Procesos y API

Rodolfo Baader

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

Sistemas Operativos, segundo cuatrimestre de 2016

(2) Dónde estamos

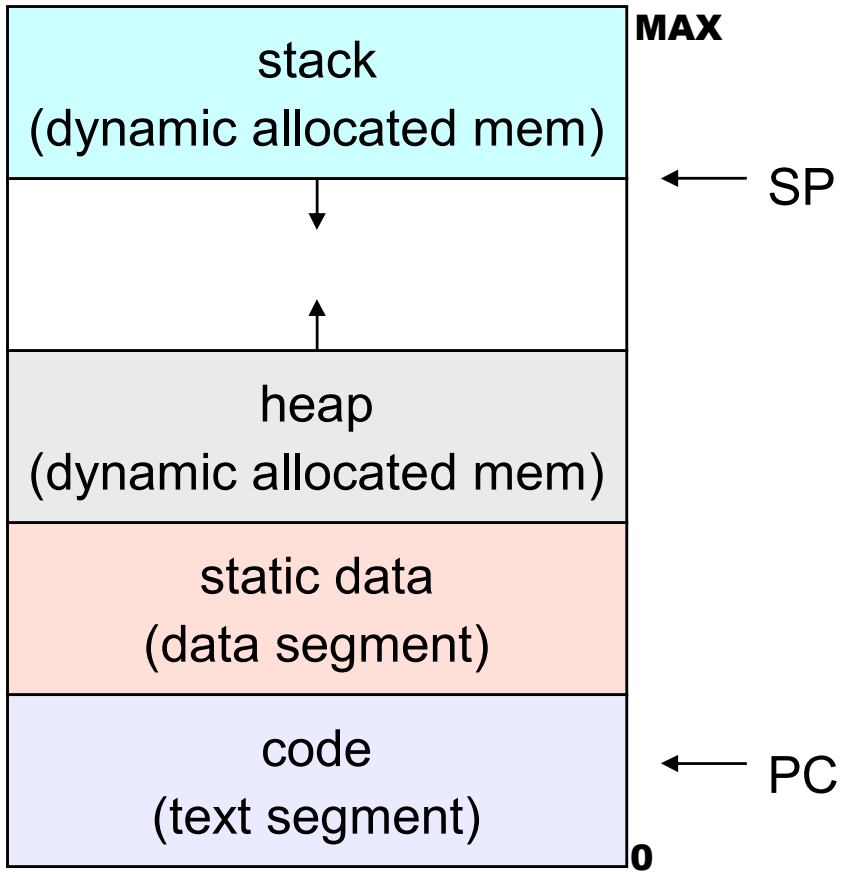
- Vimos...
  - SO = administrador de recursos + interfaz de programación.
  - Un poco de su evolución histórica.
  - Su misión fundamental.
  - Hablamos de multiprogramación.
  - Qué cosas son parte del SO y cuáles no.
- Vamos a ver...
  - Concepto de *proceso*. 
    También llamado *tarea* (task) o *trabajo* (job)
  - API del SO.

