Operaciones sobre procesos

Adaptación de diferentes referencias bibliográficas

Juan Felipe Muñoz Fernández

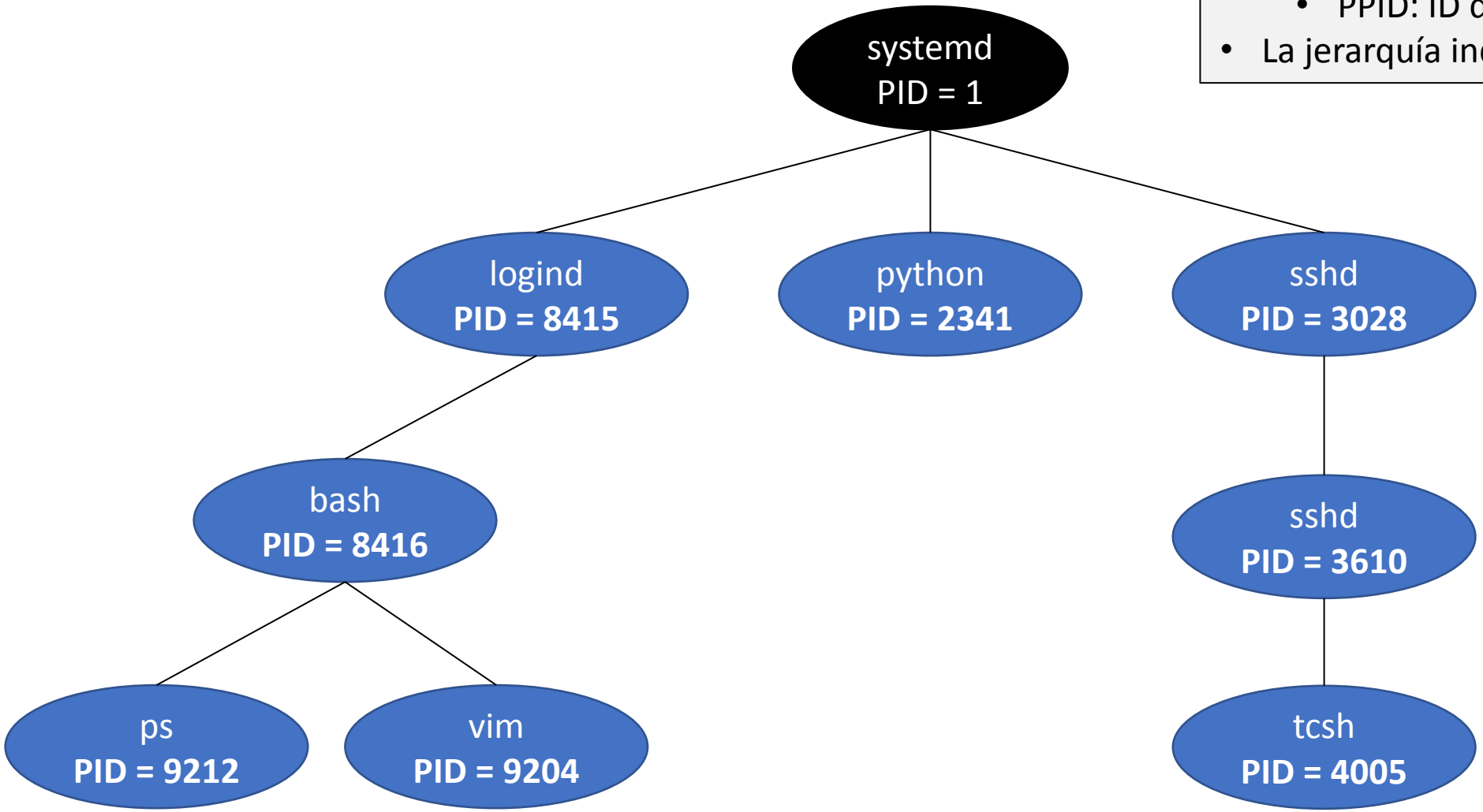
Preguntas

- ¿Cómo se crea un proceso?
- ¿Qué variaciones existen para crear procesos?
- ¿Qué sucede cuándo se crea un proceso?
- ¿Cómo se termina un proceso?

