Procesos

Adaptación

Juan Felipe Muñoz Fernández

Preguntas

- ¿Qué es un proceso?
- ¿Por qué es importante el concepto de proceso en los sistemas operativos?

- Unidad de trabajo del S.O
- Cuando un programa es ejecutado se vuelve proceso.
  - Usuario da la orden
  - S.O lo carga

→ [Se carga en memoria

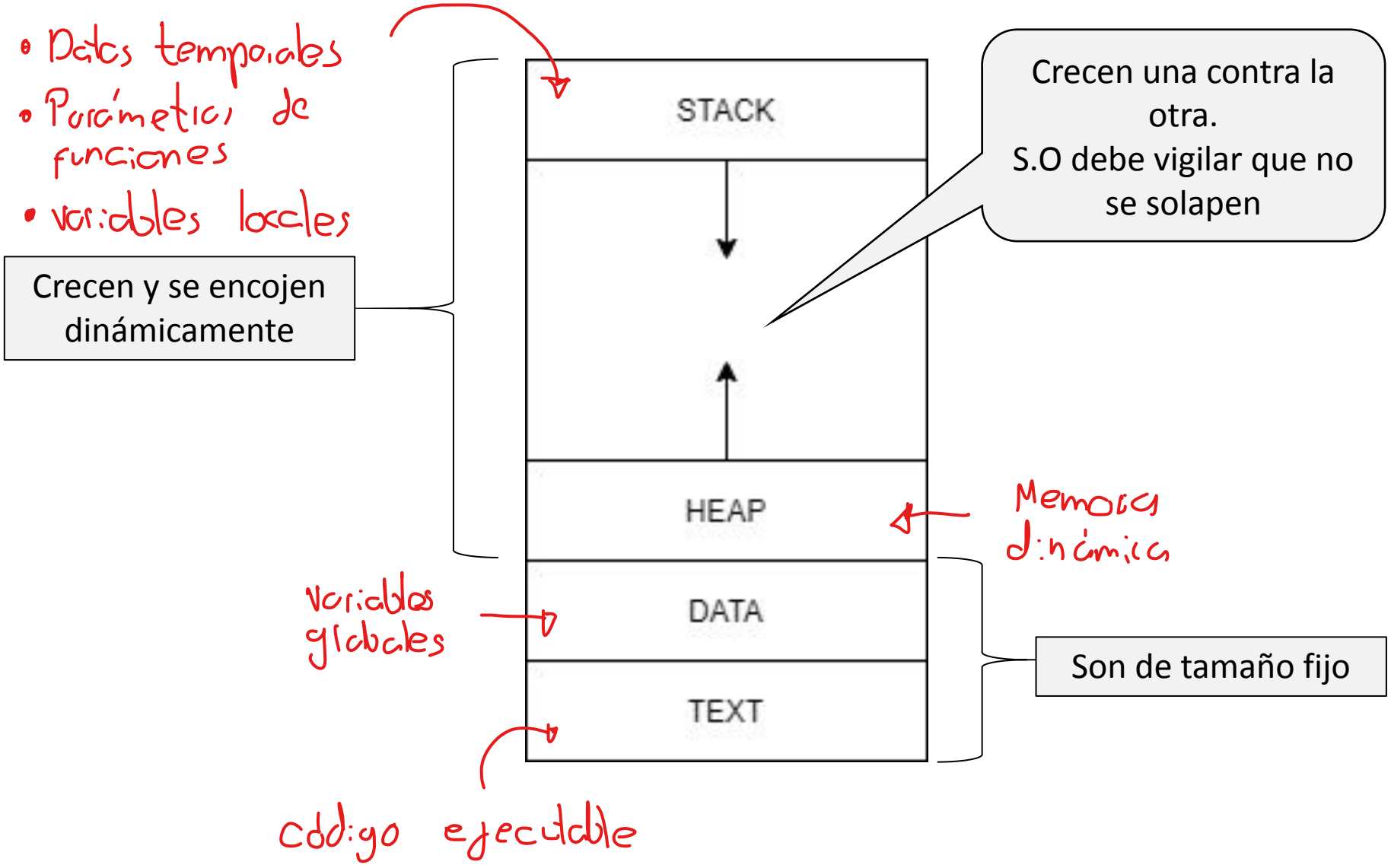
Definición

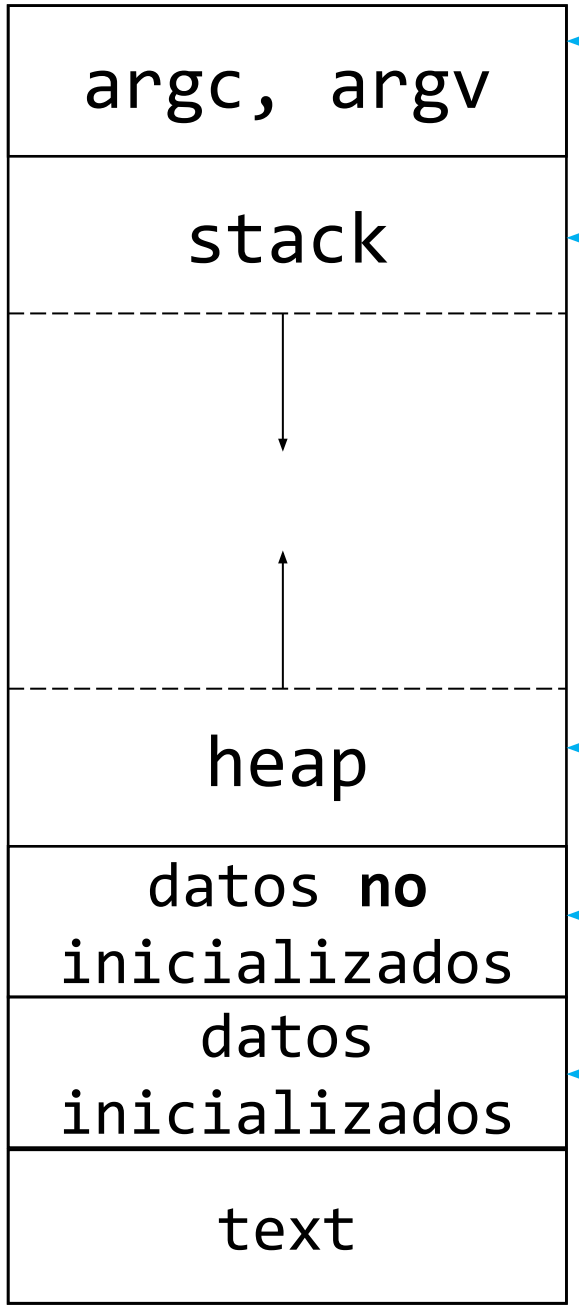
- Programa en ejecución
  - Programa en disco es una entidad estática
  - Programa cargado en memoria es una entidad dinámica
- Es la unidad de trabajo del sistema operativo
  - El S.O también tiene sus propios procesos
- Un sistema computacional moderno consiste de una colección de procesos
  - Ejecutando código de usuario (aplicaciones de usuario: ring 3)
  - Ejecutando código del núcleo del S.O (procesos del S.O: ring 0)
- Todos los procesos podrían ejecutarse concurrentemente en CPU
  - Tiempo compartido (multiplexión en el tiempo)

Definición

- Un programa se vuelve proceso cuando:
  - El archivo ejecutable es cargado en memoria
  - Todos los programas deben estar cargados en memoria para su ejecución
- ¿Cómo se carga un archivo ejecutable en memoria?
  - Usuario da la orden: doble clic, enter, orden en línea de comandos
  - El S.O lo carga automáticamente: servicios, demonios, procesos propios
- Mapa de memoria de un proceso
  - Sección ***text***: código ejecutable
  - Sección ***data***: variables globales
  - Sección ***heap***: memoria dinámica (p. ej.: listas)
  - Sección ***stack***: datos temporales, parámetros funciones, variables locales

