Contenido 2: Procesos

Lesly Reyes
lareyes2@uc.cl

Álvaro Quintana
alvaroq@uc.cl

Departamento de Ciencias de la Computación
Pontificia Universidad Católica Chile

August 14, 2024

Contenido

Procesos
- Composición y representación

Representación
- Representación: ¿Cómo lo ve el Sistema Operativo?
- Ejecución de un proceso
- Proceso: ¿Cómo creamos procesos?
- Terminando Procesos

Señales, Huérfanos, y zombies

Material Adicional

Composición y representación

¿Qué es un proceso?

Un proceso es un programa en ejecución.

- Proceso = Código (programa) + Recursos
- ¿Qué recursos?
    - Memoria, archivos
    - Sockets, dispositivos de I/O, bibliotecas, etc . . .

Multiprogramación permite mantener múltiples procesos en memoria.

Composición y representación

¿Qué es un proceso?

- La CPU puede "atender" en algún orden (scheduling) los procesos.
- Cuando la CPU atiende "simultáneamente" varios procesos tenemos multitasking.
- Sigue siendo cierto que **un nucleo solo puede atender un proceso en el mismo instante de tiempo**.
- Lo que pasa es que puede cambiar la cpu de proceso en milisegundos, por eso da la impresión que multiples procesos usan una cpu.

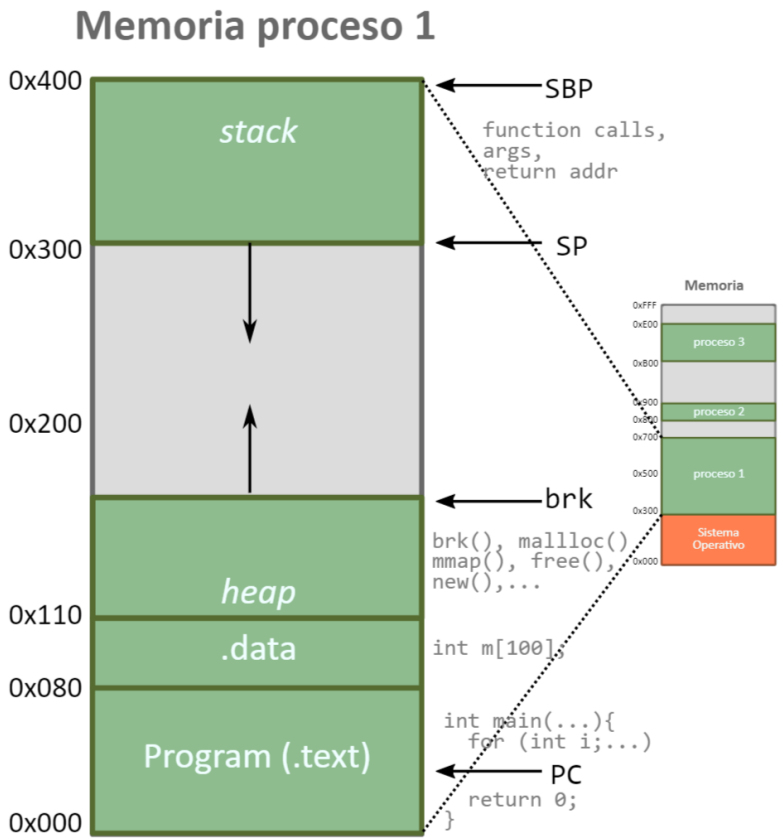
Composición y representación

¿Qué es un proceso?

- **Código** (.text, información estática)
- **Datos** (.data): variables globales
- **Heap** Memoria asignada dinámicamente (durante la ejecución)
- **Stack** Cada item del stack representa un llamado a función (call frame), y contiene:
  - Parámetros
  - Variables locales
  - Lugar de retorno (donde estaba la ejecución anterior, PC)
  - Stack base pointer (SBP) punto de inicio del stack
  - Stack pointer límite inferior del stack. De donde se sacan y ponen direcciones.

Composición y representación

¿Qué es un proceso?

