Introducción: procesos y API del SO

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

Sistemas Operativos, primer cuatrimestre de 2016

(2) Algunas aclaraciones preliminares

- Diapos numeradas.
- Su NO pregunta SÍ molesta.
- Las siguientes cosas no son equivalentes (de a pares):
  1. Presenciar esta clase.
  2. Leer los apuntes.
  3. Presenciar las clases prácticas sobre el tema.
  4. Hacer los talleres.
  5. Hacer las prácticas.
- ¿Cómo sé si entendí los temas?
  - Los prácticos: si me salen los ejercicios (en un tiempo razonable).
  - Los teóricos: si soy capaz de explicarlos con mis propias palabras.
- Bibliografía
  - Silberschatz, A. and Galvin, P.B. and Gagne, G., *Operating system concepts*, Addison-Wesley.
  - Tanenbaum, A.S., *Modern operating systems*, Prentice Hall New Jersey.

(3) Qué es un SO y por qué dedicarle toda una materia

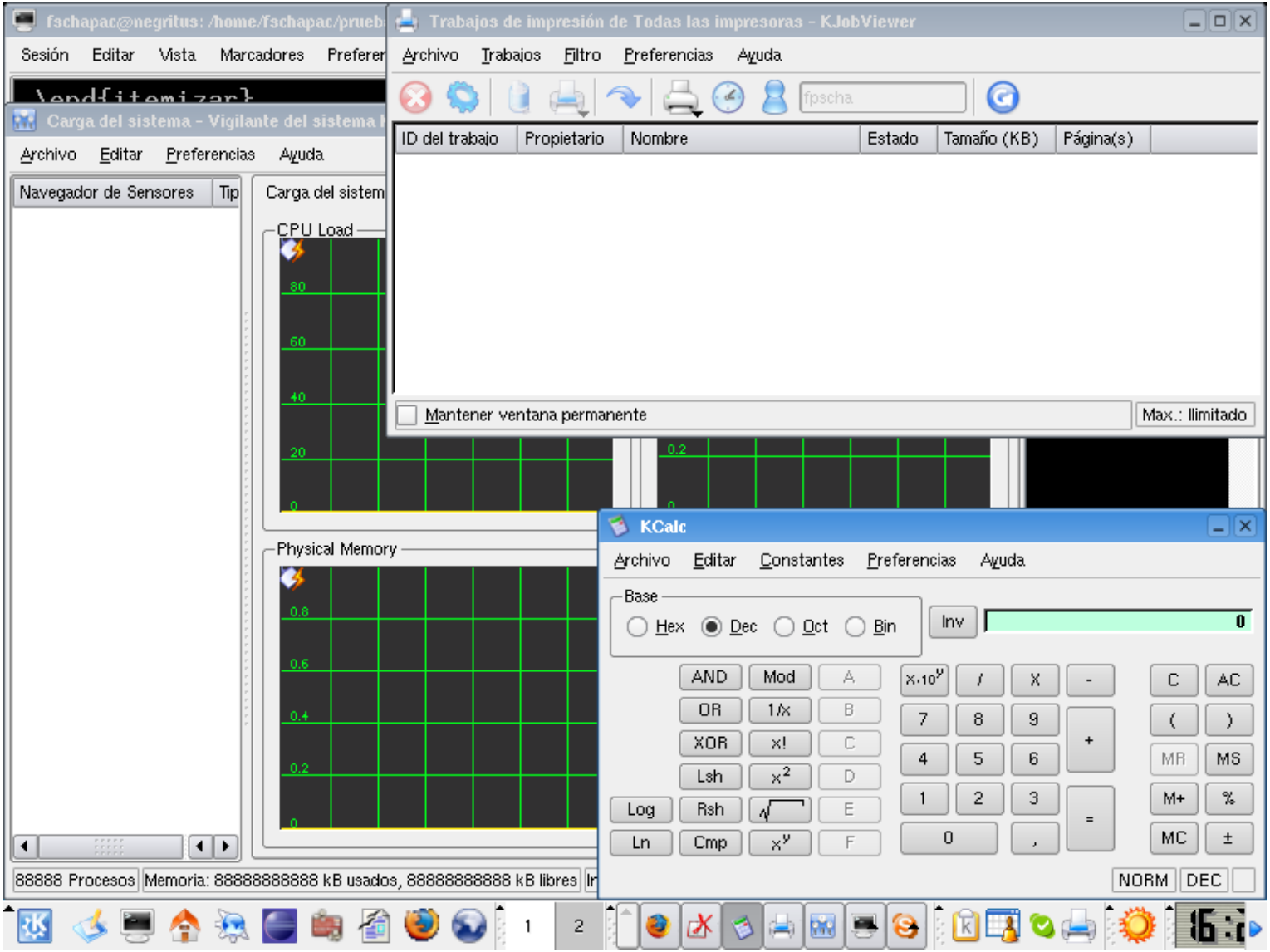
- Una forma de dividir a los sistemas informáticos:
  - Hardware (lo que se puede patear).
  - Software específico (lo que sólo se puede insultar).
- Hace falta un intermediario entre ellos 
  - ...para que el software específico no se tenga que preocupar con detalles de bajo nivel del HW (visión de usuario).
  - ...para que el usuario use correctamente el HW (visión del propietario del HW).
- Por qué toda una materia:
  - Porque esta capa es suficientemente específica e interesante como para estudiarla en detalle.
  - Porque aquí surgen problemas muy interesantes.
  - Y porque es el marco "natural" para estudiar algunos de esos problemas, que son de carácter más general.