Mapa de memoria de un proceso





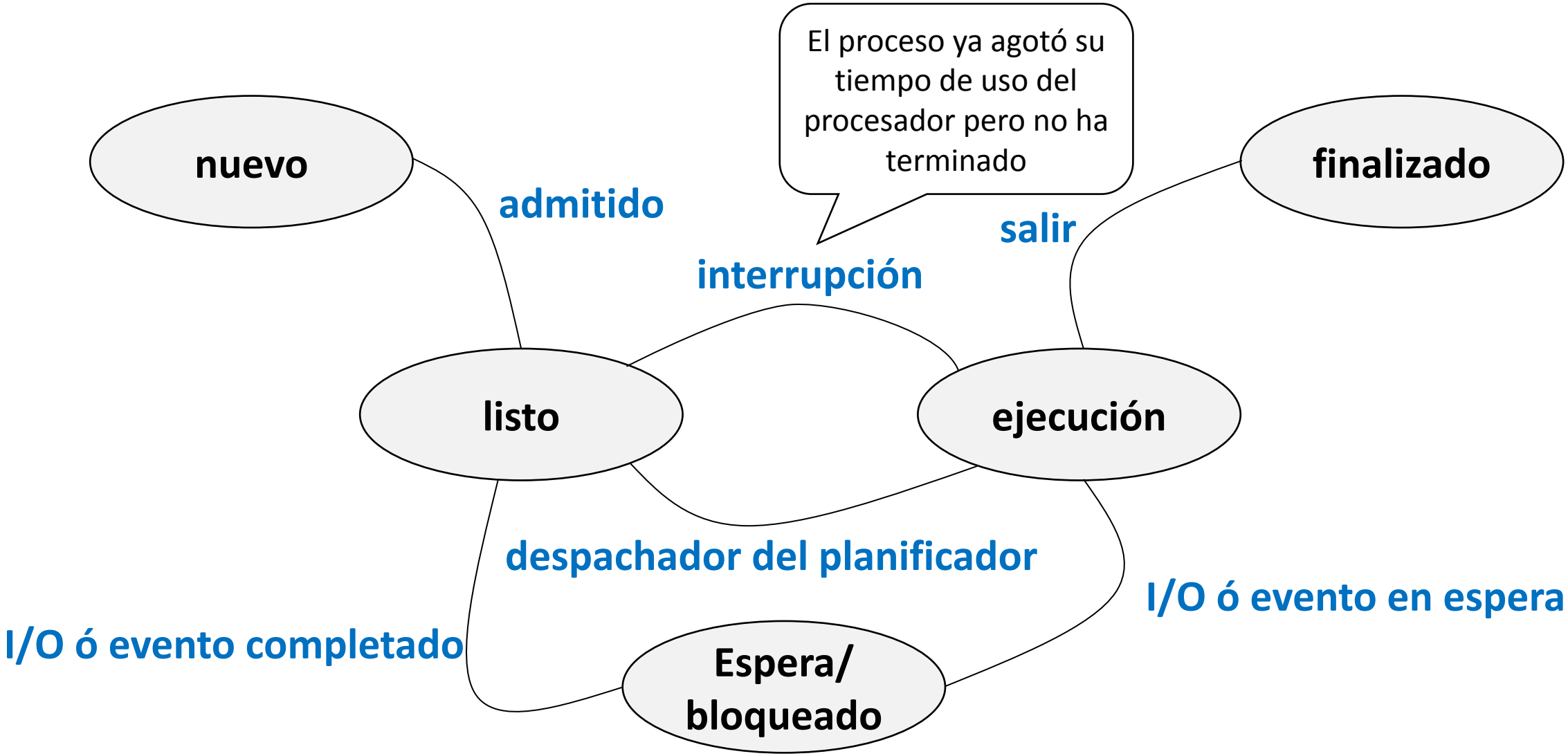

```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for (i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Formación de un proceso

- Completar toda la información que lo constituye
  - Asignar espacio de memoria (virtual) constituido por varios segmentos
  - Seleccionar un PCB libre en tabla de procesos
  - Rellenar el BCP con la información del proceso
  - Cargar en el segmento de texto el código + rutinas del sistema
  - Cargar en el segmento de datos los datos inicializados
  - Crear el segmento de pila con los parámetros que se pasan al programa

Estados y transiciones

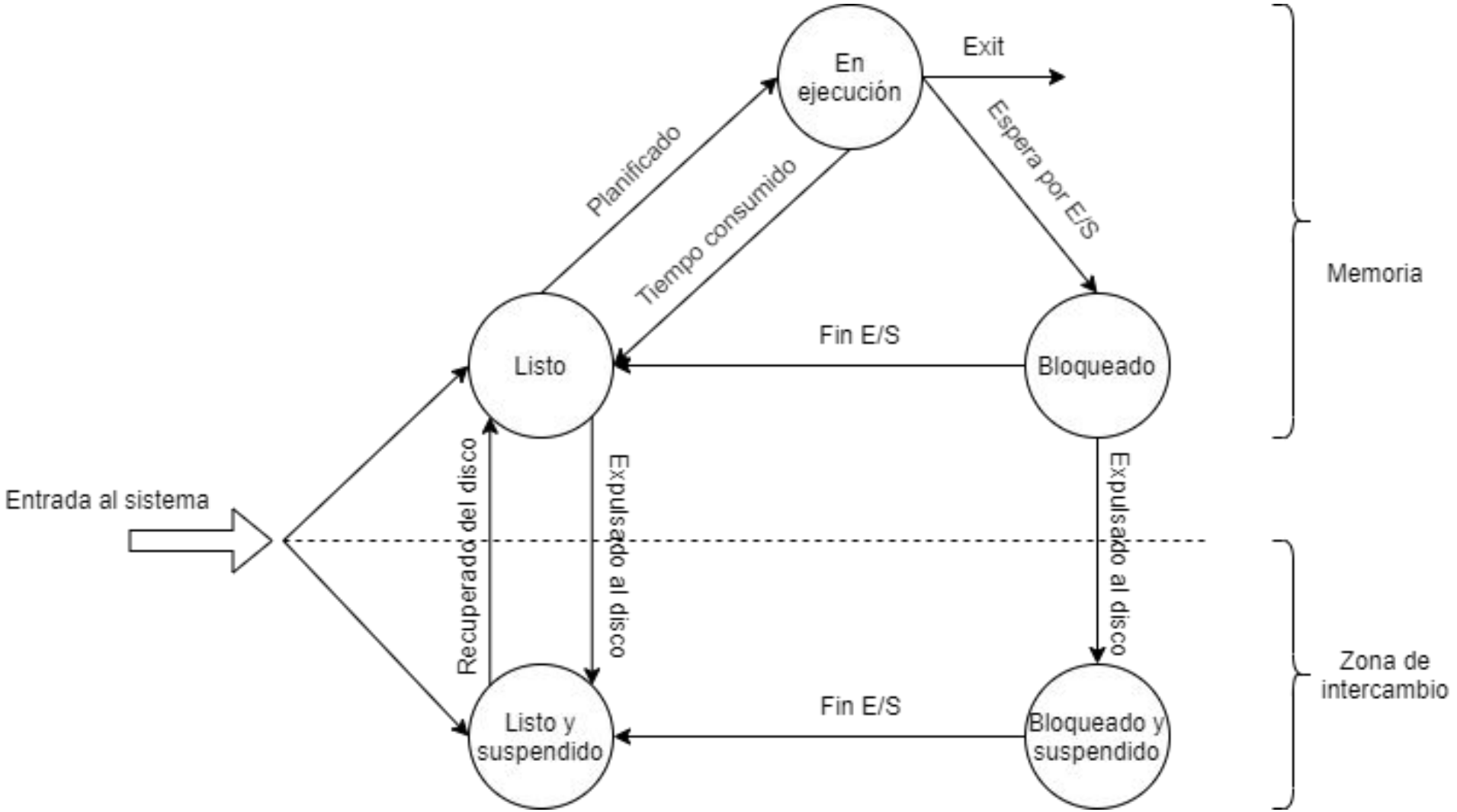
Solo un proceso puede estar en ejecución en cualquier núcleo de procesador a la vez.



Estados y transiciones

- Nuevo
  - Proceso creado (cargado en memoria)
- Ejecución
  - Instrucciones siendo ejecutadas
- Espera/bloqueado
  - El proceso está a la espera de que ocurra algo
  - Que se complete una solicitud de I/O
  - Que se reciba una señal
- Listo
  - El proceso está a la espera de que se le asigne tiempo de procesador
- Finalizado
  - El proceso completó su ejecución

Estados y transiciones



- Estado de suspensión: retira los marcos de página del proceso y los envía al área de ***swapping***.
- Liberar memoria para los procesos no suspendidos

Bloque de control de proceso (BCP/PCB)

Es una estructura de datos

- Se requiere una estructura de control y representación de la información de un proceso
  - Control de ejecución
  - Tiempos asignados
  - Recursos usados
  - Tiempos de espera

"Registra la vida de un proceso en el sistema"

- El despachador del planificador requiere cierta información del proceso
  - Asignar nuevamente tiempo de procesador
  - Hacer transiciones hacia otros estados

Bloque de control de proceso

- Registra la «vida» de un proceso en el sistema

Estado del proceso
Número de proceso (PID)
Contador de programa (IP: Instruction pointer)
Registros de CPU
Límites de memoria
Archivos abiertos
...

Bloque de control de proceso

- Estado del proceso
  - Nuevo, listo, en ejecución, en espera, terminado, etc.
- Contador del programa
  - Dirección de la siguiente instrucción a ejecutarse
- Registros de CPU
  - Todos los registros que se requieran para interrumpir y ejecutar nuevamente el proceso.
- Información del planificador
  - Prioridad, apuntadores a las colas de planificación, entre otros

Bloque de control de proceso

- Información de administración de memoria
  - Valores de límites de memoria, registros base, tablas de páginas, etc.
- Información de contabilidad
  - Cantidad de tiempo en CPU, límites de CPU, etc.
- Información de estado de I/O
  - Dispositivos de I/O asignados al proceso
  - Archivos abiertos
- Otra información
  - El PCB es dependiente de las consideraciones de diseño de cada S.O
  - No es una estructura estándar.

PCB en xv6 [1/2]