(3) Proceso

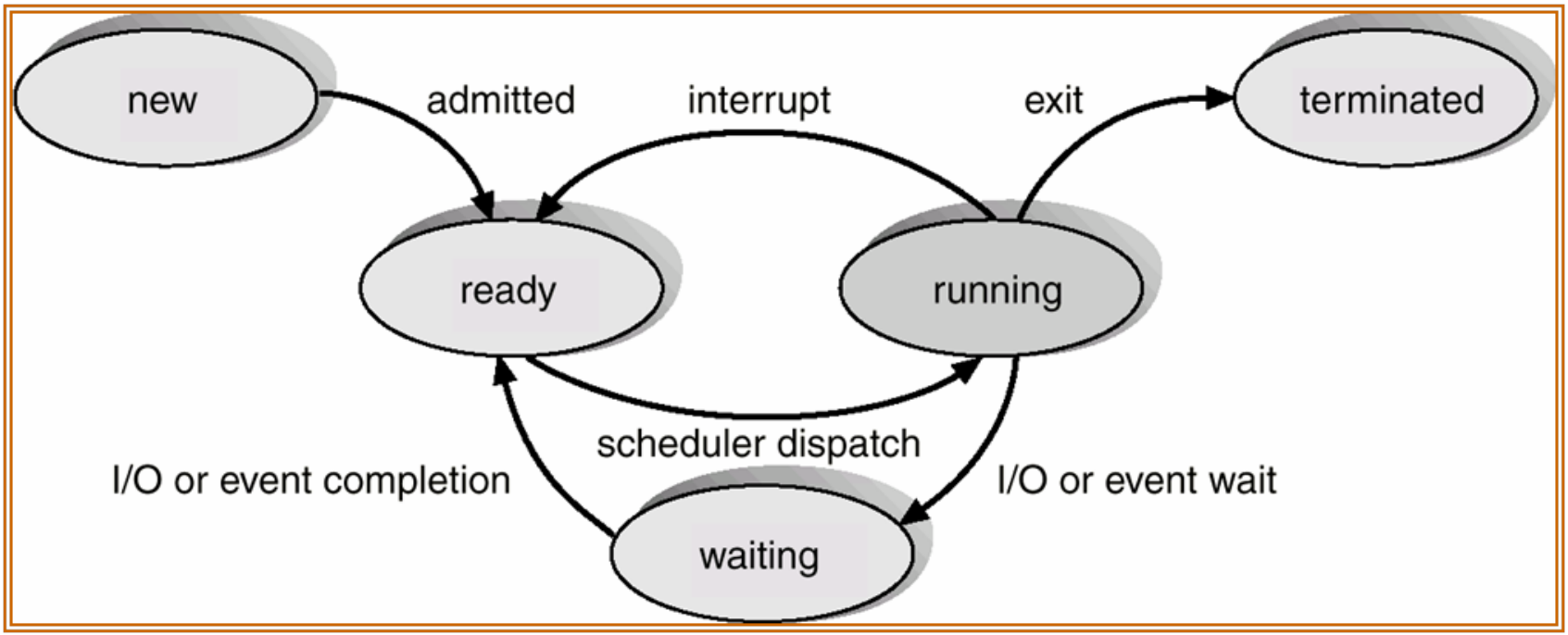
- *Programa: estático*
  - Texto escrito en algún lenguaje de programación.
  - Ese programa eventualmente se compila en código objeto, lo que también es un programa escrito en lenguaje de máquina.
- *Proceso: dinámico* 
  - Unidad de *ejecución*
    Cada proceso tiene un *identificador numérico único*, el *pid*.
  - Unidad de *scheduling*
  - Un proceso tiene *estado*

(4) Proceso: Representación en memoria

- *Texto*: código de máquina del programa.
  - Program Counter (PC): instrucción actual
- *Datos estáticos*
- *Memoria dinámica* (heap)
- *Pila de ejecución* (stack)
  - Stack Pointer (SP): tope de la pila



(5) Proceso: estado



(6) Proceso: estado

A medida que un proceso se ejecuta, va cambiando de estado de acuerdo a su actividad. 

- **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.

(7) Process Control Block (PCB)

- Estructura de datos con info del *contexto* del proceso 
  - Estado, contador de programa (PC), tope de pila (SP)
  - Registros de la CPU
  - Información para el scheduler (e.g., prioridad)
  - Información para el manejador de memoria, E/S, ...
- En linux:
  - `task_struct` en `linux/sched.h`
  - `thread_info` en `asm/thread_info.h`
  - `thread_struct` en `asm/processor.h`

(8) Proceso: cambios de estado

El SO ejerce las siguientes acciones sobre los procesos:

- Admite (o no) un proceso
- Otorga la CPU a un proceso
- Desaloja un proceso de la CPU

  Preemption

  Proceso es interrumpido por algún evento

  Blocking

  Proceso se bloquea en el acceso a un recurso

(9) Proceso: control de admisión

- ¿Qué es?