(4) Qué es y qué no es un SO

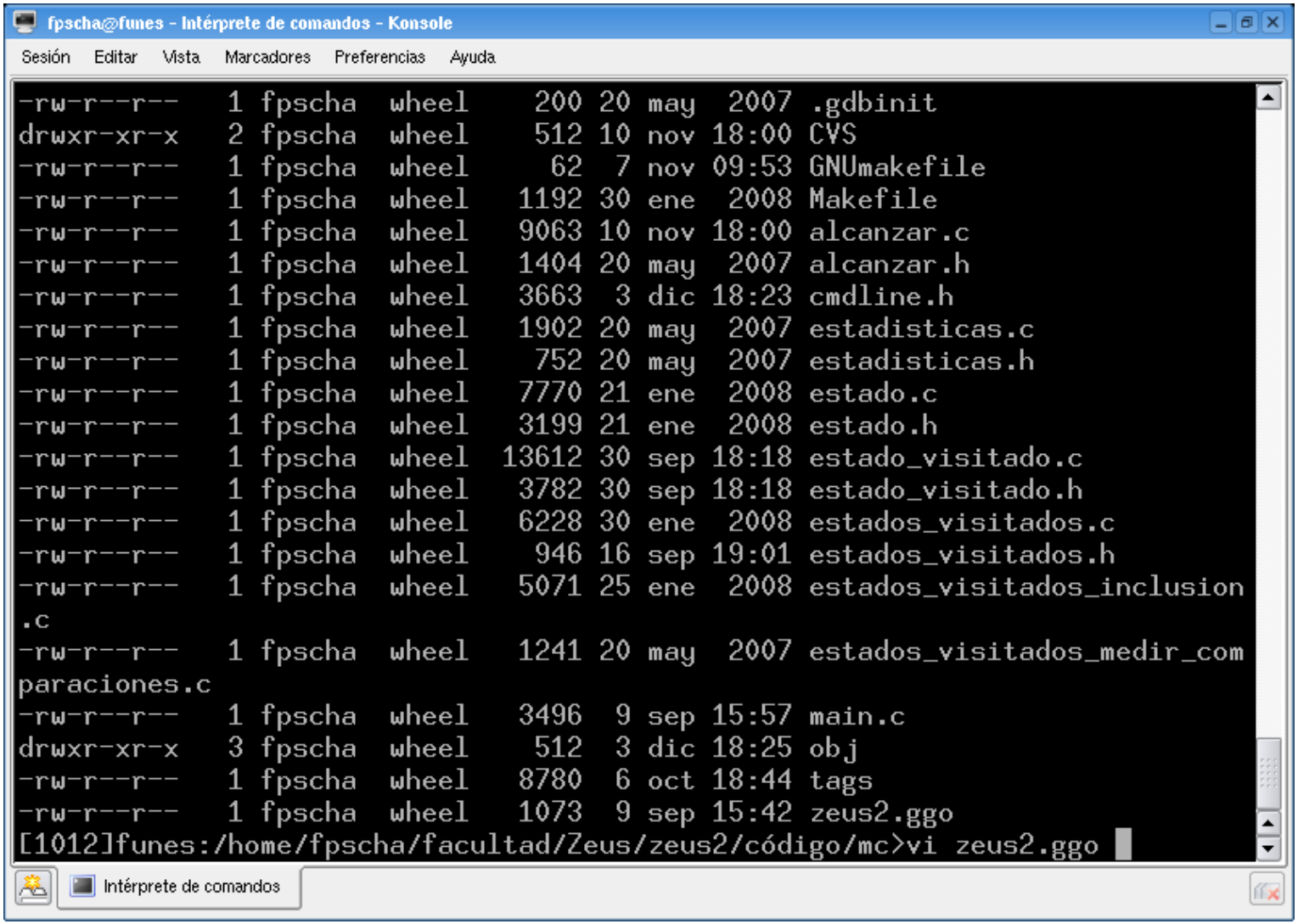
- Veamos cuánto ocupan algunos sistemas operativos:
- Ubuntu Linux 9.10: “At least 4 GB of disk space”
- FreeBSD 7.2: 5 CDs de instalación.
- Windows Seven: 16 GB para la edición “home basic”, 40 GB para las otras.
- ¡¿Todo eso es necesario?!