```
// Registros del proceso
struct context {
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};
```

```
// Estado del proceso
enum proc_state {
    UNUSED,
    EMBRYO,
    SLEEPING,
    RUNNABLE,
    RUNNING,
    ZOMBIE
};
```

PCB en xv6 [2/2]

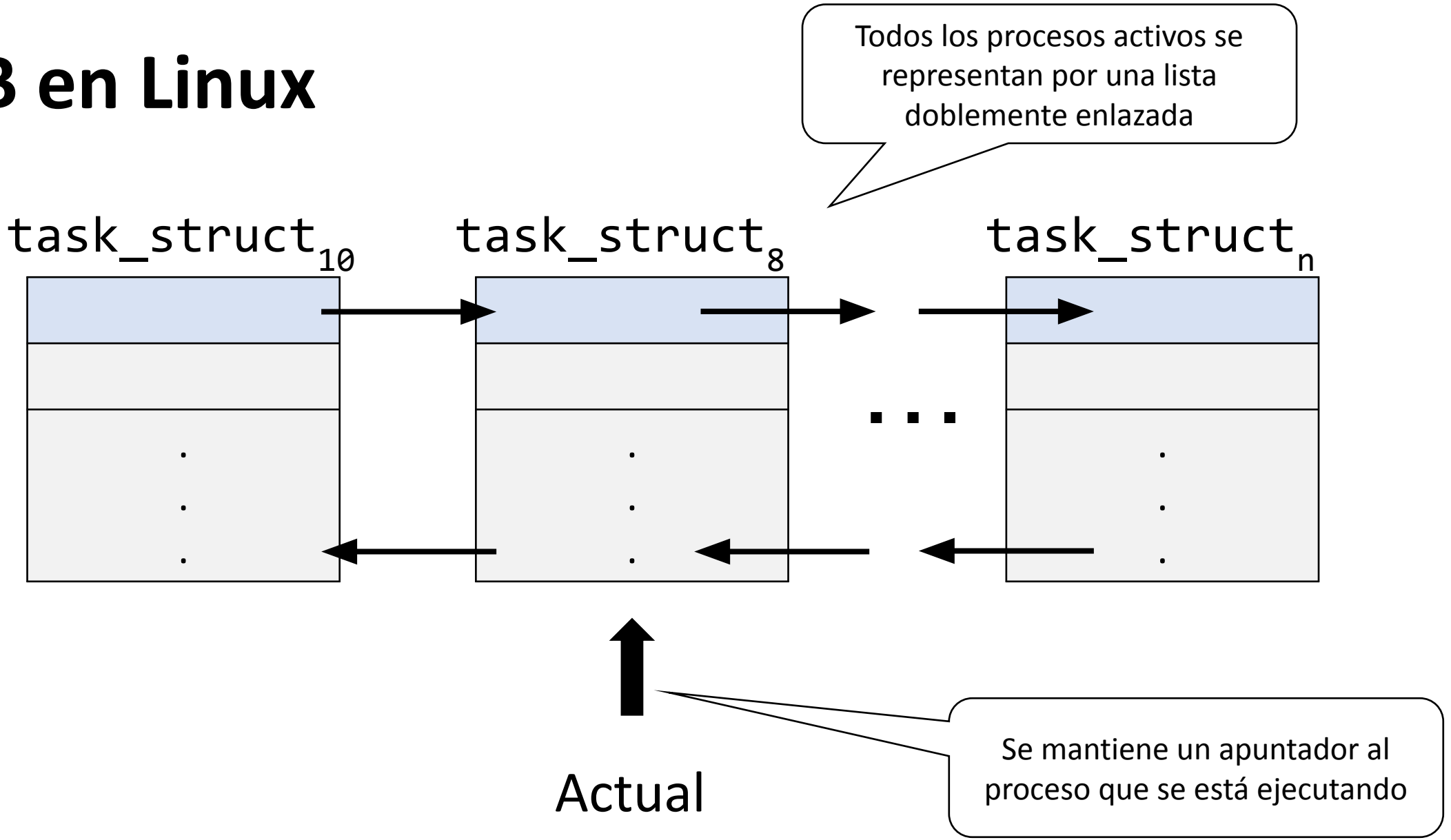
```
struct proc {
    char *mem;                  // Dónde inicial el proceso en memoria
    uint sz;                    // Tamaño del proceso en memoria
    char *kstack;               // Parte inferior pila Kernel para proceso
    enum proc_state state;      // Estado del proceso
    int pid;                    // ID del proceso
    struct proc *parent;        // Proceso padre
    void *chan;                 // Si !=0, proceso durmiendo
    int killed;                 // Si !=0 proceso killed
    struct file                 // Archivos abiertos
*ofile[NOFILE];                 // Directorio actual
    struct inode *cwd;          // Contexto del proceso
    struct context context;     // Para manejo de interrupciones
    struct trapframe *tf;
};
```

PCB en Linux