  Decidir si un proceso nuevo puede ser *admitido*

- ¿Para qué?

  Para evitar *sobrecargar* el sistema

- Carga (*load*):

  Cantidad de procesos activos

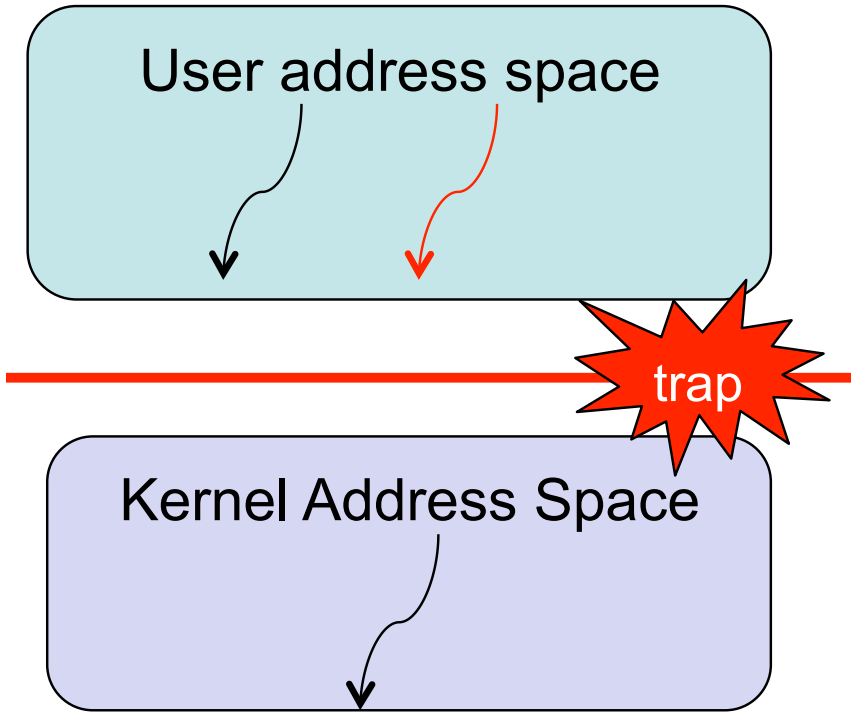
  Importa la carga promedio más que la carga puntual 

  Linux: `calc_load()` y `uptime` (`http://goo.gl/BCKcbp`)

```
$ uptime 11:33 up 1 day, 20:25, 2 users, load
averages: 2.98 2.83 2.69
```

(10) Procesos: ciclo de ejecución

- Realizar operaciones entre registros y direcciones de memoria *en el espacio de direcciones del usuario*
- Acceder a un servicio del *kernel* (*system call*)
  - Realizar entrada/salida a los dispositivos (E/S).
  - Lanzar un proceso hijo: `system()`, `fork()`, `exec()`.
- *Domain crossing*