(5) Qué es y qué no es un SO (cont.)

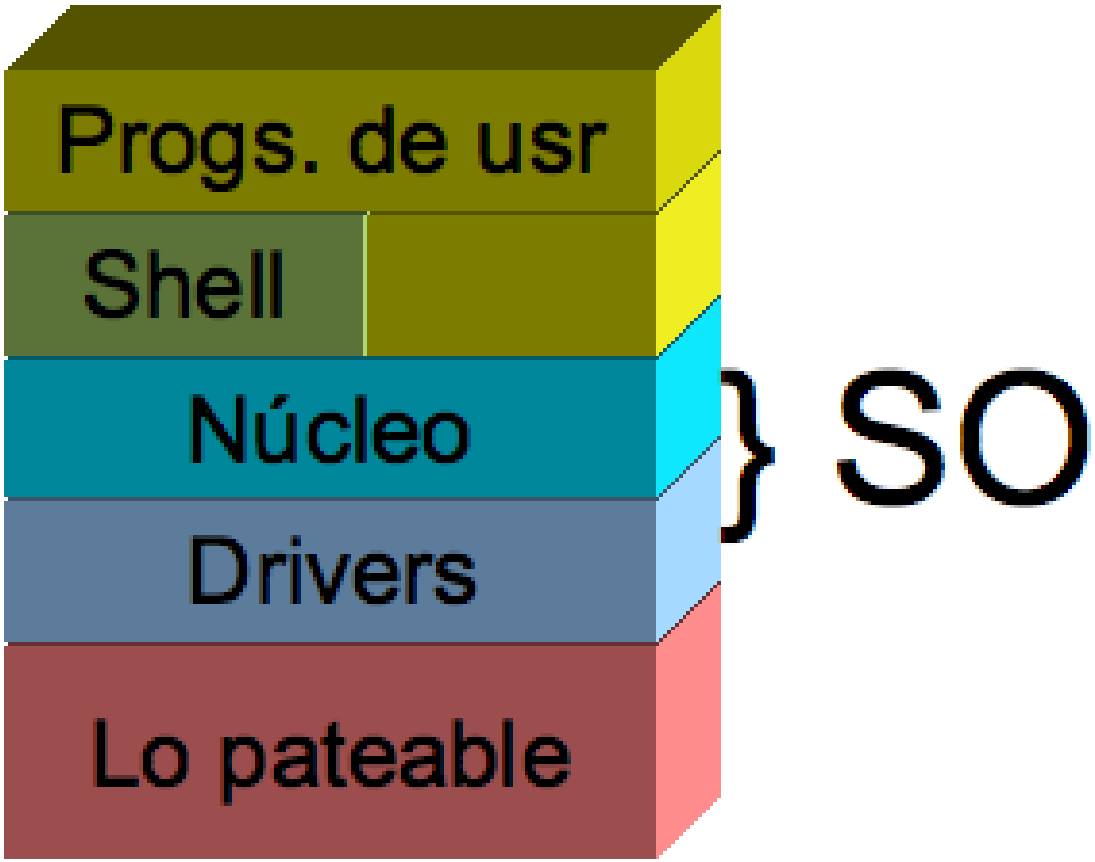
(6) Qué es y qué no es un SO (cont.)



(7) Qué es y qué no es un SO (cont.)



(8) Qué es y qué no es un SO (cont.)



(9) Elementos básicos de un SO

- Drivers: programas que son parte del sistema operativo y manejan los detalles de bajo nivel relacionados con la operación de los distintos dispositivos.
- Núcleo o Kernel: es el SO propiamente dicho, su parte central. Se encarga de las tareas fundamentales y contiene los diversos subsistemas que iremos viendo en la materia.
- Intérprete de comandos o Shell: un programa más, que muchas veces es ejecutado automáticamente cuando comienza el SO, que le permite al usuario interactuar con SO. Puede ser gráfico o de línea de comandos. Ejemplos en Unix: `sh`, `csh`, `ksh`, `bash`.
- Proceso: un programa en ejecución más su espacio de memoria asociado y otros atributos.

(10) Elementos básicos de un SO (cont.)