```
long state;                          // Estado del proceso
struct sched_entity se;              // Información del planificador
struct task struct *parent;          // Proceso padre de este proceso
struct list head children;           // Hijos de este proceso
struct files struct *files;          // Lista de archivos abiertos
struct mm_struct *mm;                // Espacio de direcciones en RAM
```

- El PCB en Linux está representador por la estructura **`task_struct`**.
- **Algunos** miembros de esta estructura son los indicados en el código anterior.
- La estructura **`task_struct`** está definida en **`<include/linux/sched.h>`**

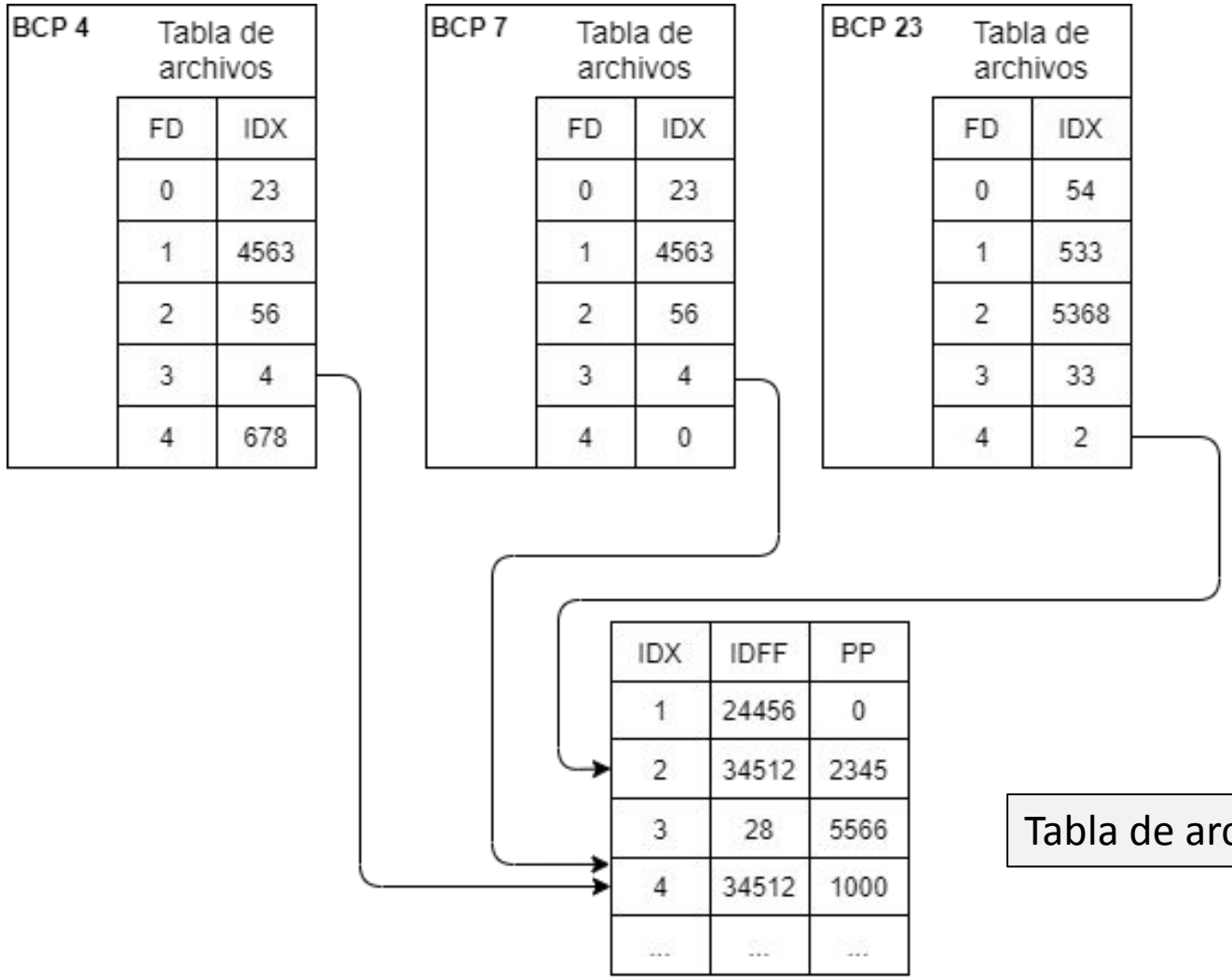
PCB en Linux


Todos los procesos activos se representan por una lista doblemente enlazada
task_struct10
task_struct8
task_structn
Actual
Se mantiene un apuntador al proceso que se está ejecutando

PCB e información compartida

- La información compartida por procesos no puede residir en el PCB
  - El PCB es único a cada proceso.
  - Información restringida de cada proceso.
- El PCB debe incluir un apuntador a la información compartida
- Considere el caso de dos procesos que abrieron el mismo archivo
  - Archivo se heredó de proceso padre.
  - Archivo se abrió de manera independiente por los dos procesos
  - Se debe compartir el puntero de posición
  - Puntero de posición no debe estar en el BCP

PCB/BCP e información compartida



- Cada apertura del archivo genera una nueva entrada en la tabla de archivos externa.
- Cada proceso tiene su PP puntero de posición en el archivo.
- BCP 7 es hijo de BCP 4, están compartiendo el mismo PP.
- El PP no puede estar en el BCP: considere casos de BCP 4 y BCP 7

Tabla de archivos externa al PCB/BCP

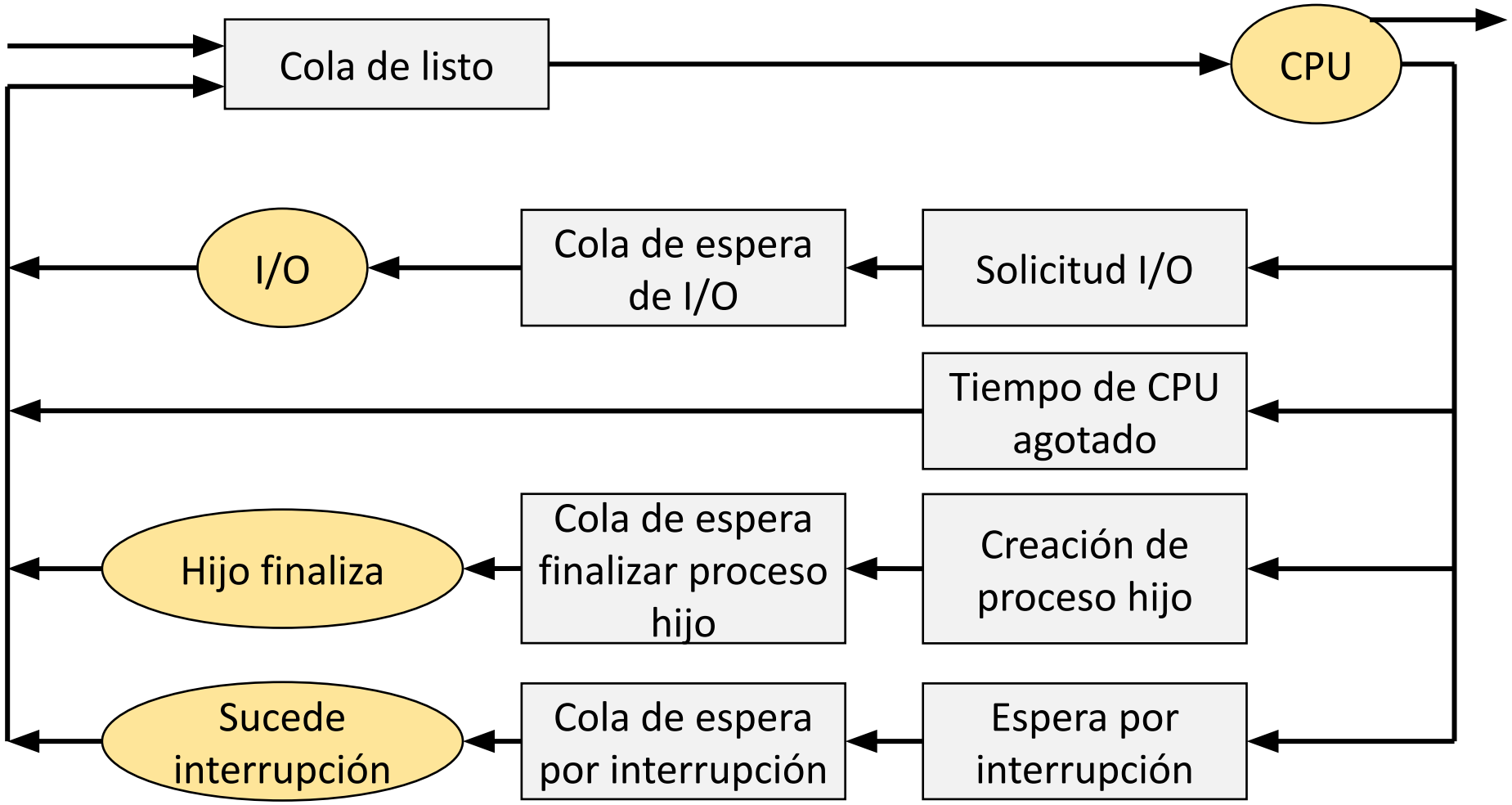
Planificación de procesos

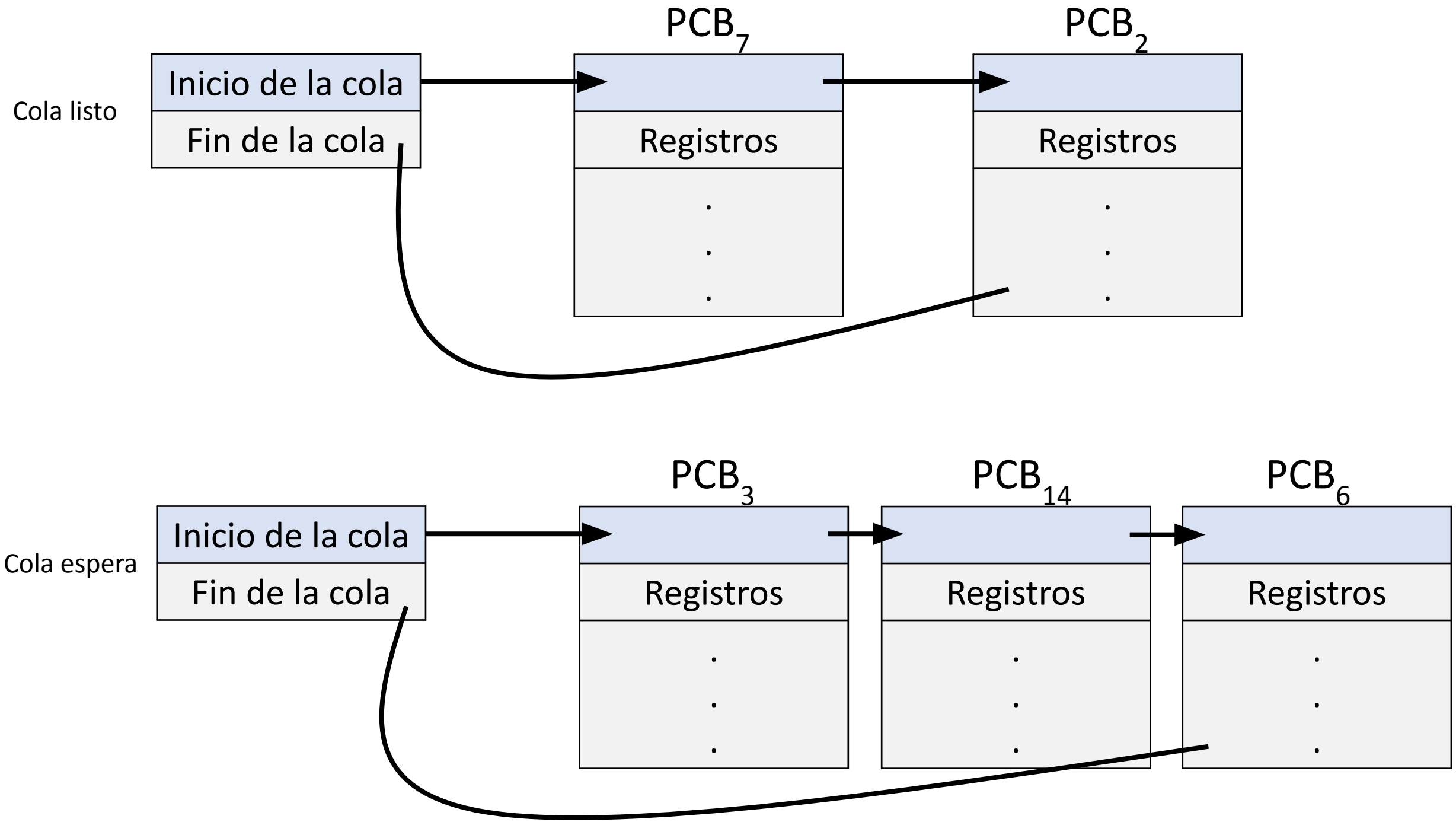
- Objetivos
  - Maximizar el uso del procesador
  - Compartir en el tiempo el uso del procesador por todos los procesos que lo requieran
    - De la mejor manera posible
    - Ningún proceso debe monopolizar el tiempo de procesador
- Cada núcleo de procesador puede ejecutar un proceso a la vez