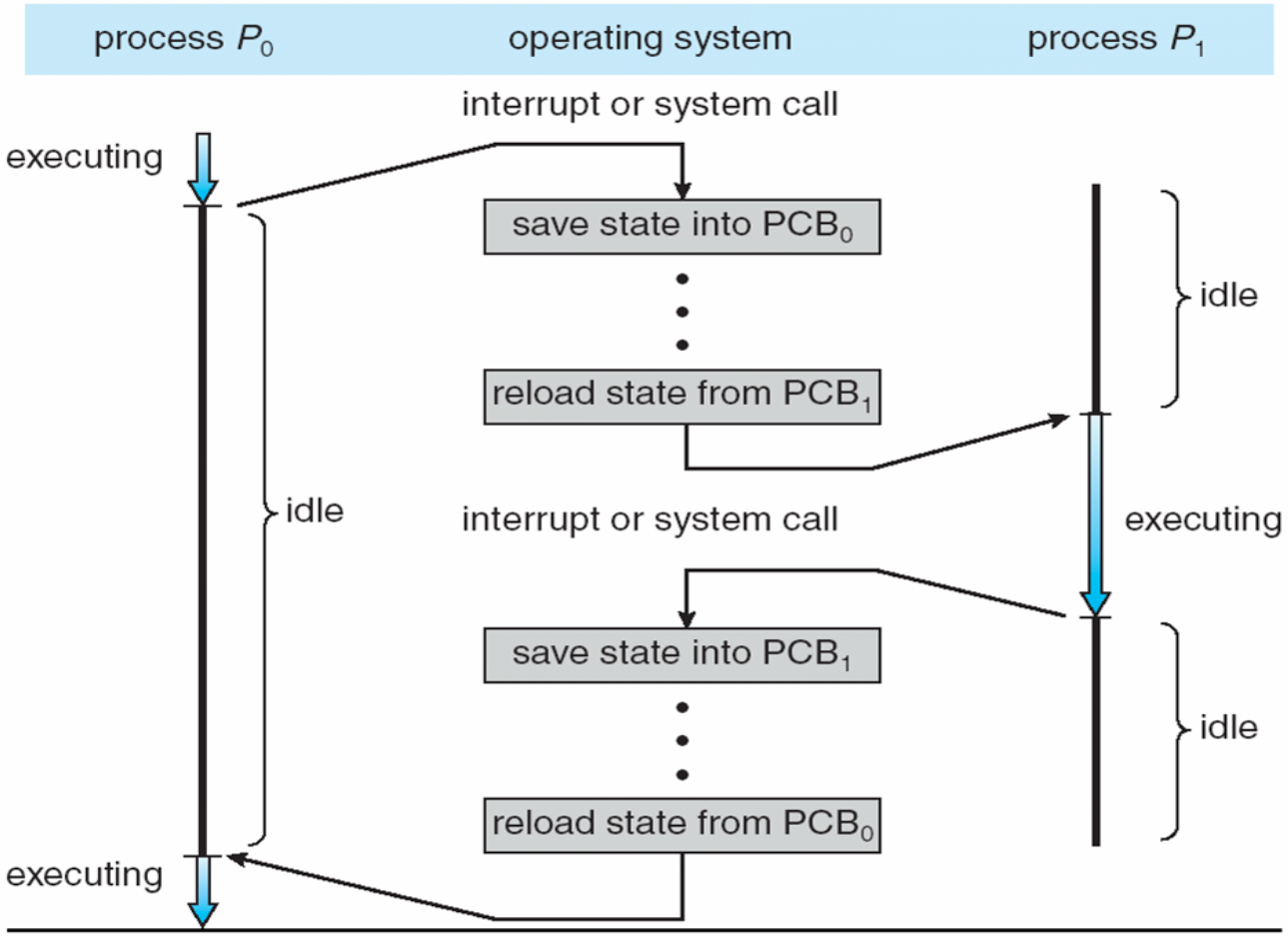
(11) Ejecución fuera del kernel

- ¿Por cuánto tiempo lo dejamos ejecutar?
  - Hasta que termina: Es lo mejor para el proceso, pero no para el sistema en su conjunto. Además, podría no terminar.
  - *time-sharing*: un "ratito" (*quantum*).
    Curiosidad: Bemer. Origins of timesharing. `http://goo.gl/83Ptc4`
- Los SO modernos hacen *preemption*: cuando se acaba el quantum, le toca el turno a *otro* proceso.
- Surgen dos preguntas:
  - Quién y cómo decide a quién le toca
    → *scheduling*
  - Qué significa hacer que se ejecute otro proceso
    → *context switch*

(12) Cambio de contexto (context switch)

- Para el proceso desalojado el SO debe:
  - Guardar el PCB.
  - Guardar los registros.
- Para el proceso asignado el SO debe:
  - Cargar el PCB.
  - Cargar los registros.
- El tiempo utilizado en cambios de contexto es tiempo muerto, no se está haciendo nada productivo.
- Dos consecuencias de esto:
  - *Duración*: Impacto en la arquitectura del HW
  - *Cantidad*: Quantum apropiado para minimizar los cambios 
- Implementación: colgarse de la interrupción del clock.