- Archivo: secuencia de bits con un nombre y una serie de atributos que indican permisos, etc.
- Directorio: colección de archivos y directorios que contiene un nombre y se organiza jerárquicamente.
- Dispositivo virtual: una abstracción de un dispositivo físico bajo la forma, en general, de un archivo, de manera tal que se pueda abrir, leer, escribir, etc.
- Sistema de archivos: es la forma de organizar los datos en el disco para gestionar su acceso, permisos, etc.

(11) Elementos básicos de un SO (cont.)

- Directorios del sistema: son directorios donde el propio SO guarda archivos que necesita para su funcionamiento, como por ejemplo, `/boot`, `/devices` o `C:\Windows\system32`.
- Binario del sistema: son archivos, que viven en los directorios del sistema. Si bien no forman parte del kernel, suelen llevar a cabo tareas muy importantes o proveer las utilidades básicas del sistema. Ejemplo:
  - `/usr/sbin/syslogd`: es el encargado de guardar los eventos del sistema en un archivo.
  - `/bin/sh`: el Bourne Shell.
  - `/usr/bin/who`: indica qué usuarios están sesionados en el sistema.
- Archivo de configuración: es un archivo más, excepto porque el sistema operativo saca de allí información que necesita para funcionar. Por ejemplo, `/etc/passwd` o `C:\Windows\system32\user.dat`.

(12) Elementos básicos de un SO (cont.)

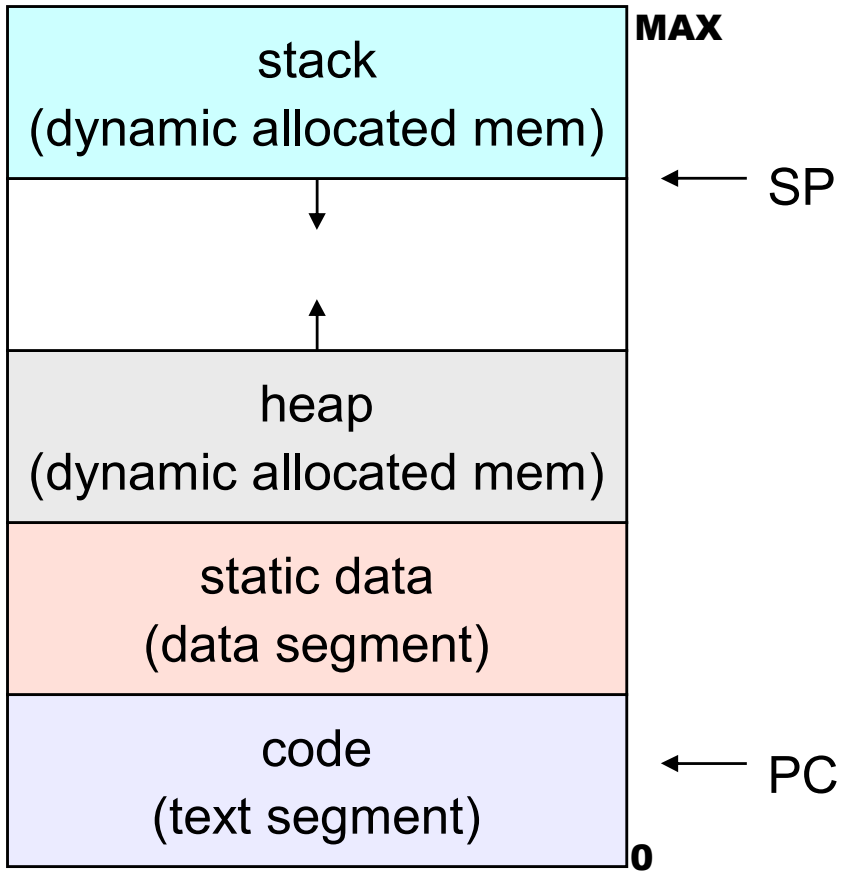
- Usuario: la representación, dentro del propio SO, de las personas o entidades que pueden usarlo. Sirve principalmente como una forma de aislar información entre sí y de establecer limitaciones.
- Grupo: una colección de usuarios.