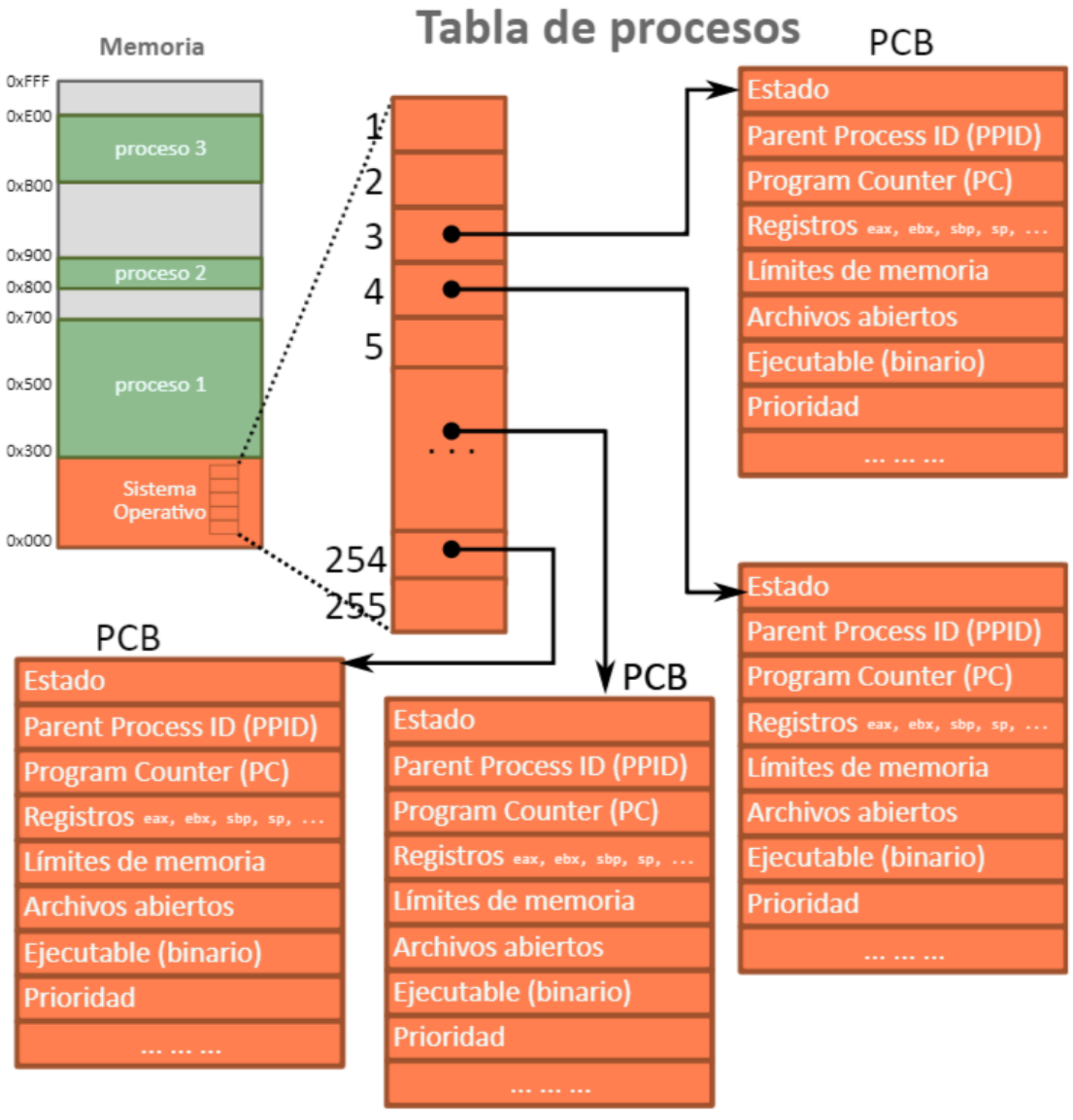


Representación: ¿Cómo lo ve el Sistema Operativo?

Sistema Operativo mantiene Tabla de Procesos. Información de cada proceso almacenado en su **Process Control Block (PCB)**

- Estado
- Identificador (PID, Process ID)
- Program Counter (PC)
- Registros de CPU: estado de ejecución
- Información de scheduling: prioridades, tipo de cola, ...
- Información de memoria: límites, tabla de páginas/segmentos, ...
- Contabilidad (accounting)
- Información de I/O: archivos y dispositivos abiertos, ...
- PCB pueden tener mucha información. Miremos en include/linux/sched.h

Representación: ¿Cómo lo ve el Sistema Operativo?