(13) Cambio de contexto (esquema)



(14) Cambio de contexto (artículos)

- Costo
  - Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. `http://goo.gl/GqGvKt`
  - Francis M. David, et al. 2007. Context switch overheads for Linux on ARM platforms. `http://goo.gl/pj9Dwj`
- Prevención
  - Jaaskelainen, et al. Reducing context switch overhead with compiler-assisted threading. 2008. `http://goo.gl/8th0dy`.
  - Kloukinas, Yovine. 2011. A model-based approach for multiple QoS in scheduling: from models to implementation. `http://goo.gl/4xL1JT`

(15) Actividades de un proceso (cont.)

- Llamadas al sistema (sys calls).
  - En todas ellas se debe llamar al kernel → domain crossing
- Entrada/Salida
- Terminación (`exit()`)
  - Liberar todos los recursos del proceso.
  - Status de terminación
  - Linux (C standard): `EXIT_SUCCESS`, `EXIT_FAILURE`
  - Este código de status le es reportado al padre.
    ¿Qué padre?

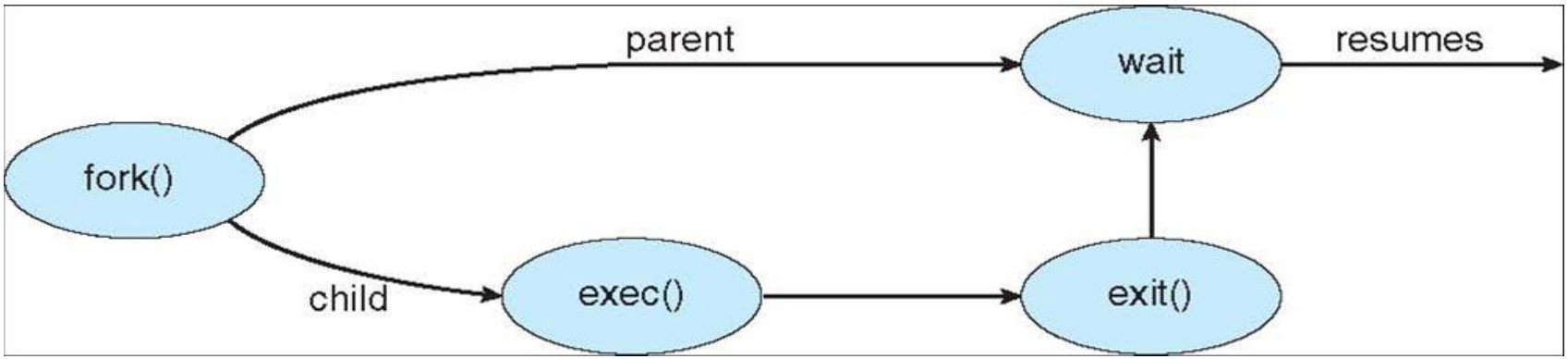
(16) Árbol de procesos

- Los procesos están organizados jerárquicamente.
- Cuando el SO comienza, lanza un proceso *root* o *init*.
- Por eso es importante la capacidad de lanzar un proceso hijo:
  - `fork()` crea un proceso hijo igual al actual (padre)
  - El resultado es el pid del proceso hijo
  - El proceso hijo ejecuta el mismo código que el padre
  - `exec()` reemplaza el código binario por otro
  - `wait()` suspende al padre hasta que termine el hijo
  - Cuando el hijo termina, el padre recibe el status del hijo

(17) Árbol de procesos (cont.)

Cuando lanzamos un programa desde el shell, ¿qué sucede?