(13) Proceso

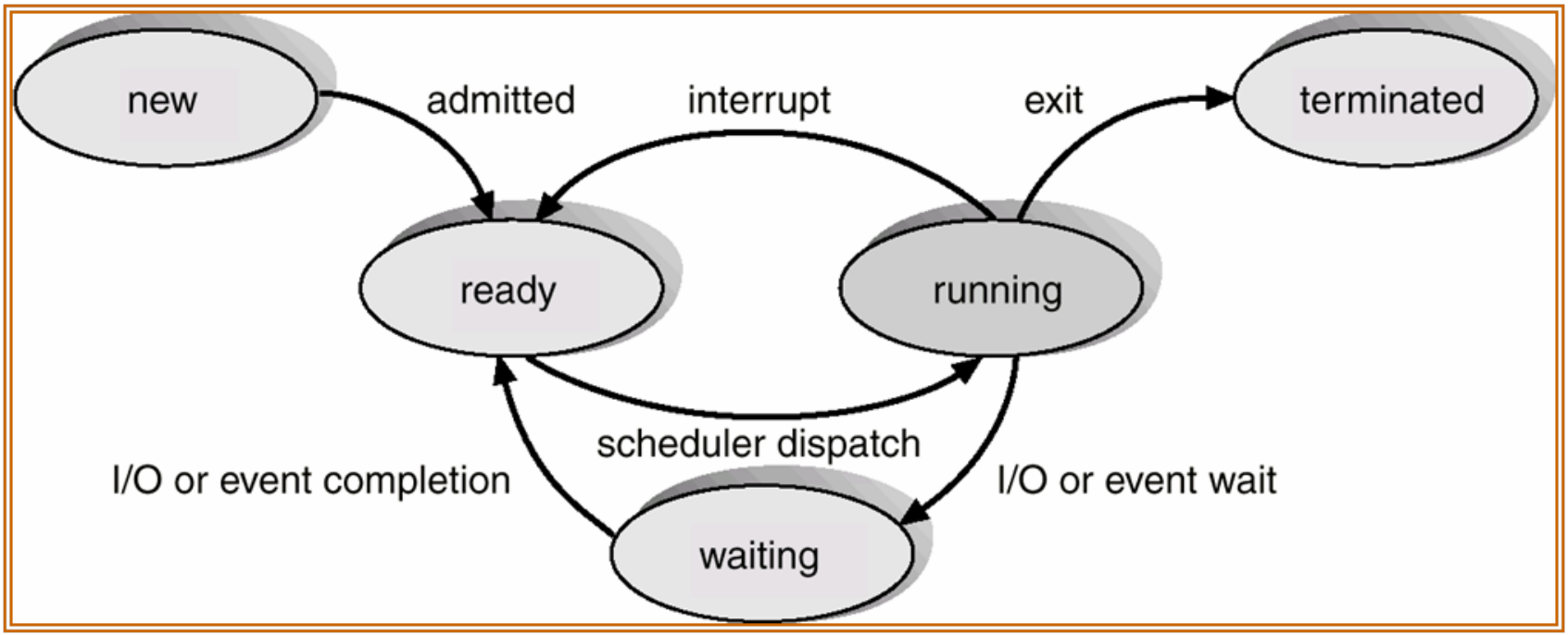
- *Programa: estático*
  - Texto escrito en algún lenguaje de programación.
  - Ese programa eventualmente se compila en código objeto, lo que también es un programa escrito en lenguaje de máquina.
- *Proceso: dinámico* 
  - Unidad de *ejecución*
    Cada proceso tiene un *identificador numérico único*,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

(14) 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



(15) Proceso: estado



(16) Proceso: estado

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

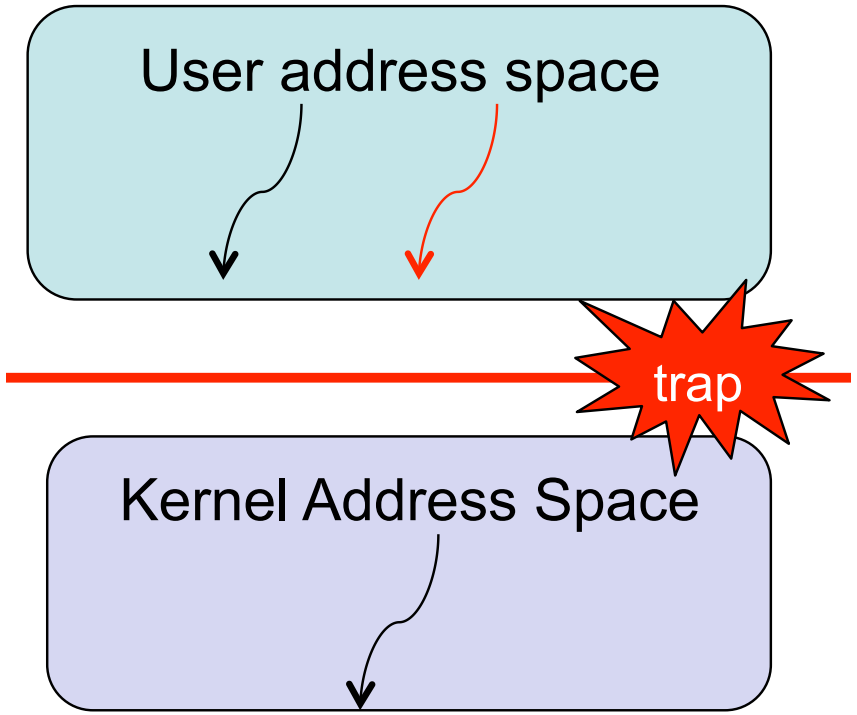
- **Corriendo**: está usando la CPU.
- **Bloqueado**: no puede correr hasta que algo externo suceda (típicamente E/S lista).
- **Listo**: el proceso no está bloqueado, pero no tiene CPU disponible como para correr.