  - Sistemas multinúcleo pueden ejecutar más de un proceso a la vez
- Grado de multi programación
  - Número de procesos actualmente en memoria

Colas de planificación de procesos

- Estado de **listo** es una cola
  - Proceso queda a la espera de que se le asigne tiempo de procesador
  - Usualmente se implementa como una lista enlazada (ligada)
  - Encabezado de la cola apunta al primer PCB en la lista
  - Cada PCB incluye un apuntador al siguiente PCB en la lista
- El S.O implementa varias colas para manejar los diferentes estados de un proceso

Colas: una de listo y tres de espera




Cola listo
Inicio de la cola
Fin de la cola
PCB7
Registros
PCB2
Registros
Cola espera
Inicio de la cola
Fin de la cola
PCB3
Registros
PCB14
Registros
PCB6
Registros

Colas: una de listo y tres de espera

- Los círculos/óvalos indican los recursos que sirven a las colas
- Flechas indican el flujo de un proceso en el sistema
- Eventos que pueden suceder cuando se ejecuta un proceso
  - Hace una solicitud de I/O. P. Ej.: la lectura de un archivo Cola de I/O.
  - Crea un proceso hijo. Cola hasta que hijo termine.
  - Sale de CPU por tiempo agotado o por alguna interrupción del proceso.
- Tres colas de espera para representar el estado de espera o bloqueado de un proceso

Planificación de CPU

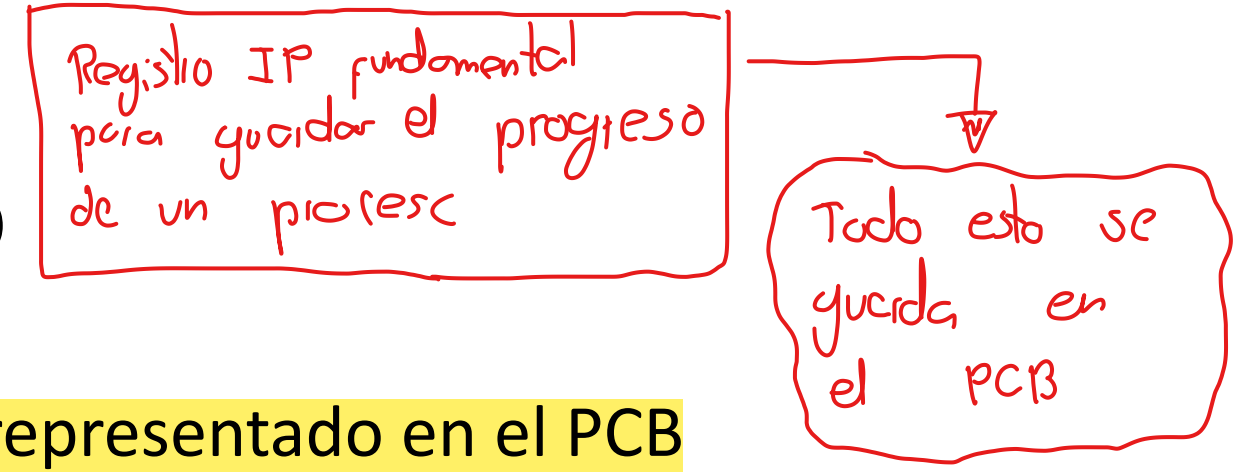
- Objetivos del planificador
  - Seleccionar un proceso (entre varios) para ejecutarse
  - De la mejor manera posible
- Procesos intensivos de I/O
  - Se ejecutan unos milisegundos antes de quedar en espera por I/O
- Procesos intensivos de CPU
  - Requieren más tiempo de uso de CPU
- Ningún proceso puede monopolizar el tiempo de CPU

Cambios de contexto

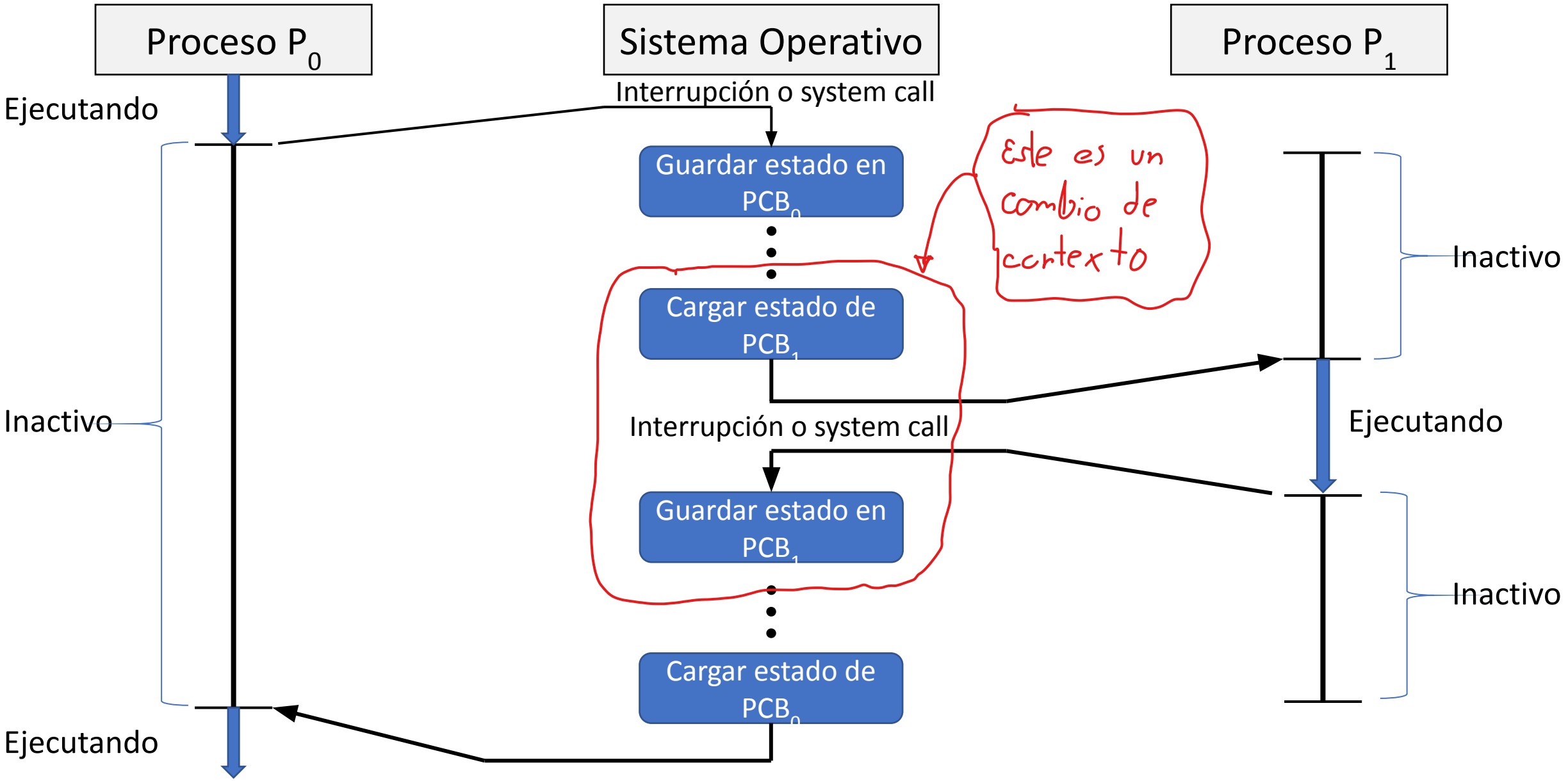
Cuando un proceso es interumpido

- Cuando ocurre una interrupción (p. ej.: I/O) el S.O debe sacar de CPU al proceso en ejecución
  - Para atender la interrupción
  - El S.O ejecuta código que atiende la interrupción
  - Debe ejecutar otro proceso distinto en la CPU: Un proceso propio del S.O
- Se debe guardar el contexto del proceso interrumpido
  - Proceso que se estaba ejecutando antes de la interrupción
  - Para seguir ejecutándolo cuando se le asigne nuevamente tiempo de CPU