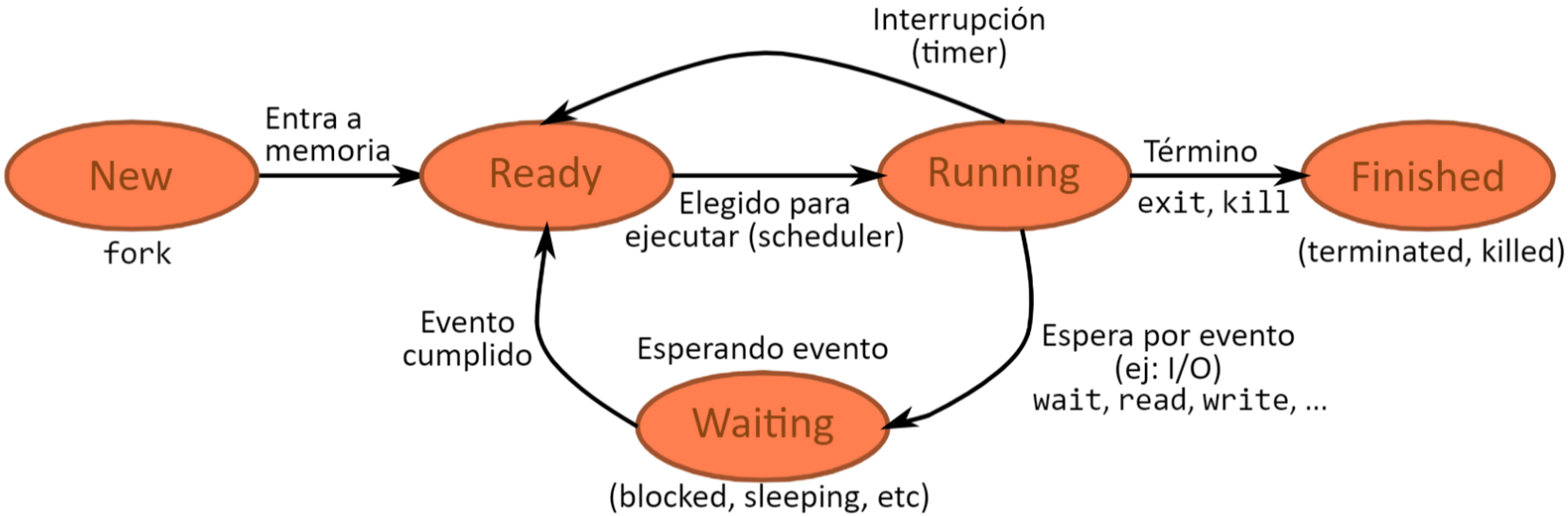
Estados de un proceso

Un proceso en ejecución puede cambiar de estado

- New : En creación
- Running : En ejecución
- Waiting : Esperando (I/O, signal)
- Ready : Listo para ejecutar. Esperando asignación de CPU
- Terminated: Ejecución terminada

Estados de un proceso

Un proceso en ejecución puede cambiar de estado

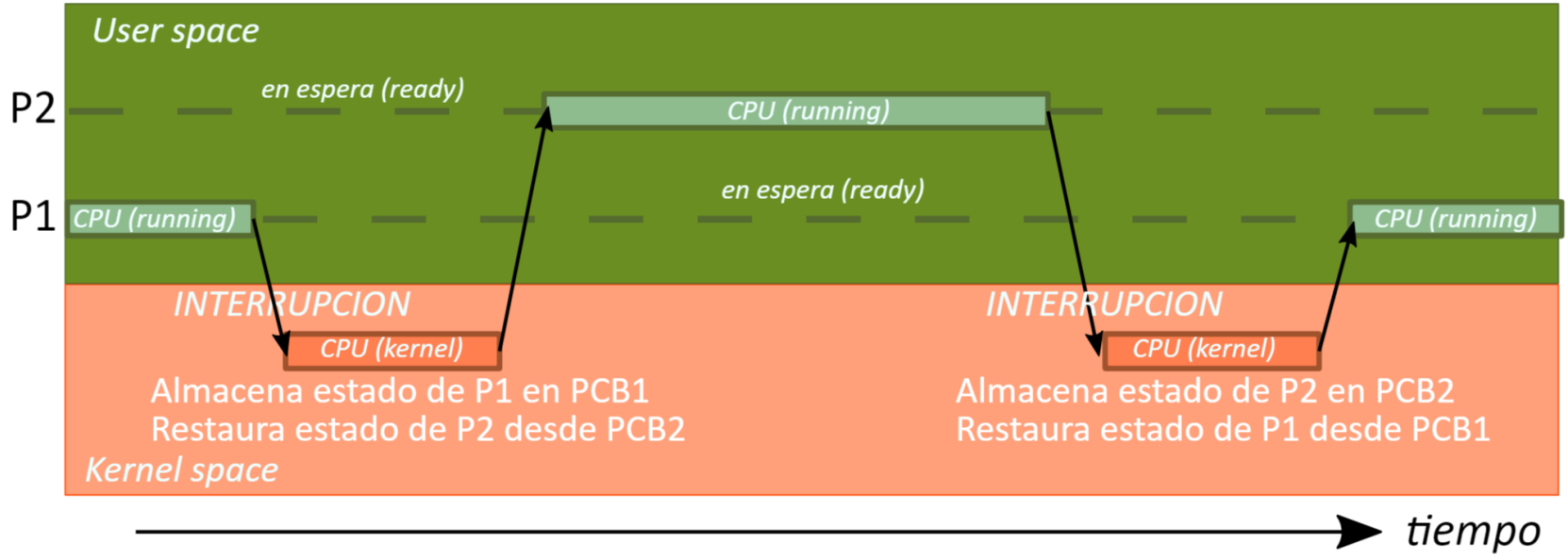


Estados de un proceso

El cambio de procesos se conoce como Context Switch

Multitasking

- Sistema Operativo actúa luego de una interrupción (syscall, timer, evento, . . .).
- Sistema Operativo almacena estado de registros de P1 en PCB1. P1 queda "en pausa".
- Sistema Operativo restaura estado de registros desde PCB2.
- Proceso P2 continúa su ejecución.



