

# Procesos y API del SO

Fernando Schapachnik<sup>1</sup>

<sup>1</sup>Departamento de Computación, FCEyN,  
Universidad de Buenos Aires, Buenos Aires, Argentina

Sistemas Operativos, primer cuatrimestre de 2018


## (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

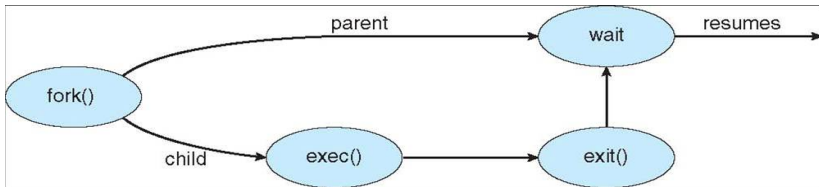
- El proceso indica al sistema operativo que ya puede liberar todos sus recursos (`exit()`).
- Además, indica su status de terminación (usualmente, un código numérico).
- Este código de status le es reportado al padre.
- ¿Qué padre?

## (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 *root* o *init*.
- 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.
  - Por ende: `fork() + wait() = system()`
  - 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.
- 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.

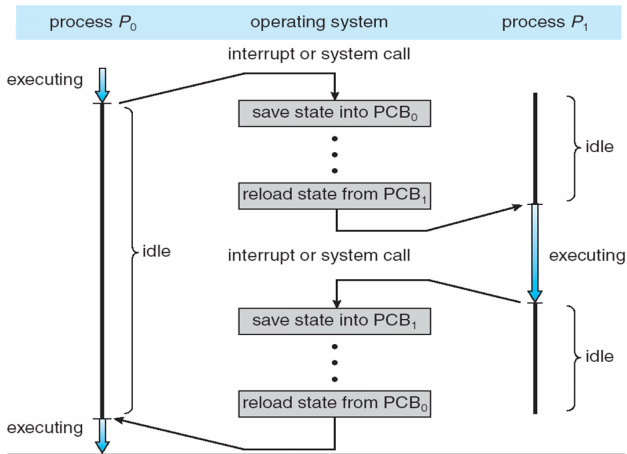
## (11) Scheduler

- 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. ⚠
- Este componente nos agrega una nueva forma de pensar a los procesos:
- Un proceso es también una *unidad 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) 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.



## (16) 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).

## (17) 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.

## (18) 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”.

## (19) 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). Lo veremos más adelante.
- Multiuso:

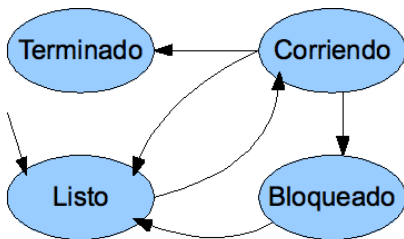


## (20) Multiprogramación desde el código

- 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).


## (21) 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.

## (22) 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.

## (23) Un proceso, varias visiones


- Un proceso se puede pensar como un espacio de memoria.
- O como una entrada en la tabla de procesos.
- A nivel conceptual, un proceso es también una *unidad de scheduling*.




## (24) IPC

- Los procesos aislados no son de demasiada utilidad en el mundo real.
- Además de hacer E/S hacia los dispositivos, es deseable que puedan intercambiar información entre sí.
- La comunicación entre procesos suele llamarse *IPC*, por *InterProcess Communication*. 
- Hay varias formas de IPC:
  - Memoria compartida.
  - Algún otro recurso compartido (por ejemplo, archivo, base de datos, etc.).
  - Pasaje de mensajes.
- Vamos a concentrarnos en el pasaje de mensajes.
- En particular, entre procesos de la misma máquina (aunque esto no es obligatorio).

## (25) IPC (cont.)

- Los SO brindan varias interfaces para hacer IPC:
  - Unix SysV *Transport Layer Interface*.
  - BSD Sockets. Esta es la línea evolutivamente ganadora.
- La idea de los sockets es la siguiente:
  - Un socket es el extremo de una comunicación. Piensen en el enchufe de la pared.
  - Luego de crearlo tengo que unirlo a algo, a una forma de comunicación concreta. Sería como conectar el enchufe a los cables.
  - Una vez que tengo un socket conectado puedo leer y escribir de él como si fuese un dispositivo.
  - Es decir, vale todo lo que vimos relacionado con bloqueo y scheduling.
  - Los detalles los vamos a ver en la práctica.
  - Lo importante es que para un proceso, hacer IPC es como hacer E/S. 

## (26) IPC sincrónico/asincrónico

- La comunicación entre procesos puede ser sincrónica o asincrónica: 
- Sincrónica
  - 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.
- Asincrónica
  - 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).

## (27) 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.
- En el taller:
  - Vamos a entender IPC más en detalle.
  - Vamos a ver en la práctica varios de los conceptos de hoy.
- En la próxima teórica:
  - Nos metemos a fondo con scheduling.

## (28) Cambio de contexto (bibliografía)

- 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>