Creación de procesos

- Sabemos que un proceso es un programa en ejecución
- Un proceso puede crear nuevos procesos
  - Procesos del S.O crean procesos como servicios o demonios (Linux), shells, inicios de sesión, etc.
- Proceso padre: proceso que crea nuevos procesos
- Proceso(s) hijo(s): procesos creados por un proceso padre
- Se da relación padre – hijo entre procesos
  - Es importante esta relación en S.O como Linux en donde hay una jerarquía de procesos

- Cada proceso tiene identificador único PID
- En Linux existe una relación de jerarquía entre procesos
  - PPID: ID del proceso padre
- La jerarquía indica quién crea a quién



Creación de procesos

- Un proceso hijo puede obtener los recursos directamente del S.O.
- Un proceso hijo puede estar restringido a un subconjunto de recursos del proceso padre.
  - Evita que un proceso sobrecargue el sistema creando muchos proceso hijos.
- El proceso padre tiene que compartir sus recursos entre todos sus procesos hijos.
  - Memoria
  - Archivos abiertos

Cuando un proceso crea un nuevo proceso

- **Existen dos posibilidades de ejecución**
  - **Proceso padre** continua su ejecución concurrentemente con sus proceso hijos
  - **Proceso padre** espera hasta que todos o algunos de sus procesos hijos hayan terminado.
- **Existen dos posibilidades para el espacio de memoria del nuevo proceso**
  - **Proceso hijo** es un duplicado del proceso padre: misma sección de código y datos (**TEXT**, **DATA**)
  - **Proceso hijo** tiene nuevas secciones de código y datos: nuevo programa en ejecución.