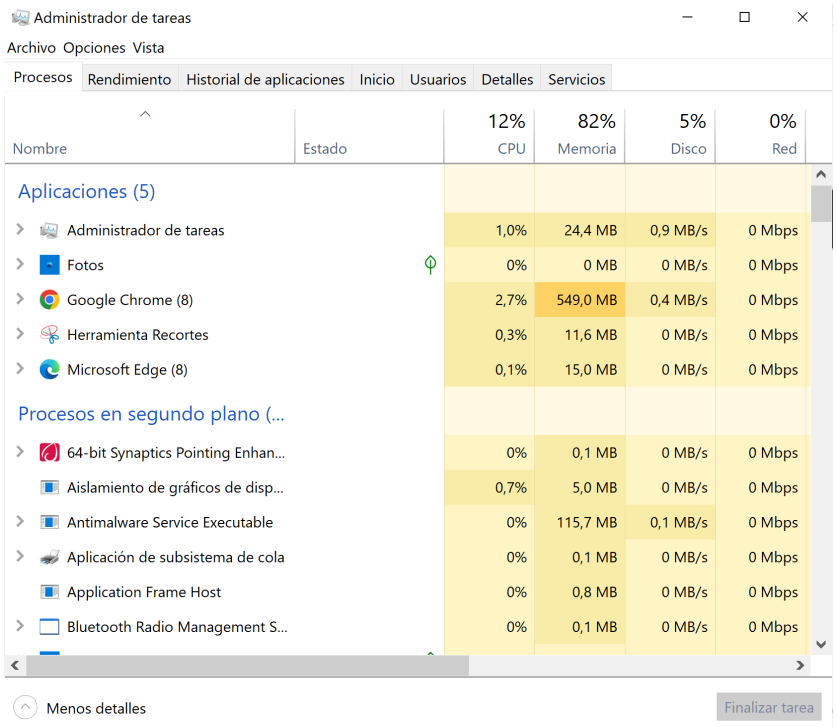
Estados de un proceso

¿Cómo averiguo el estado de un proceso?

Desde un terminal

```
$ htop
$ top
$ ps aux
```

Desde un administrador (win10)



¿Cómo creamos procesos?

Creación de procesos: ¿el huevo o la gallina?

- ¿Quién crea un proceso?... otro proceso
- ¿Quién crea ese otro proceso? otro proceso
- ¿Quién crea ese otro proceso? otro proceso. . .

Durante la inicialización del kernel se crea un proceso "raíz", con PID=1

- Linux: systemd, init, upstart
- macOS: launchd
- Windows: InitialSystemProcess, System

¿Cómo creamos procesos?

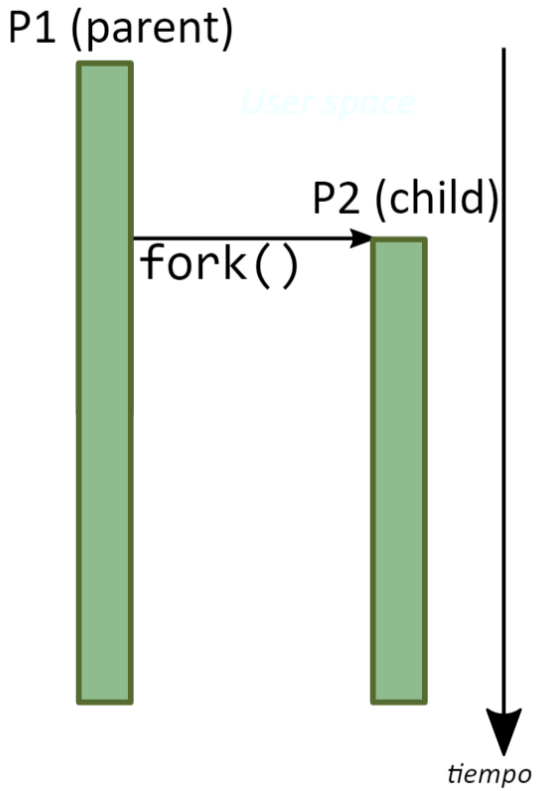
Todos los procesos tienen un proceso padre, o madre (parent).
La relación de creación permite formar un árbol de procesos

```
$ pstree --show-pids
```

```
systemd(1)─┬─crond(572)───rsync(984)
           ├─dbus-daemon(563)
           ├─konsole(30672)───zsh(30675)───tmux(11098)
           ├─systemd(756)─┬─ncmpcpp(759)
           │              └─emacs(773)─┬─aspell(798)
           │                           └─{gmain}(774)
           ├─systemd-logind(573)
           └─tmux(11100)─┬─zsh(9853)
                         └─zsh(11221)───vim(13615)
```

¿Cómo creamos procesos?

Un proceso (parent) crea a otro proceso hijo (child)



¿Cómo creamos procesos?

¿Quién sigue ejecutando?

- Parent e hijo continúan ejecutando concurrentemente.
- Ambos existen en la memoria del computador.
- Cuál proceso continúa en estado running depende de la implementación.

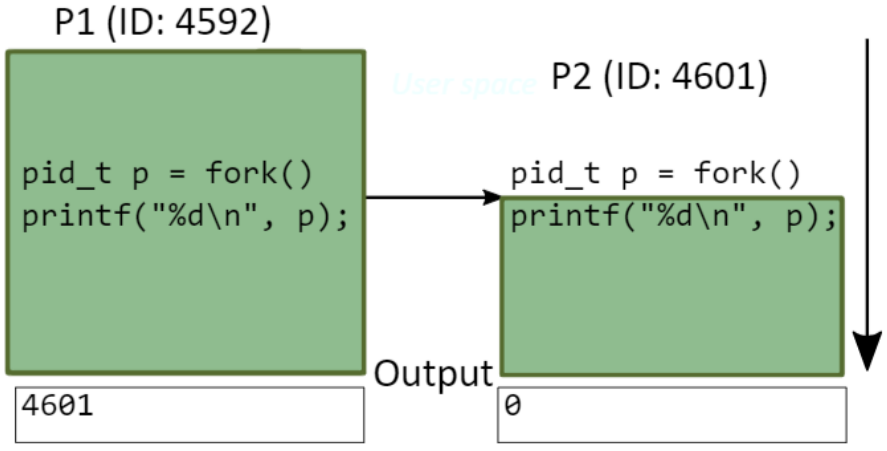
¿Qué "personalidad" tiene el hijo? (código, espacio de direcciones)

- El hijo es un duplicado casi exacto del padre
- El hijo es un nuevo proceso (otro espacio de direcciones)
- El hijo copia la memoria del padre a su nuevo espacio.
- El hijo continúa ejecutando en la misma instrucción que el padre: el retorno de fork()

¿Cómo creamos procesos?