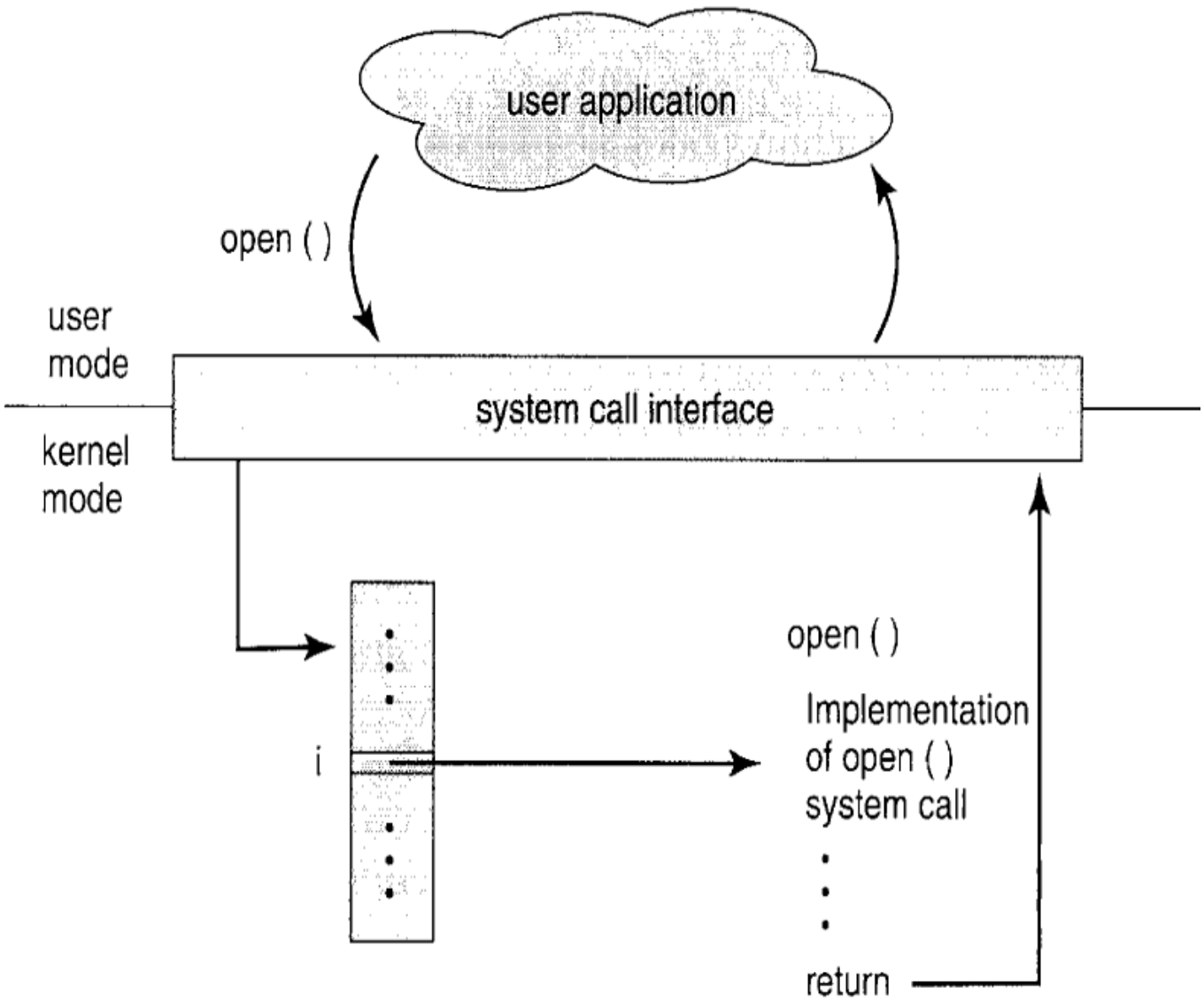
- El shell hace un `fork()`.
- El hijo hace un `exec()`.