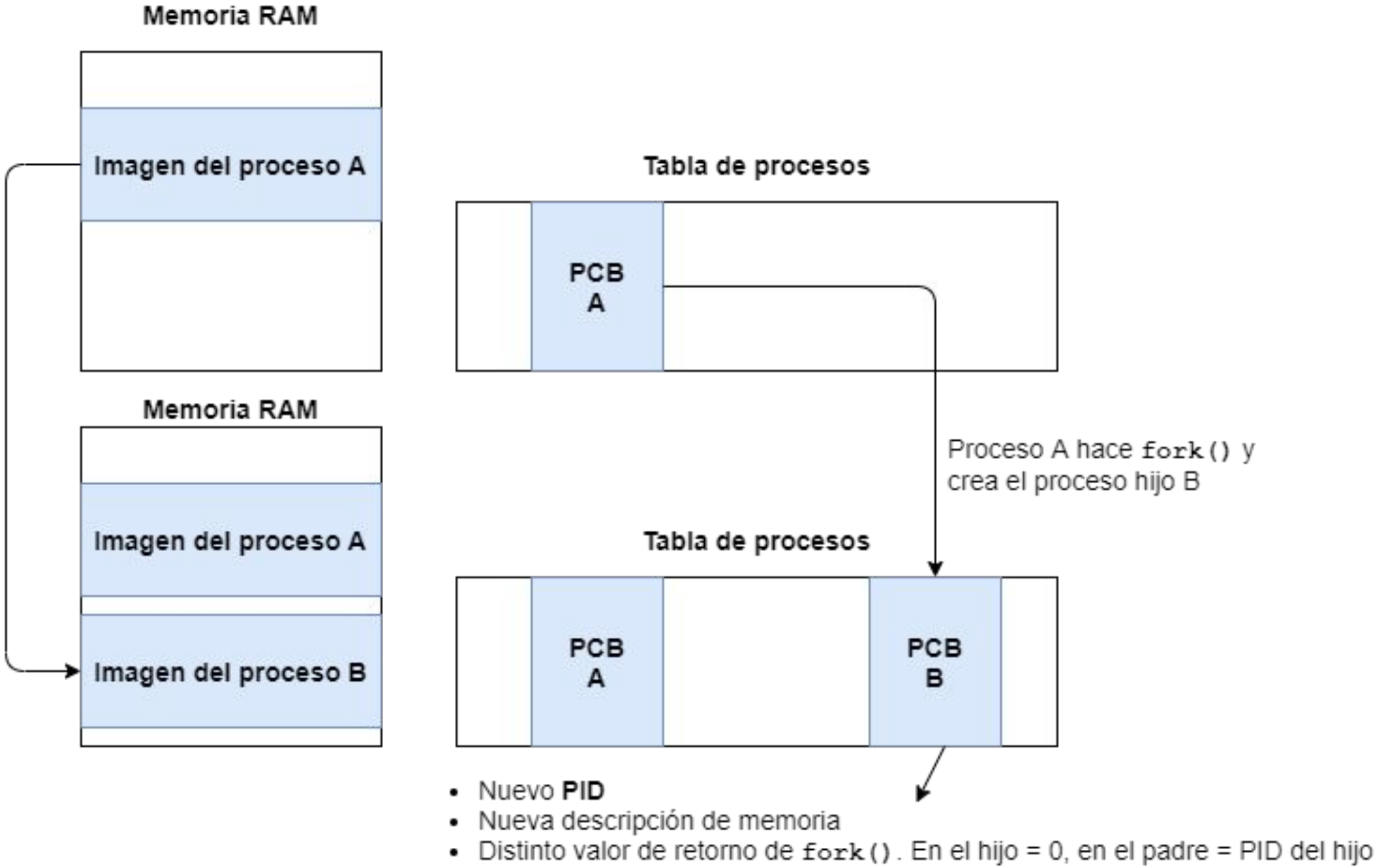
p01-fork.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    printf("Proceso padre (pid:%d)\n", (int) getpid());
    int rc = fork(); // Creación proceso hijo
    if (rc < 0) {
        // Falla creación proceso hijo
        printf("Falló fork()\n");
        exit(1);
    } else if (rc == 0) {
        // Proceso hijo: nuevo proceso
        printf("Proceso hijo (pid:%d)\n", (int) getpid());
    } else {
        // Proceso padre sigue por aquí
        printf("Proceso padre de (pid:%d)\n", rc);
    }
    return 0;
}
```

`fork()` y `getpid()` son ***system calls*** del sistema operativo Linux

¿Qué implica que compartan el mismo mapa de memoria?

- El proceso hijo **NO** inicia su ejecución en `main()`
- Proceso padre y proceso hijo **comparten el mismo mapa de memoria al momento del `fork()`.**
- Son dos procesos: dos PID diferentes, dos mapas de memoria, dos PCB
- **Resultado de ejecución NO determinístico**



- Nuevo **PID**
- Nueva descripción de memoria
- Distinto valor de retorno de `fork()`. En el hijo = 0, en el padre = PID del hijo