Syscall Fork

- Crea un nuevo proceso como copia del padre.
- Ambos continúan ejecutando desde la instrucción de retorno de fork()
- fork() retorna PID del hijo al padre, y retorna 0 al hijo



¿Cómo creamos procesos?

Syscalls Fork Exec

Syscall `fork()`

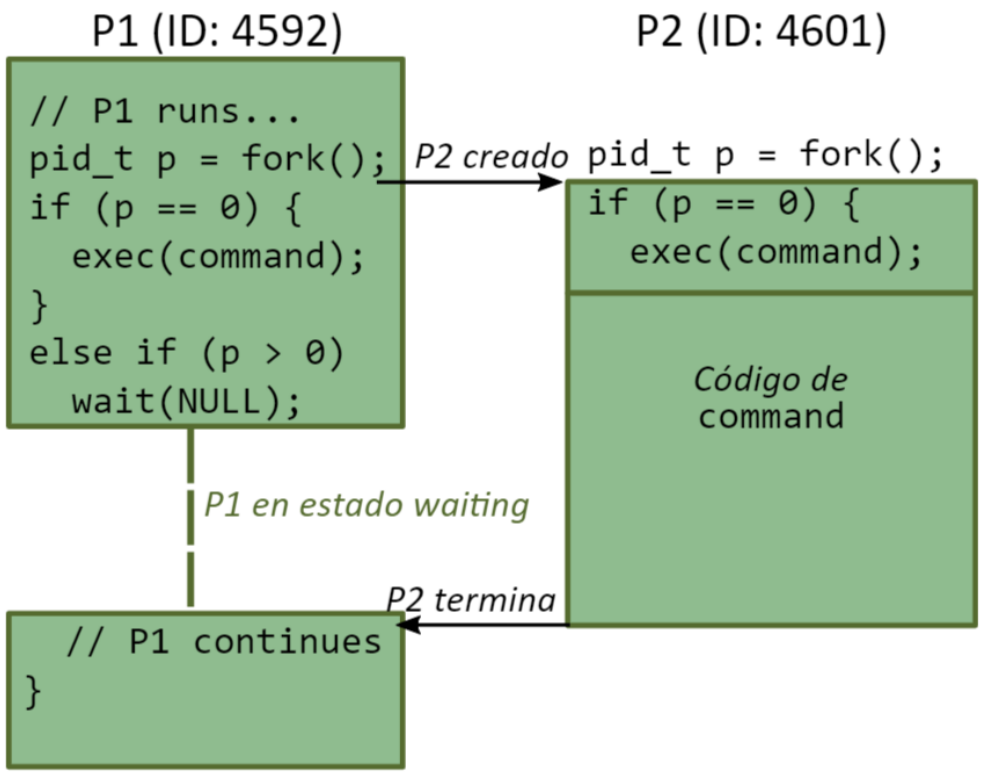
- Crea un nuevo proceso como **copia** del padre.
- Ambos continúan ejecutando desde la instrucción de retorno de `fork()`
- `fork()` retorna `PID` del hijo al padre, y retorna `0` al hijo

Syscall `exec()`

- Carga un binario en memoria **reemplazando** el código de quien lo llamó, e inicia su ejecución.
- El programa nuevo se "roba" el proceso (la memoria se sobreescribe)

Flujo normal de un Proceso

Esperando a los hijos



Flujo normal de un Proceso

Esperando a los hijos

Un flujo común de ejecución para lanzar un programa nuevo:

- Proceso hace fork(), para que el hijo haga exec()
- Proceso padre puede crear más hijos,...
- ... o esperar que el hijo termine antes de lanzar otro programa

Padre debe ser capaz de saber cuando algún hijo termina

- El padre conoce el pid del hijo (se lo da fork())

Flujo normal de un Proceso

Syscalls básicas

Syscall `fork()`

Clona el procesos (**copia** del padre)

Syscall `exec()`

Reemplaza el proceso por otro (sobreescribe)

Syscall `wait()`

Espera el término de un proceso hijo

Flujo normal de un Proceso

Ejemplo

Código simplificado de una *shell*:

```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
        type_prompt( );                         /* display prompt on the screen */
        read_command(command, parameters);      /* read input from terminal */

        if (fork( ) != 0) {                     /* fork off child process */
             /* Parent code. */
             waitpid(–1, &status, 0);           /* wait for child to exit */
        } else {
             /* Child code. */
             execve(command, parameters, 0);    /* execute command */
        }
}
```

Terminando Procesos

Todo proceso, al terminar su ejecución, ejecuta exit()

- Proceso entrega un código de retorno (¿para qué?)
- Sistema Operativo recupera todos los recursos asignados a ese proceso