Árbol de procesos: pstree

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

(18) Llamadas al sistema



(19) Tipos de llamadas al sistema

- Control de Procesos
- Administración de archivos
- Administración de dispositivos
- Mantenimiento de información
- Comunicaciones

(20) Ejemplos de llamadas al sistema

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

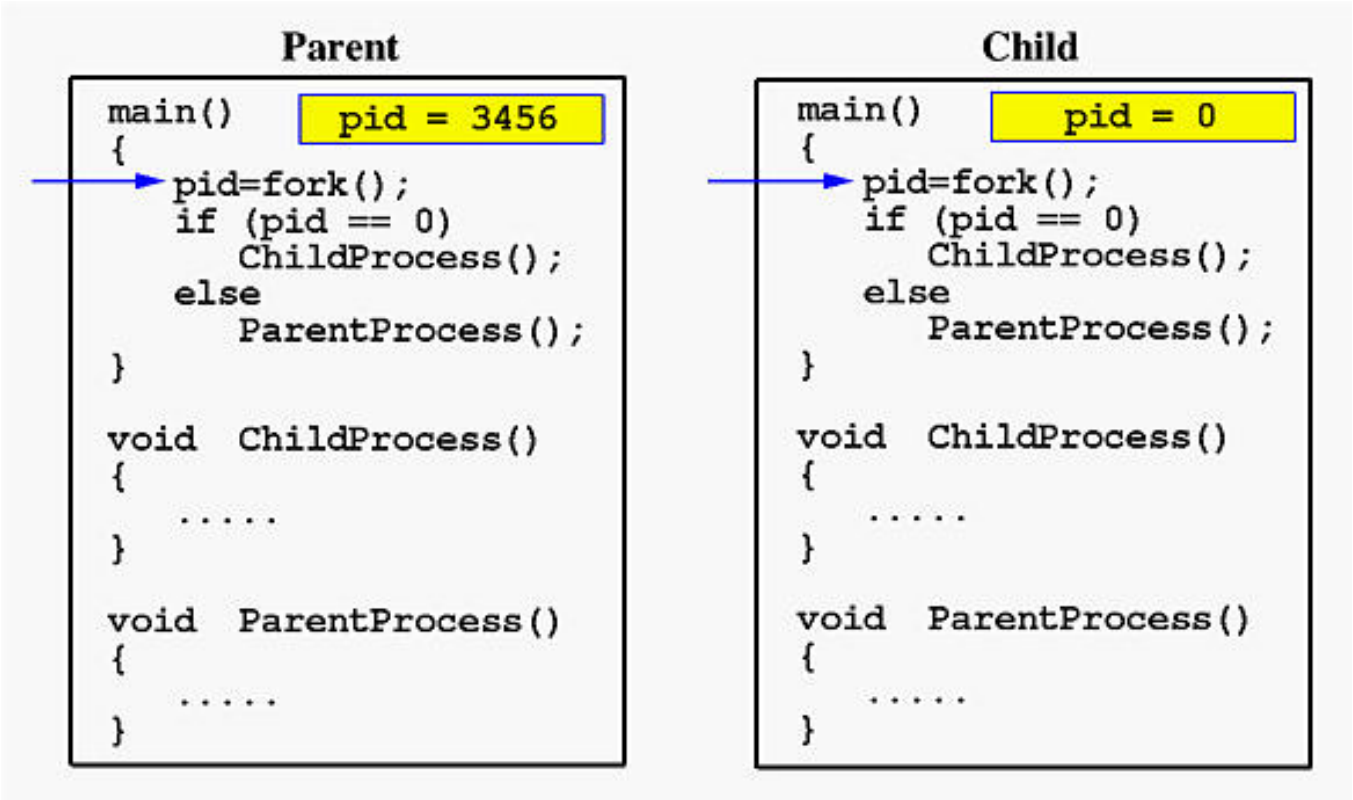
(21) POSIX

- 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).