En el diagrama anterior…

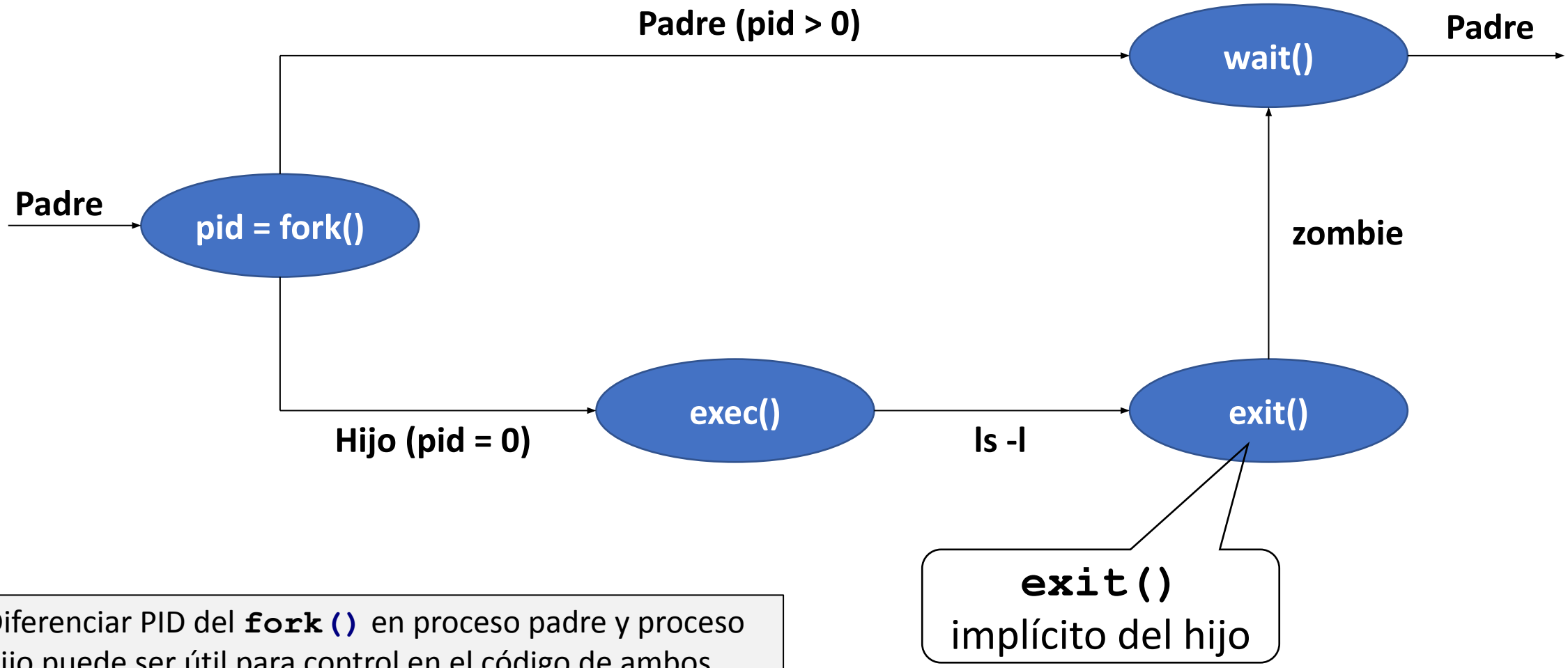
- Datos y pila de **B** son iguales a los de **A** en instante de `fork()`.
- Registro **IP** tiene mismo valor para **A** y **B** en `fork()`.
- Proceso **B** su propio PID.
- Proceso **B** no está en la misma zona de memoria de **A**.
- **B** con copia de descriptores de **A**
  - Archivos abiertos por **A** se ven en **B**
- **A** y **B** comparten punteros de posición en archivos
  - Porque comparten descriptores de archivos abiertos
  - PCB de **B** es copia de PCB de **A** con algunas variaciones: p. ej.: el PID, mapa de memoria,
- Modificaciones de datos (memoria) en **A** no interfieren con **B** y viceversa.

p02-fork-execlp.c

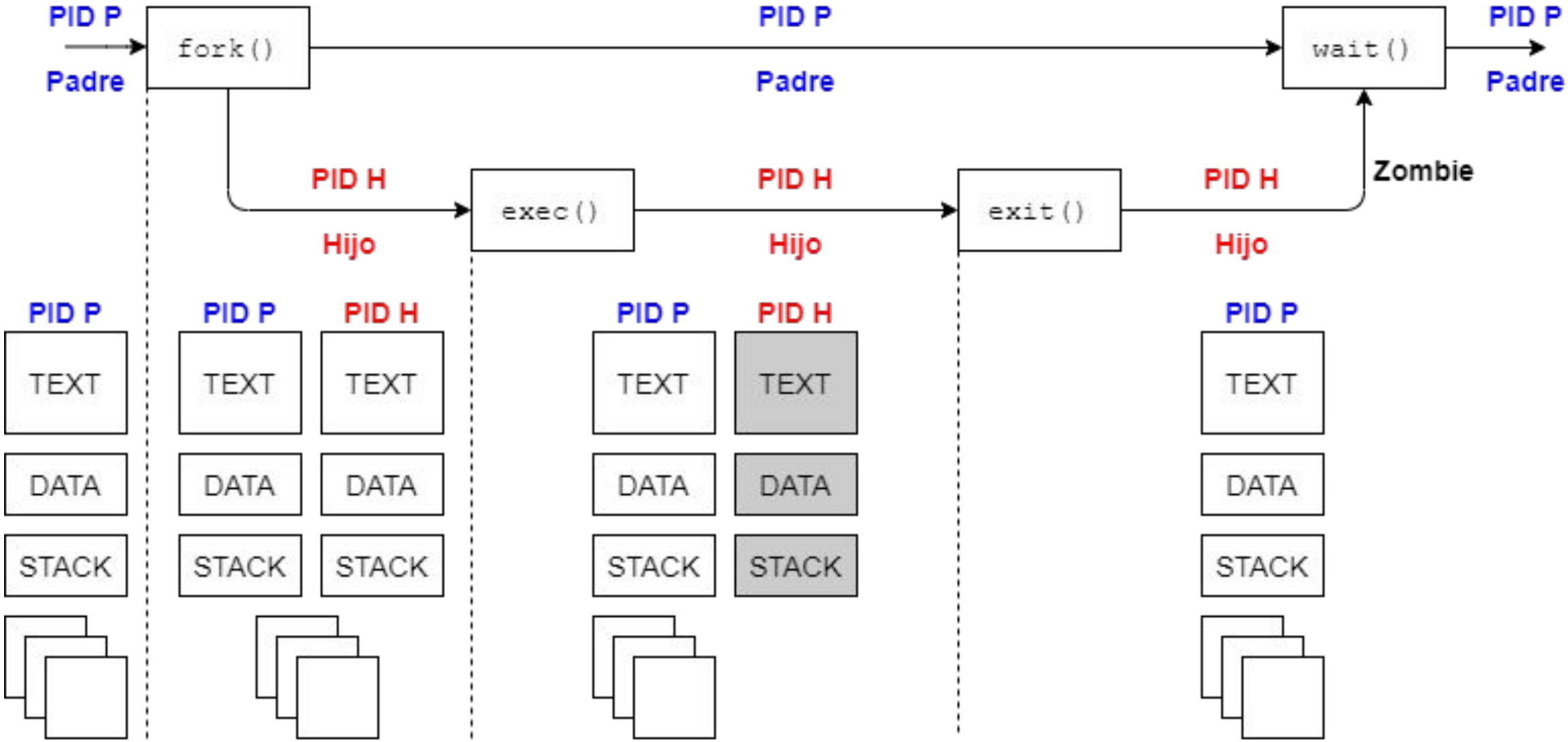
```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main(){
    pid_t pid; /* Se crea nuevo proceso */
    pid = fork();
    if (pid < 0) { /* error en fork() */
        printf("Falló fork()");
        return 1;
    }
    else if (pid == 0) { /* Proceso hijo */
        execlp("/bin/ls","ls","-l",NULL);