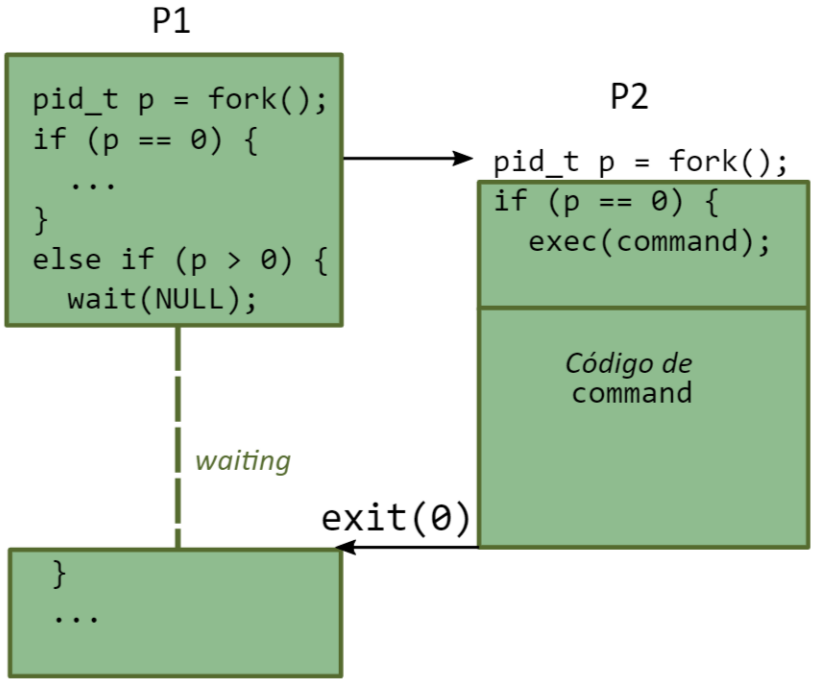
El código de retorno se utiliza para informar cómo resultó la ejecución

- También se le llama código de error.
- Convención: 0 indica que terminó de manera normal (sin problemas)
- Convención: distinto de 0 se usa para indicar un problema (error, o situación especial)
- La documentación del programa debería relacionar errores a códigos de retorno.

Terminando Procesos

Todo proceso, al terminar su ejecución, ejecuta exit()

La syscall wait() permite recibir un valor del hijo, desde donde se puede extraer el exit code.



Ahora tenemos la syscall exit () que termina el proceso con un código de retorno dado, y lo entrega al padre.

Señales, Huérfanos, y zombies

kill

Un proceso puede enviar señales (signals) a otros procesos **Syscall kill()**

- Envía una señal a otro proceso (por defecto SIGTERM)
- kill -l permite ver las señales disponibles
- SIGTERM indica al proceso que debe terminar
- SIGKILL elimina al proceso de la tabla de procesos (sin piedad)