(17) 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`

(18) 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*



(19) Actividades de un proceso (cont.)

- Llamas al sistema (sys calls).
  - En todas ellas se debe llamar al kernel $\rightarrow$ domaing 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?

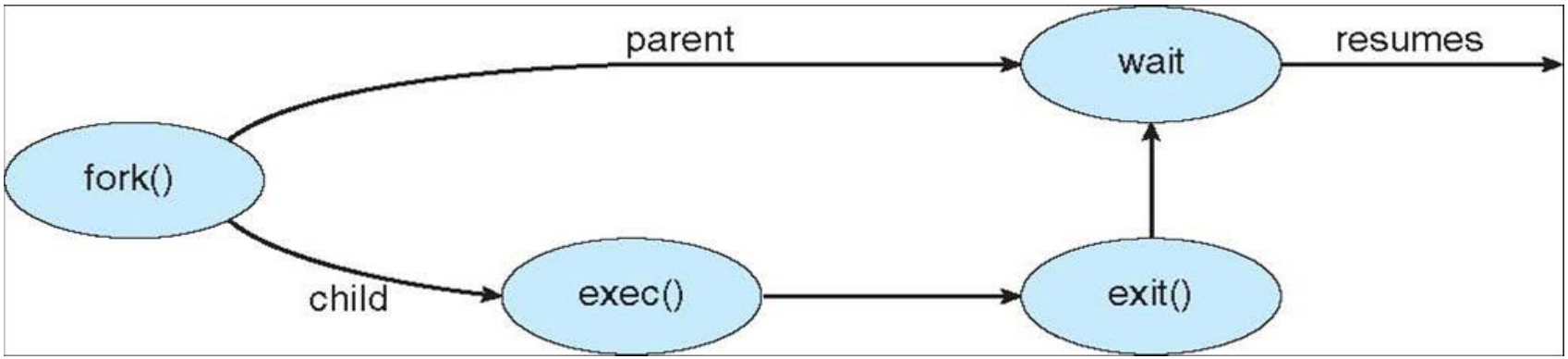
(20) Árbol de procesos

- 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

(21) Árbol de procesos (cont.)

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

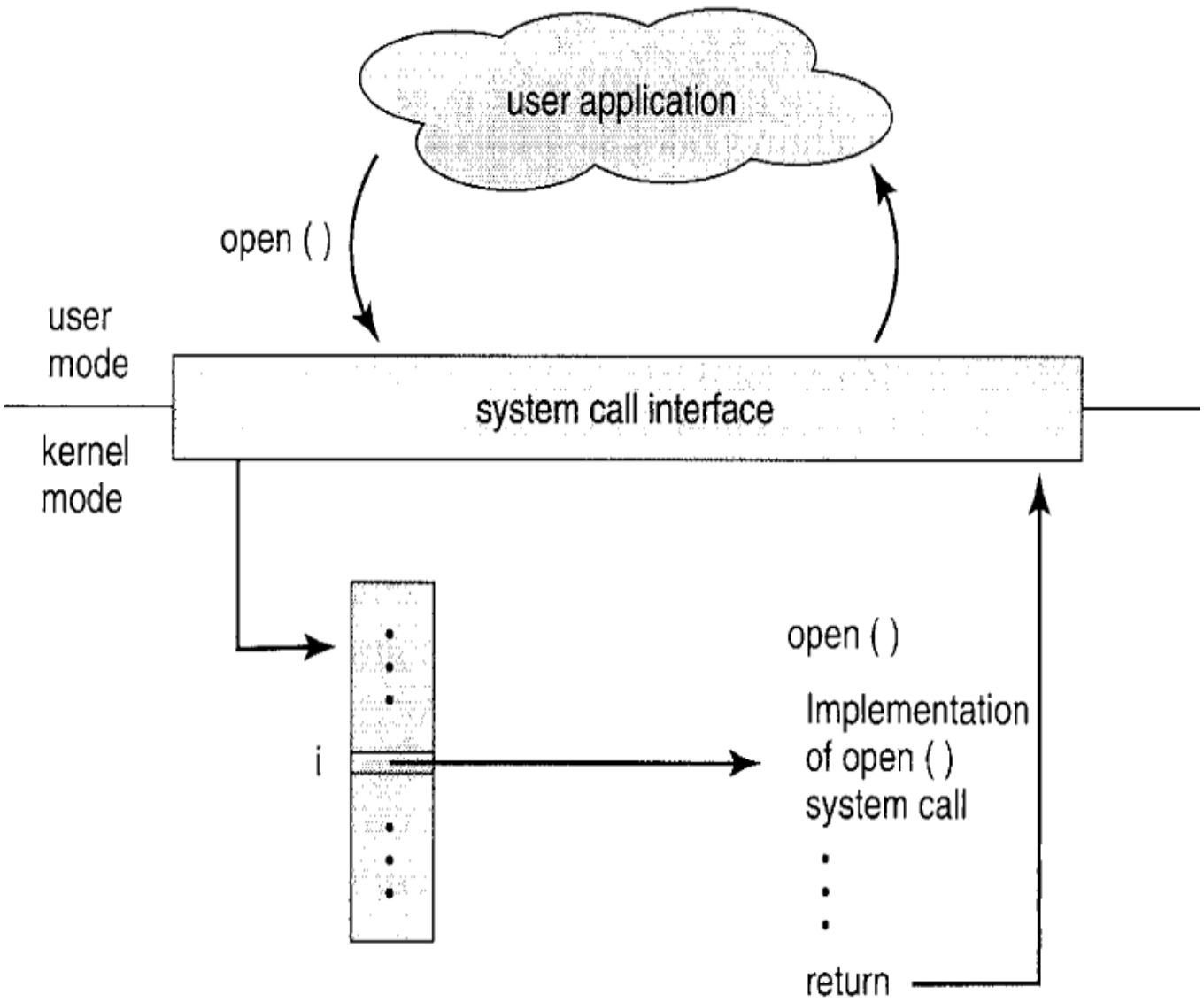
- El shell hace un `fork()`.
- El hijo hace un `exec()`.