    }
    else { /* Proceso padre */
        wait(NULL); /* Espera a que hijo termine */
        printf("Hijo termina\n");
    }
    return 0;
}
```

- Proceso hijo con nuevo mapa de memoria con llamada a `execlp()`.
- Hijo es una copia del proceso padre al momento el `fork()`
- Son dos procesos, dos PID diferentes.
- `wait()` hace que resultado de ejecución sea determinístico

Creación proceso hijo y `execlp()`



Diferenciar PID del `fork()` en proceso padre y proceso hijo puede ser útil para control en el código de ambos casos.


PID P
Padre
fork()
PID P
Padre
wait()
PID P
Padre
PID H
Hijo
exec()
PID H
Hijo
exit()
PID H
Hijo
Zombie
PID P
TEXT
DATA
STACK
PID P
PID H
TEXT
DATA
STACK
PID P
PID H
TEXT
DATA
STACK
PID P
TEXT
DATA
STACK

El file system `/proc` en Linux

- El *file system* **`/proc`** es una interfaz a estructuras de datos en el kernel
- Es un *file system* en memoria RAM pero se mapea como un directorio del sistema de archivos.
- Permite ver información de los procesos en ejecución
  - P. Ej.: información del PCB del proceso
- Permite modificar en tiempo de ejecución ciertos parámetros del kernel
- Ver el **mapa de memoria** en **`/proc`**
  - `cat /proc/<pid>/maps`
- Ver el **mapa de memoria** con GDB
  - `gdb -p <pid>`
  - Orden en GDB: `info proc mappings <pid>`

Sobre `exec()`

- Sobre escribe mapa de memoria de proceso que llama con imagen de memoria de ejecutable indicado.
- Segmentos de ***stack*** y ***heap*** se reinician.
- Transforma el proceso que llama en el nuevo proceso llamado.
  - P. Ej.: `p02-fork-execlp  ls -l`

¿Por qué `fork()` + `exec()`?

- Piense en el comportamiento de la shell de Linux
- ¿Cómo se imagina la shell (por dentro) desde este punto de vista?

Creación de procesos en Windows

- Se usa el API **`CreateProcess()`**
  - https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa
- Proceso hijo **NO** hereda espacio de direccionamiento de proceso padre.
- **`CreateProcess()`** exige que se pase el nombre de un ejecutable para cargarlo en el espacio de memoria del proceso hijo.
- A diferencia de **`fork()`**, **`CreateProcess()`** espera no menos de 10 parámetros.