Señales, Huérfanos, y zombies

Señales de linux

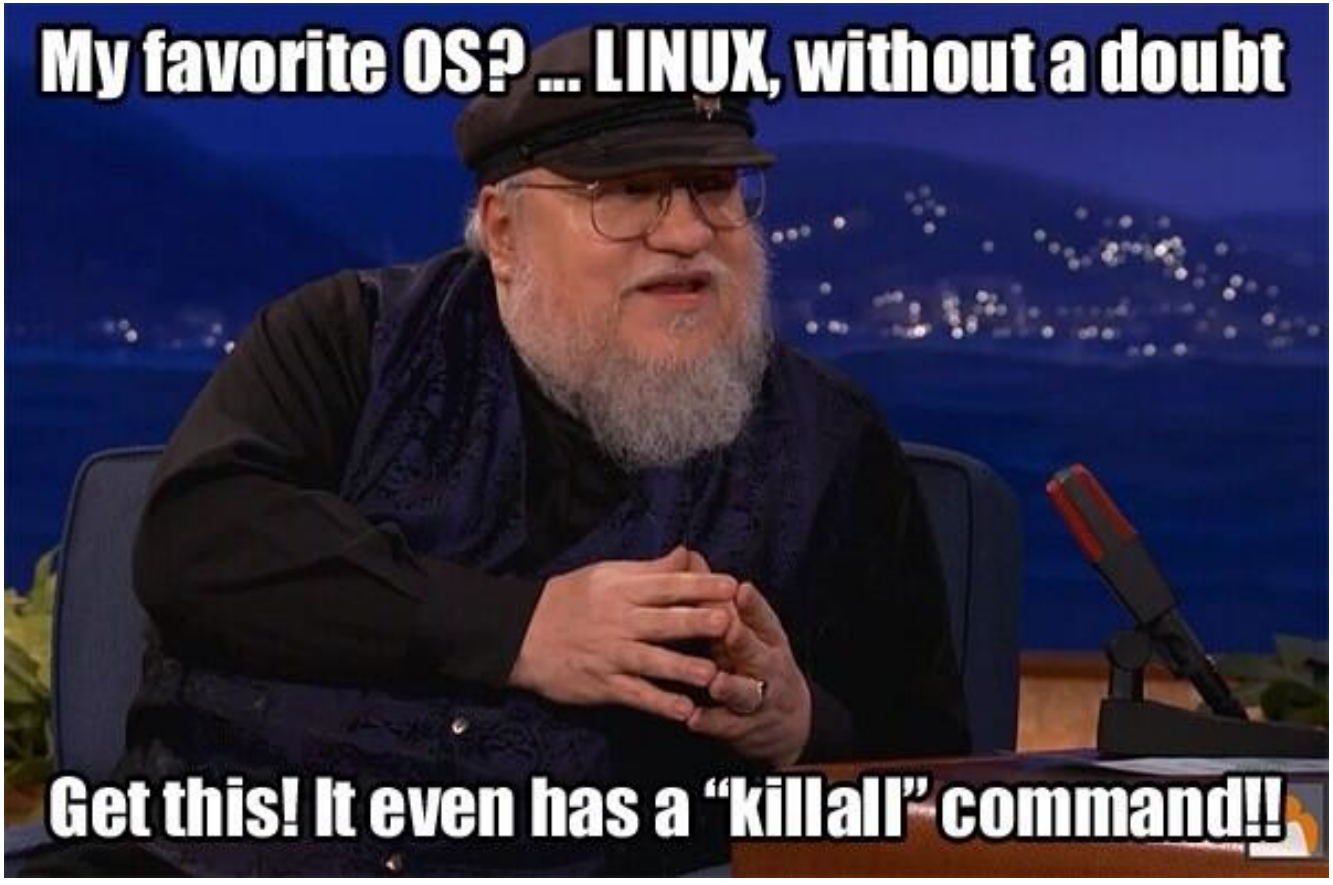
Linux Signals

Signal	Value	Description
1	SIGHUP	Hang up the process.
2	SIGINT	Interrupt the process.
3	SIGQUIT	Stop the process.
9	SIGKILL	Unconditionally terminate the process.
15	SIGTERM	Terminate the process if possible.
17	SIGSTOP	Unconditionally stop, but don't terminate the process.
18	SIGTSTP	Stop or pause the process, but don't terminate.
19	SIGCONT	Continue a stopped process.

Señales, Huérfanos, y zombies

killall

El comando killall permite enviar una signal a un grupo de procesos.



Señales, Huérfanos, y zombies

Si el padre muere, ¿deben morir los hijos?

- Sistema Operativo podría permitir (o no) que un hijo exista si el padre ha terminado.
- Linux permite que procesos hijos sigan viviendo, aún cuando el padre haya terminado.
- Sin embargo, podría parecer que los hijos sí mueren.