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,...
```

(22) Llamadas al sistema



(23) Tipos de llamadas al sistema

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

(24) 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).

(25) API

- Creación y control de procesos

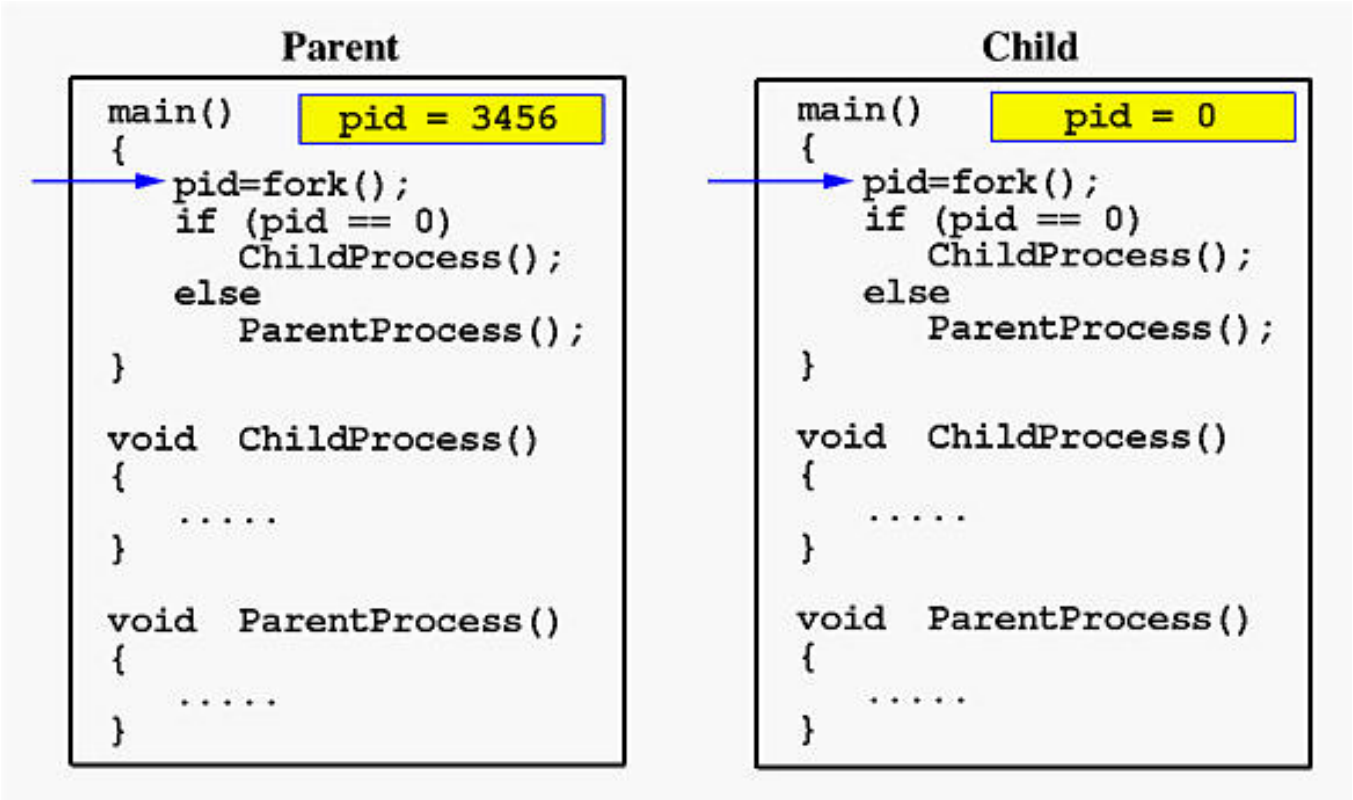
```
pid_t fork(void);
pid_t vfork(void);
// vfork crea un hijo sin copiar la memoria del padre
// El hijo tiene que hacer exec