- Creación y control de procesos

```
pid_t fork(void);
pid_t vfork(void);
// vfork crea un hijo sin copiar la memoria del padre
// El hijo tiene que hacer exec

int execve(const char *fn, char *const argv[],
    char *const envp[]);

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);

void exit(int status);

// Linux, no POSIX
int clone(...)
// El hijo comparte parte del contexto con el padre
// Usado para implementar threads
```

(23) Creación de procesos (fork)



(24) Creación de procesos (fork)

- Ejemplo tipo

```
1  int main(void) {
2    int foo = 0;
3    pid_t pid = fork();
4    if (pid == -1) exit(EXIT_FAILURE);
5    else if (pid == 0) {
6       printf("%d: Hello world\n", getpid());
7       foo = 1;
8    }
9    else {
10      printf("%d: %d created\n", getpid(), pid);
11      int s; (void)waitpid(pid, &s, 0);
12      printf("%d: %d finished(%d)\n", getpid(), pid, s);
13   }
14   printf("%d: foo(%p)= %d\n", getpid(), &foo, foo);
15   exit(EXIT_SUCCESS);
16 }
```

(25) Creación de procesos (fork)

- Ejemplos de ejecuciones posibles

```
$ ./main
3724: 3725 created
3725: Hello world
3725: foo(0x7fff5431fb6c)= 1
3724: 3725 finished(0)
3724: foo(0x7fff5431fb6c)= 0

$ ./main
3815: Hello world
3815: foo(0x7fff58c3eb6c)= 1
3814: 3815 created
3814: 3815 finished(0)
3814: foo(0x7fff58c3eb6c)= 0
```

(26) Creación de procesos (vfork)

- Ejemplo tipo

```
1  void bar(const char *fname) {
2      char *arg[] = { NULL, "Hello, world!", NULL};
3      char *env[] = { NULL };
4      char str[255];
5
6      sprintf(str, "%d", getpid());
7      arg[0] = str;
8
9      execve(fname, arg, env);
10 }
11
12 // Hijo
13     else if (pid == 0) {
14         foo = 1;
15         bar("./bar");
16     }
```

(27) Creación de procesos (vfork)

- Ejecución

```
$ ./main
1121: 1122 created
1122(1122): Hello, world!
1121: 1122 finished(0)
1121: foo(0x7fff53939b8c)= 1
```

- Manejo de archivos

```
// creación y apertura
int open(const char *pathname, int flags);

// lectura
ssize_t read(int fd, void *buf, size_t count);

// escritura
ssize_t write(int fd, const void *buf, size_t count);

// actualiza la posición actual
off_t lseek(int fd, off_t offset, int whence);
// whence = SEEK_SET -> comienzo + offset
// whence = SEEK_CUR -> actual + offset
// whence = SEEK_END -> fin + offset
```

(29) Inter-Process Communication (IPC)

- Hay varias formas de IPC:
  - Memoria compartida.
  - Algún otro recurso compartido (archivo, base de datos, etc.).
  - Pasaje de mensajes
- BSD/POSIX *sockets*
  - Un socket es el extremo de una comunicación (enchufe)
  - Para usarlo hay que conectarlo
  - Una vez conectado se puede leer y escribir de él
  - Hacer IPC es como hacer E/S. 
  - Los detalles los vamos a ver en la práctica.

(30) IPC: modos

- Sincrónico
  - El emisor no termina de enviar hasta que el receptor no recibe.
  - Si el mensaje se envió sin error suele significar que también se recibió sin error.
  - En general involucra bloqueo del emisor.
- Asincrónico
  - El emisor envía algo que el receptor va a recibir en algún otro momento.
  - Requiere algún mecanismo adicional para saber si el mensaje llegó.
  - Libera al emisor para realizar otras tareas, no suele haber bloqueo, aunque puede haber un poco (por ejemplo, para copiar el mensaje a un buffer del SO).

(31) 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.
  - Introdujimos IPC.