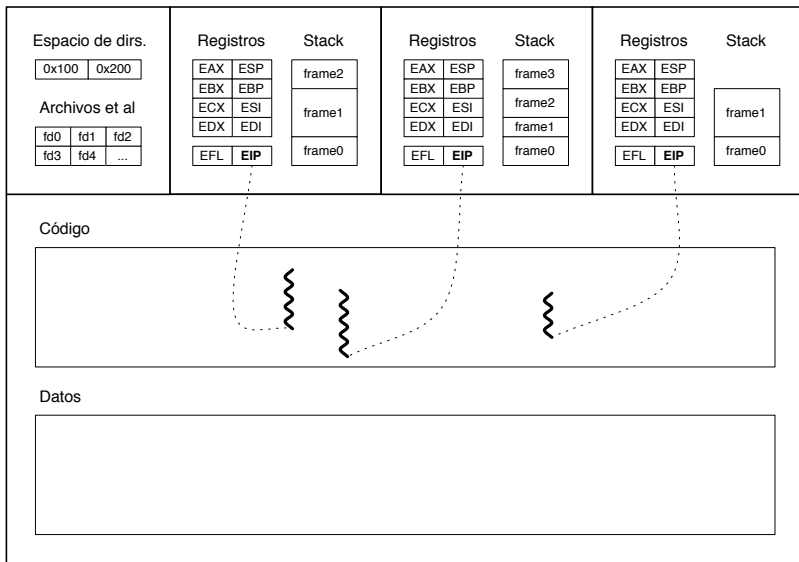
¿Qué es un proceso?



Concurrencia con varios procesos



Concurrencia con threads



(34) Dónde estamos

- Vimos
 - El concepto de proceso en detalle.
 - Sus diferentes actividades.
 - Qué es una system call.
 - Una introducción al scheduler.
 - Hablamos de multiprogramación, y vimos su relación con E/S.
 - Hablamos de threads.
- En la próxima teórica:
 - Vemos comunicación entre procesos (IPC)