```c
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

void _tmain(int argc, TCHAR* argv[]) {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // Programa a ejecutar
    TCHAR myProgram[] = L"C:\\Windows\\system32\\calc.exe";
    // Inicia proceso hijo
    if (!CreateProcess(NULL, // Usar la línea de comandos
        myProgram,      // Ejecutable
        NULL,           // Manejador del proceso: no heredable
        NULL,           // Manejador del hilo: no heredable
        FALSE,          // No herencia
        0,              // Sin flags
        NULL,           // Usar bloque del entorno del padre
        NULL,           // Use el directorio de inicio del padre
        &si,            // Apuntador a estructura STARTUPINFO
        &pi))           // Apuntador a estructura PROCESS_INFORMATION
    {
        printf("Falló CreateProcess() (%d).\n", GetLastError());
        return;
    }
    // Esperar hasta que proceso hijo termine.
    WaitForSingleObject(pi.hProcess, INFINITE);
    // Terminar proceso padre y cerrar manejadores.
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Terminación de un proceso

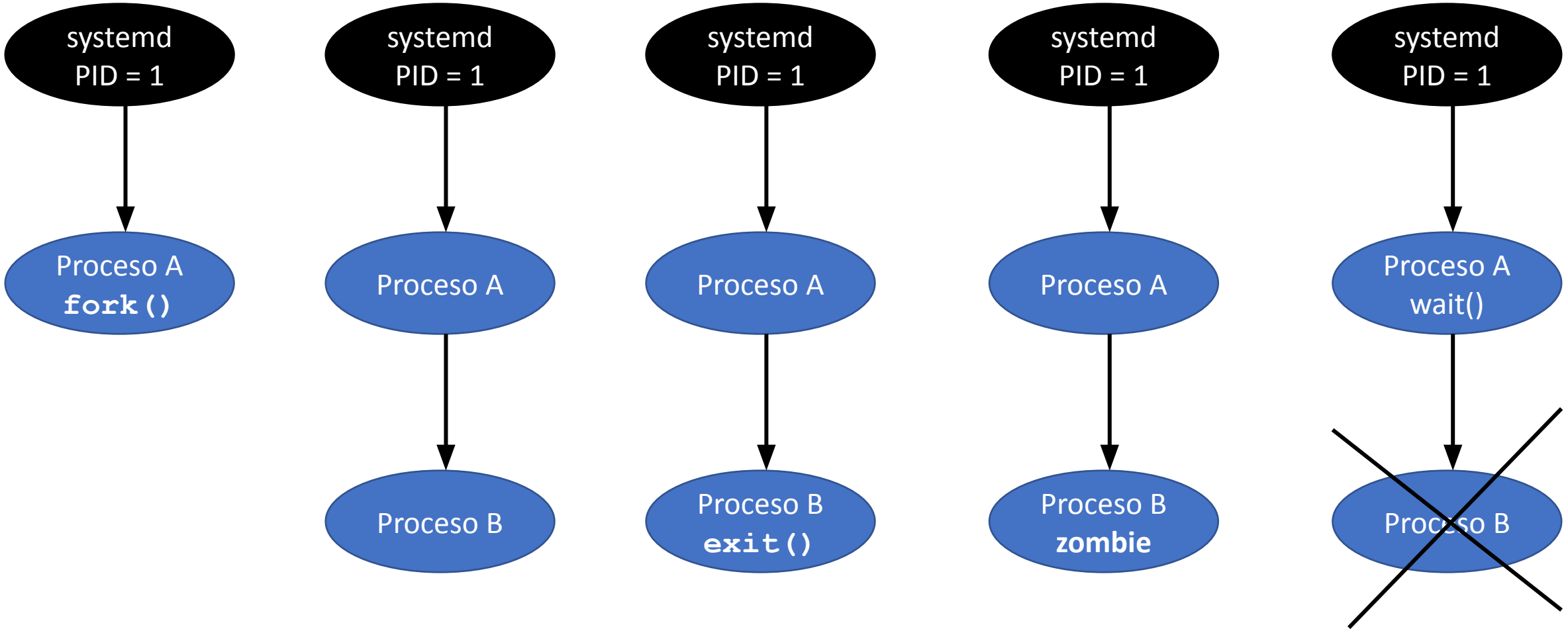
- En Linux usualmente se llama a **`exit()`**.
  - Se retorna el estado del proceso cuando se pasa como parámetro a **`exit()`**.
  - `exit(0)`, `exit(1)`, etc.
- Se pueden terminar procesos en otras circunstancias
  - Proceso excede tiempo y recursos
  - No se requiere más el proceso hijo
  - El padre termina y el S.O no permite a los hijos existir sin el padre.
- En el caso Windows **`TerminateProcess()`** es una llamada que ejecuta el padre para terminar procesos hijos.
  - Se requiere del proceso hijo: **`PROCESS_INFORMATION.hProcess`**