Cambios de contexto



- El contexto de un proceso está representado en el PCB
- El cambio de contexto implica al núcleo del S.O
  - Guardar el contexto del proceso interrumpido en el PCB
  - Cargar el contexto del proceso que será ejecutado
  - Recordar que cada vez que se crea un proceso se crea el PCB del proceso
- El cambio de contexto es un gasto necesario (*overhead*).
  - El sistema no está haciendo nada útil mientras realiza el cambio de contexto
    - CPU no se usa en este cambio de contexto
  - Unos cuantos nano/**micro**/mili/segundos
    - Velocidad de la memoria RAM, número de registros, instrucciones especiales


Proceso P_0
Sistema Operativo
Proceso P_1
Ejecutando
Interrupción o system call
Guardar estado en PCB_0
Este es un cambio de contexto
Cargar estado de PCB_1
Inactivo
Inactivo
Interrupción o system call
Ejecutando
Guardar estado en PCB_1
Inactivo
Cargar estado de PCB_0
Ejecutando

Llamadas al sistema y cambios de contexto

- Consideremos la siguiente llamada al sistema

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

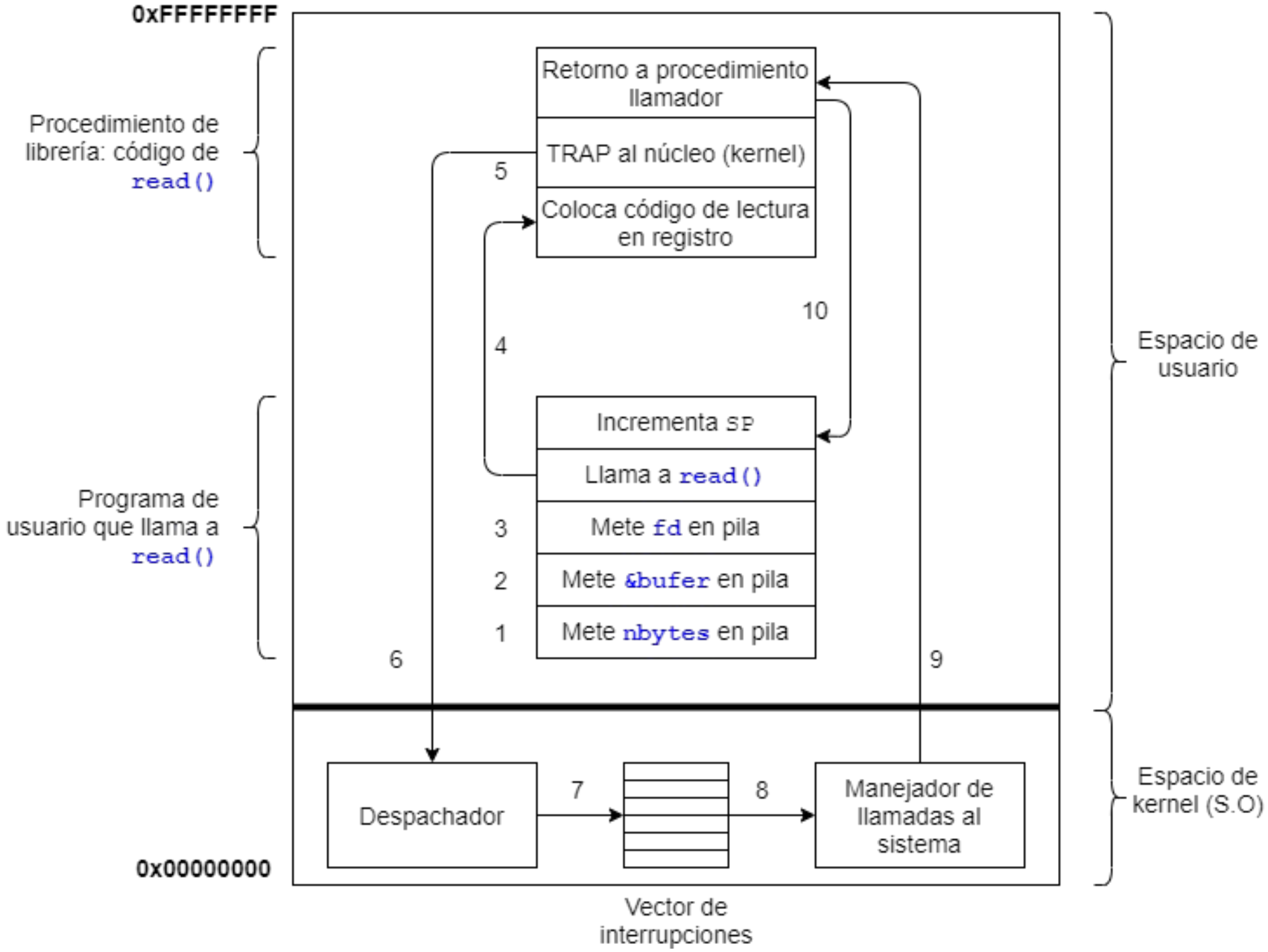
- Donde,
  - `<unistd.h>` archivo de cabecera que debe incluirse para hacer la llamada
  - `ssize_t` tipo de dato que retorna: entero con signo (POSIX.1.)
  - `read` nombre de la llamada
  - `int fd` manejador (*handle*) de un archivo abierto
  - `void *buf` apuntador al búfer para recibir los datos leídos
  - `size_t count` número de bytes a leer.
- La llamada `read()` retorna el número de bytes leídos

Llamadas al sistema y cambios de contexto

- Programa llamador mete parámetros en pila en orden inverso.
  - Primer y tercer parámetro por valor
  - Segundo parámetro por referencia
- Se hace llamada a `read()`
- Código de `read()` coloca en registro código de la llamada al sistema
  - El S.O espera que el código esté en ese registro
- Código de `read()` ejecuta instrucción TRAP
  - Interrupción por software
- Se ejecuta código del núcleo del S.O que examina número de llamada

Llamadas al sistema y cambios de contexto

- Pasa control a manejador de llamadas al sistema
  - En el núcleo del S.O.
  - Atiende llamada
- Se devuelve control al código de `read()` posterior a la instrucción TRAP