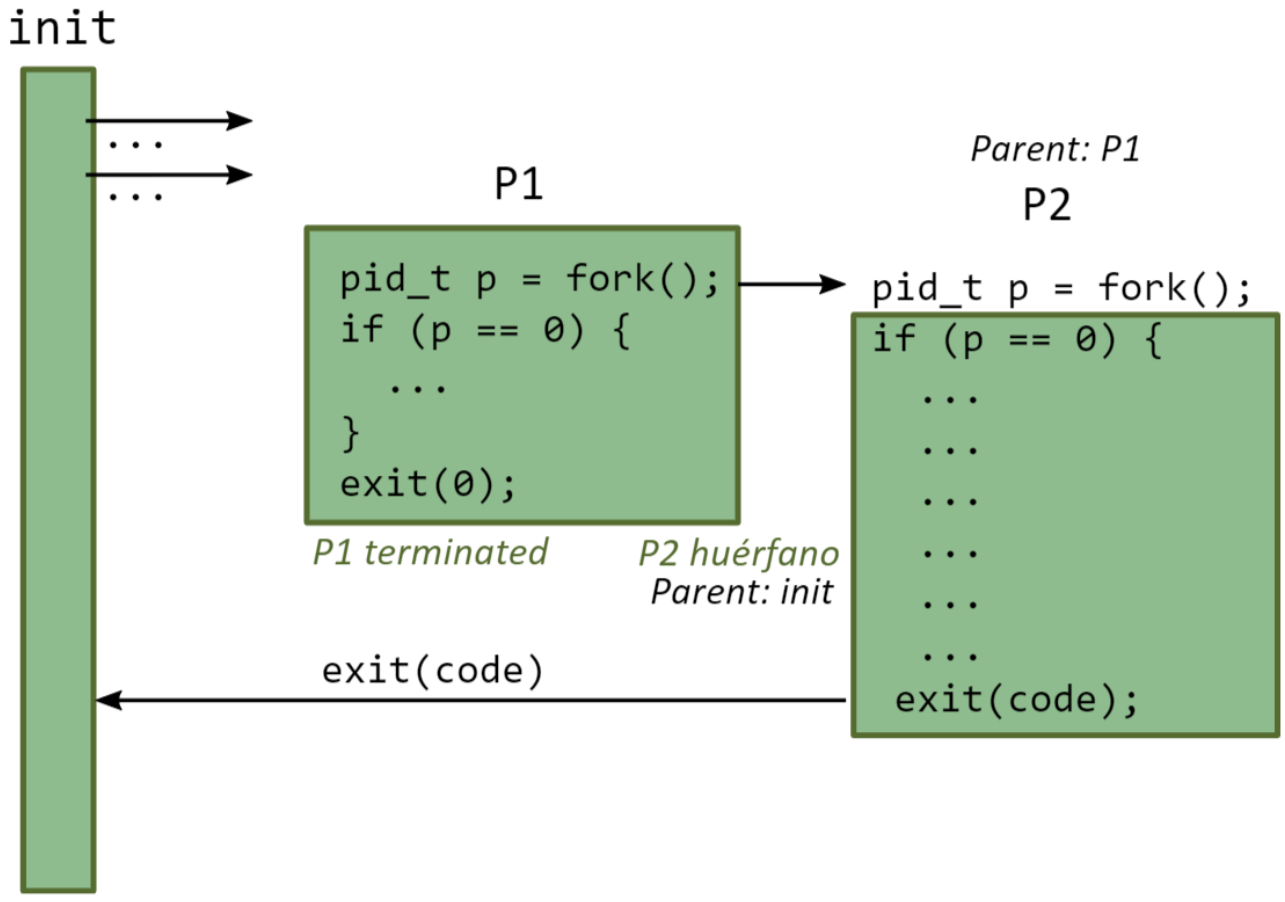
Señales, Huérfanos, y zombies

Procesos huérfanos

- Linux: Cuando un padre termina (exit) o muere (kill), sus hijos quedan huérfanos y pasan a ser hijos de init.
- Sin embargo, podría parecer que los hijos sí mueren (las condiciones pueden ser complicadas).
- init hace wait() periódicamente por sus hijos.

Señales, Huérfanos, y zombies

Procesos huérfanos



- P1 termina antes que P2
- P2 pasa a ser hijo de `init`
- `init` hace `wait()`

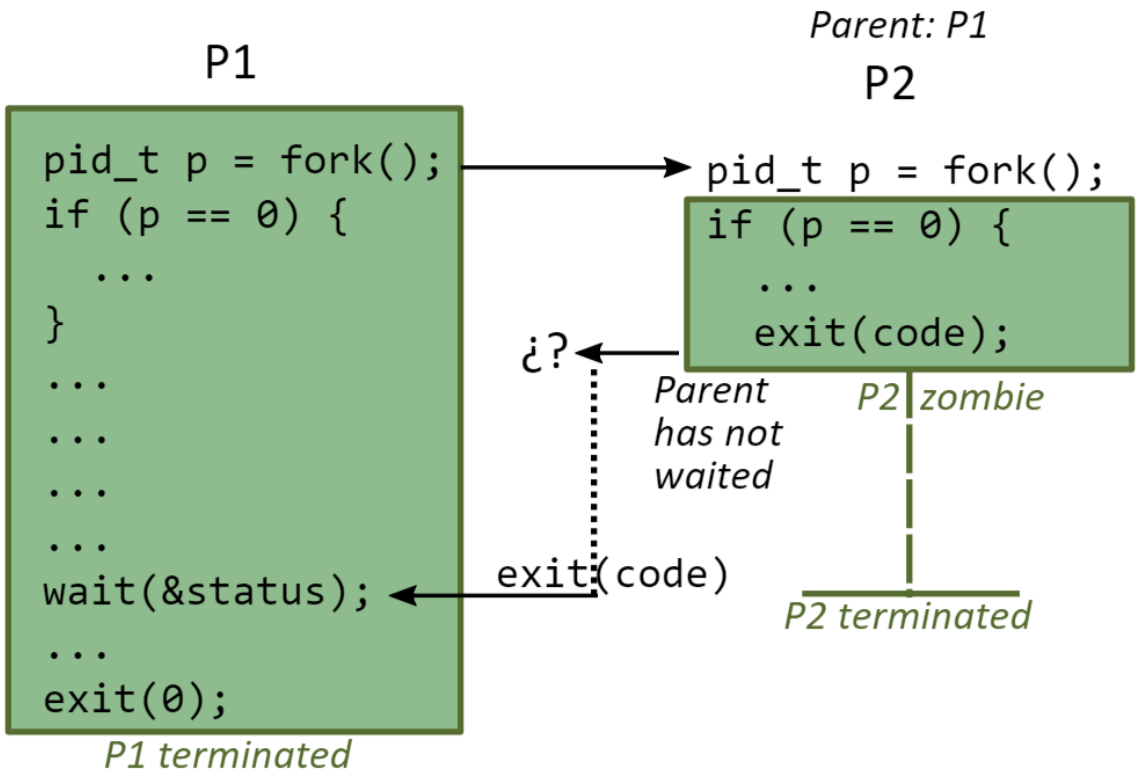
Señales, Huérfanos, y zombies

Procesos zombies

- Linux: Cuando un proceso termina y su padre no hace wait().
- Proceso terminado no se borra inmediatamente de la tabla de procesos (pero tampoco ejecuta).
- Proceso queda en estado zombie hasta que el padre hace wait().

Señales, Huérfanos, y zombies

zombies



- P2 termina antes que P1
- P1 no ha hecho `wait`
- No se puede eliminar el PCB de P2 hasta que P1 haga `wait`
- P2 queda zombie hasta que P1 hace `wait`

Material Adicional

- Vídeo Introducción procesos que son: Link Video
- Vídeo explicativo de la estructura de un proceso: Link Video
- Video explicativo Procesos y tiempo de ejecución: Link Video
- Video ejemplo FORK: Link Video
- Video ejemplos de código procesos huérfanos y zombies: Link Video

Próximo contenido a revisar será Scheduling