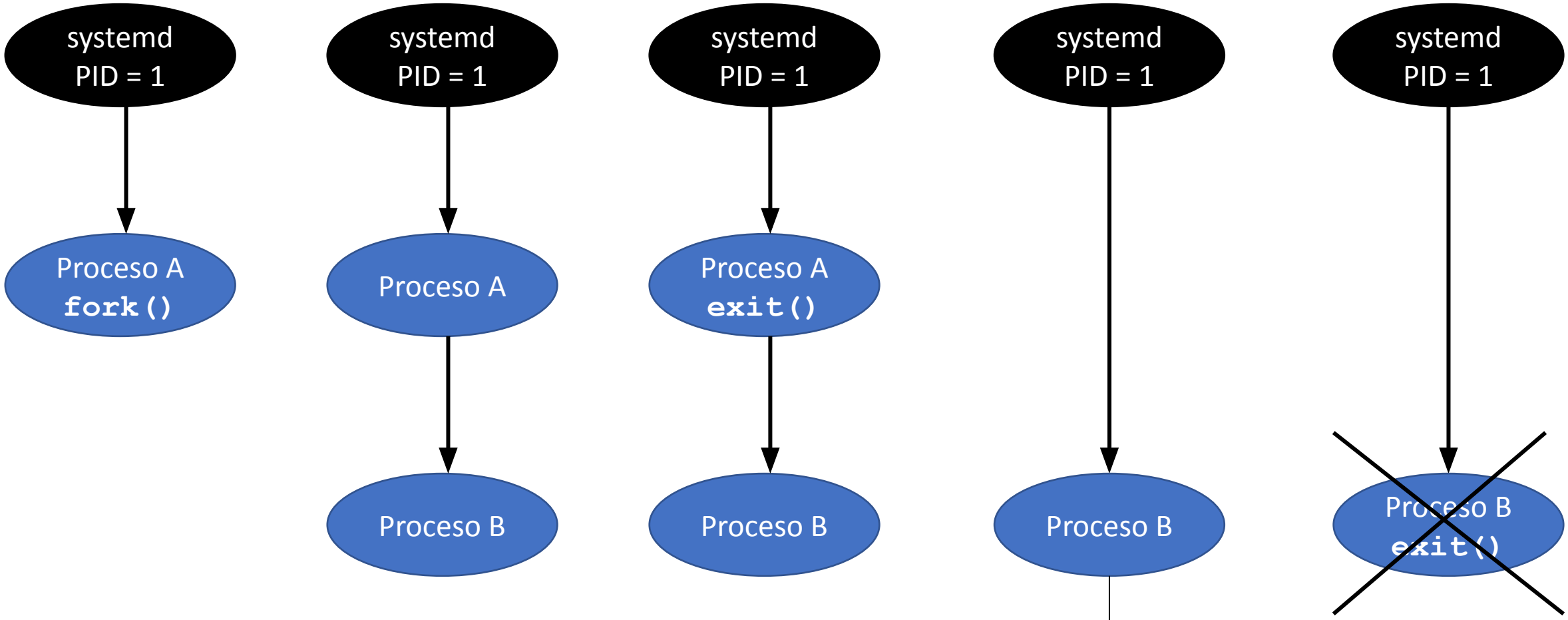
Terminación de un proceso

- En algunos sistemas no permiten a proceso hijo existir sin el padre.
- Cuando proceso hijo termina
  - S.O obtiene recursos asignados
  - Se mantiene información del proceso hijo en PCB hasta que padre llama a **`wait().`**
- En Linux
  - Proceso hijo terminado pero padre no ha llamado a **`wait()`** se denomina proceso **zombie**.
  - Todos los procesos transitan muy brevemente este estado.
  - Cuando se llama a **`wait()`** se recupera PID de proceso hijo terminado y proceso sale del estado **zombie**.

Uso de `wait()` en Linux

- Esperan por un cambio de estado en el proceso hijo
- Un cambio de estado en el proceso hijo se considera
  - Proceso hijo terminado: de manera normal o anormal
  - Proceso hijo detenido por una señal
  - Proceso reanudado por una señal.
- En caso de **proceso hijo terminado** y se llama a **`wait()`** implica
  - Liberar recursos asignados al proceso hijo
  - Eliminar información del PCB
- En caso de **proceso hijo terminado** y no llamar a **`wait()`** implica
  - Proceso hijo a estado zombie: sin recursos pero aún con el PCB.
  - Proceso hijo reasignado a **PID = 1**, llama periódicamente a **`wait()`**


systemd
PID = 1
Proceso A
fork()
systemd
PID = 1
Proceso A
Proceso B
systemd
PID = 1
Proceso A
Proceso B
exit()
systemd
PID = 1
Proceso A
Proceso B
zombie
systemd
PID = 1
Proceso A
wait()
Proceso B



- BCP de **B** con información obsoleta del PID del padre
- Proceso **B** queda huérfano
- Procesos huérfanos en **B** pasan a **systemd**
- **systemd** está en un bucle infinito de **`wait()`**

Señales y excepciones

- Se usan para notificar a procesos
- En Linux: señales
- En Windows: excepciones
- Es una interrupción al proceso
  - Se detiene la ejecución en la instrucción donde se recibe la señal.
  - Se bifurca a ejecutar código de tratamiento de la señal. **No siempre**.
  - Continua ejecución en instrucción en donde fue interrumpido. **No siempre**.
- La señal la puede enviar un proceso
  - Proceso padre a sus hijos pero no a otros que no sean sus hijos.

Señales y excepciones

- La señal la puede enviar el sistema operativo
  - Desbordamiento en operaciones aritméticas
  - División por cero
  - Ejecutar instrucción no válida: código de operación incorrecto
  - Direccionar una posición de memoria prohibida
- Tipos de señales
  - Excepciones de hardware
  - Comunicación
  - E/S asíncrona

Armado de señales

- Se debe especificar al S.O cuál es el código que trata con la señal recibida.
  - Armar la señal.
  - Indica el nombre de la señal.
  - Indicar rutina que atiende señal.
- Algunas señales se pueden ignorar por un proceso.
  - Algunas no.
- Algunas señales se pueden enmascara por un proceso.
  - El S.O las bloquea hasta que el proceso las desenmascara.
- Si señal no está armada o enmascarada usualmente se mata al proceso que la recibe.

Excepciones

- Caso típico en programación

```
try
{
   Código que podría producir una excepción
}
catch/except()
{
   Código para el tratamiento de la excepción
}
```

Referencias

- Carretero Pérez, J., García Carballeira, F., de Miguel Anasagasti, P., & Pérez Costoya, F. (2001). Procesos. In *Sistemas operativos. Una Visión Aplicada* (pp. 77–160). McGraw Hill.
- Silberschatz, A., Baer Galvin, P., & Gagne, G. (2018). Process Management. In *Operating Systems Concepts* (10th ed., pp. 105–115). John Wiley & Sons, Inc.