- Se devuelve control al programa llamador a la instrucción posterior a `read()`
- Programa llamador limpia la pila
  - Incrementa el apuntador de pila


0xFFFFFFFF
Procedimiento de librería: código de read()
Programa de usuario que llama a read()
Retorno a procedimiento llamador
TRAP al núcleo (kernel)
Coloca código de lectura en registro
Incrementa SP
Llama a read()
Mete fd en pila
Mete &bufer en pila
Mete nbytes en pila
1
2
3
4
5
6
7
8
9
10
Despachador
Manejador de llamadas al sistema
Espacio de usuario
Espacio de kernel (S.O)
0x00000000
Vector de interrupciones

Llamadas al sistema y cambios de contexto

- ¿Cuántos cambios de contexto hubo en el caso anterior?
  - Una llamada al sistema no siempre produce un cambio de contexto a menos que la llamada sea bloqueante.
  - Si la llamada es bloqueante se aprovecha el tiempo para darle tiempo de procesador a otro proceso.

Vector de interrupciones

- Cada entrada del vector contiene
  - La dirección de la rutina de tratamiento de esa interrupción
  - El vector está indexado por el número de interrupción
- Se transfiere el control (se ejecuta) las instrucciones a las que apunta la entrada en el vector

Sistemas operativos monotarea

- **Monotarea o monoproceso**
  - Solo existe un proceso a cada instante en el procesador
  - El segundo proceso no puede ejecutarse hasta que el primero halla finalizado completamente
  - La memoria RAM la ocupa el S.O y el proceso en ejecución.
  - MS-DOS
- **Multitarea o multiproceso**
  - Permite la coexistencia de varios procesos activos a la vez
  - El S.O (planificador y despachador) selecciona un proceso para ejecutarse
  - Se requiere **cambio de contexto** entre proceso y proceso.

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.
- Tanenbaum, A. S. (2009). Introducción. In *Sistemas Operativos Modernos* (3rd ed., pp. 3–75). Pearson Educación.

Operaciones sobre procesos

Adaptación de diferentes referencias bibliográficas

Juan Felipe Muñoz Fernández

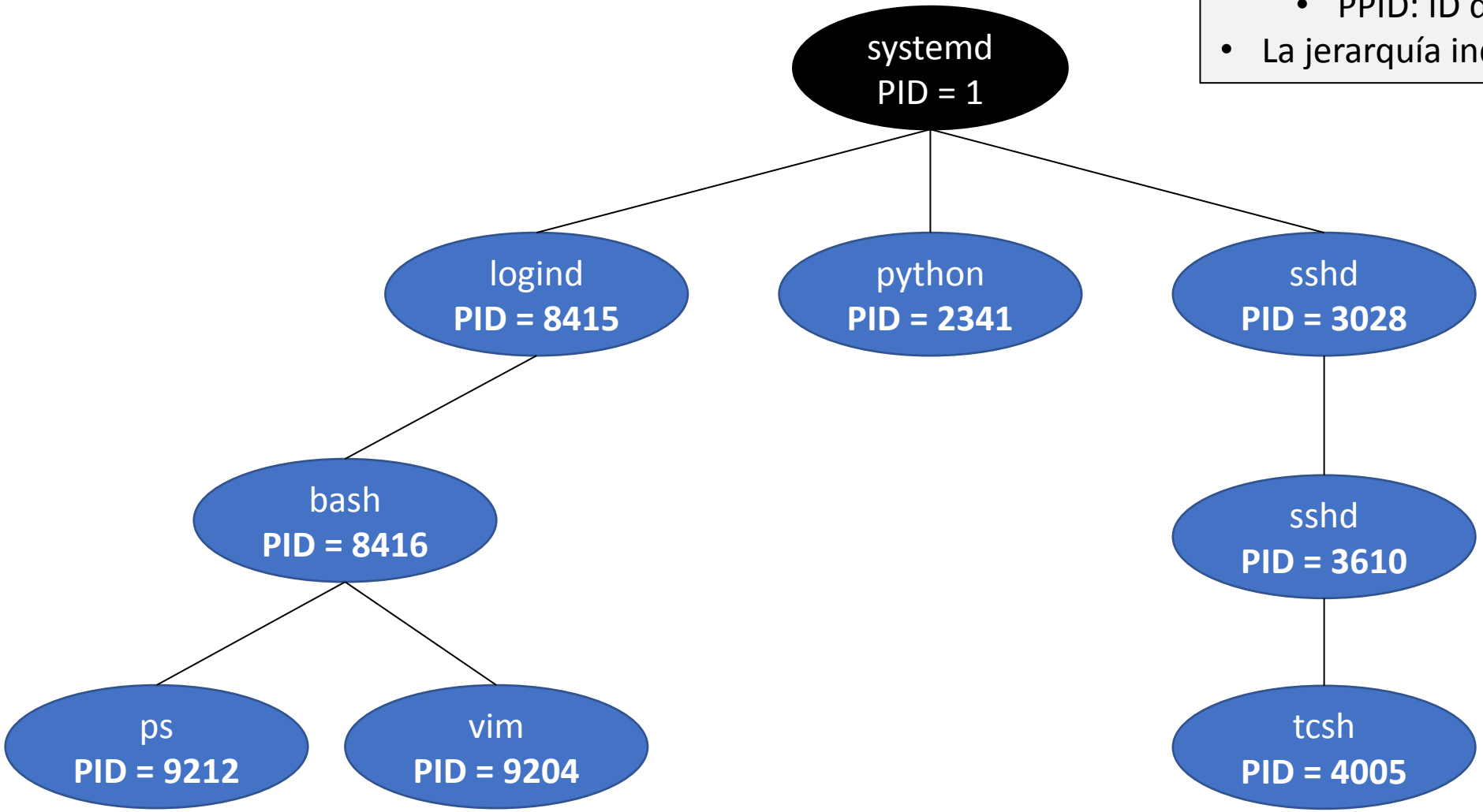
Preguntas

- ¿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.

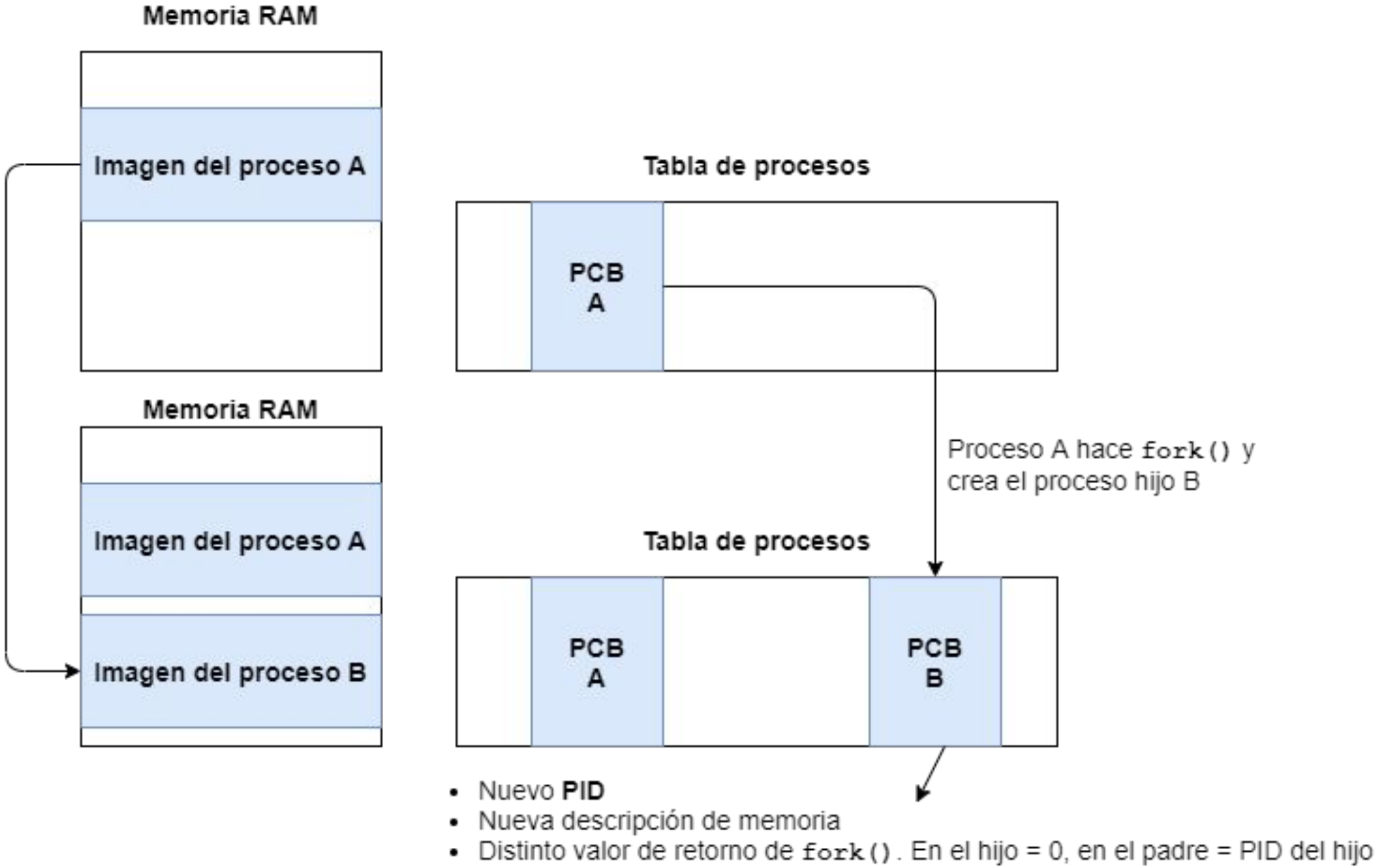
p01-fork.c