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);
```

(26) Creación de procesos (fork)



(27) 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 }
```

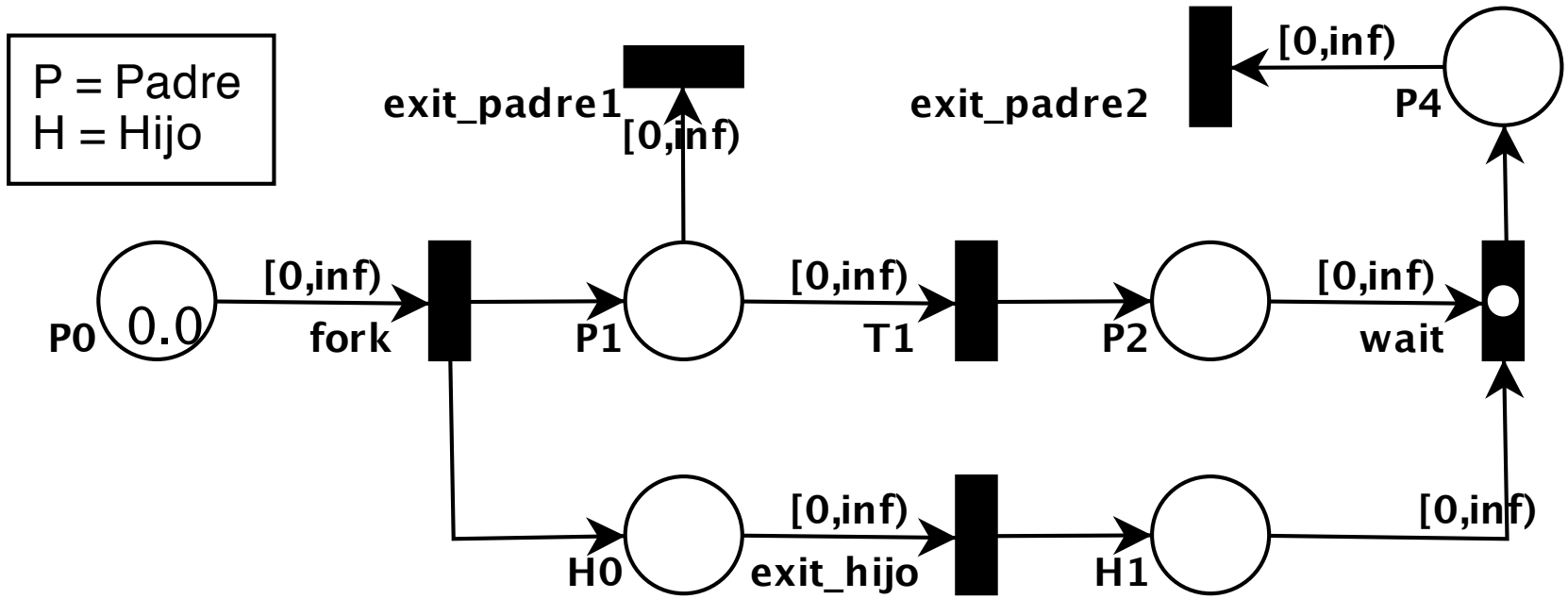
(28) Creación de procesos (fork)

- 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
```

(29) Modelo formal (Petri Net)



Herramienta: `http://www.tapaal.net/`

(30) 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     }
```

(31) 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
```

(33) 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.

(34) 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).

(35) Dónde estamos

- Vimos
  - Qué es (y qué no es) parte del sistema operativo.
  - 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.
- En el taller:
  - Vamos a entender IPC más en detalle.
  - Vamos a ver en la práctica varios de los conceptos de hoy.