```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…

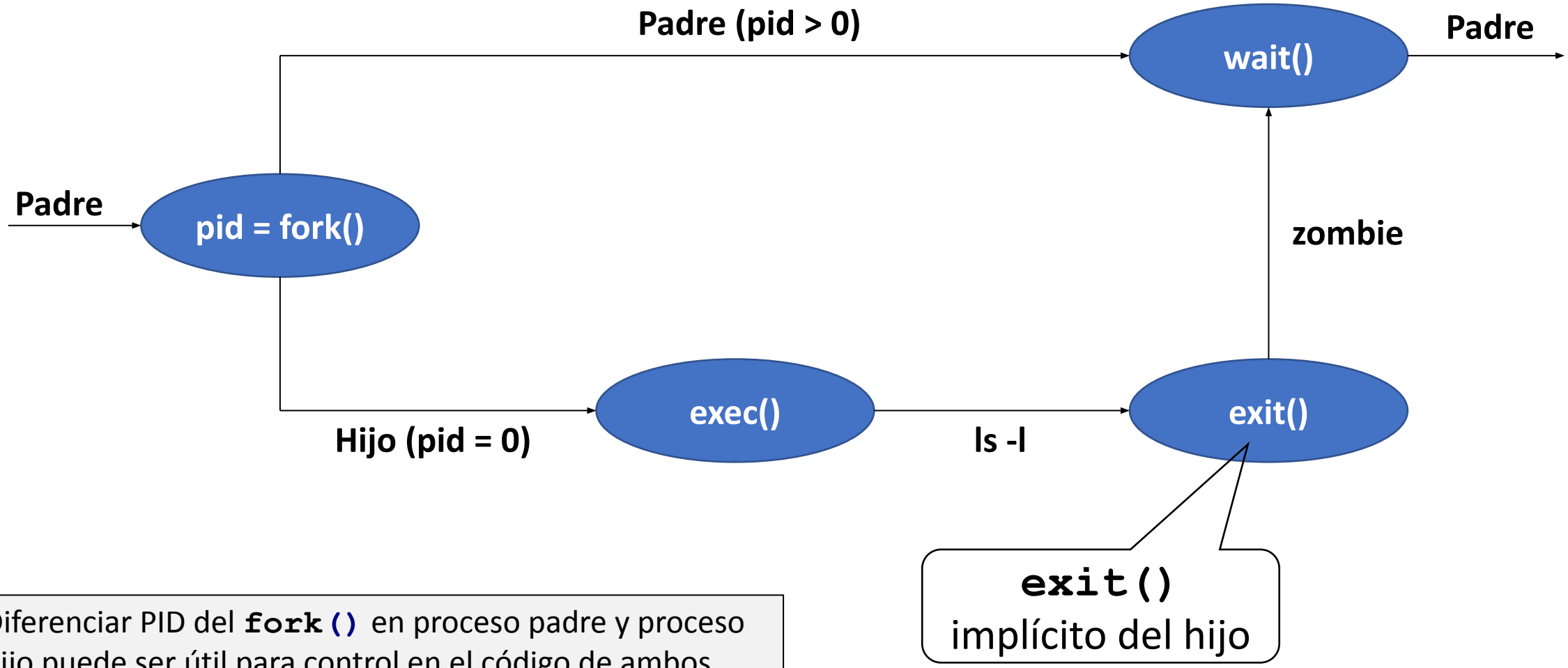
- Datos y pila de **B** son iguales a los de **A** en instante de `fork()`.
- Registro **IP** tiene mismo valor para **A** y **B** en `fork()`.
- Proceso **B** su propio PID.
- Proceso **B** no está en la misma zona de memoria de **A**.
- **B** con copia de descriptores de **A**
  - Archivos abiertos 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

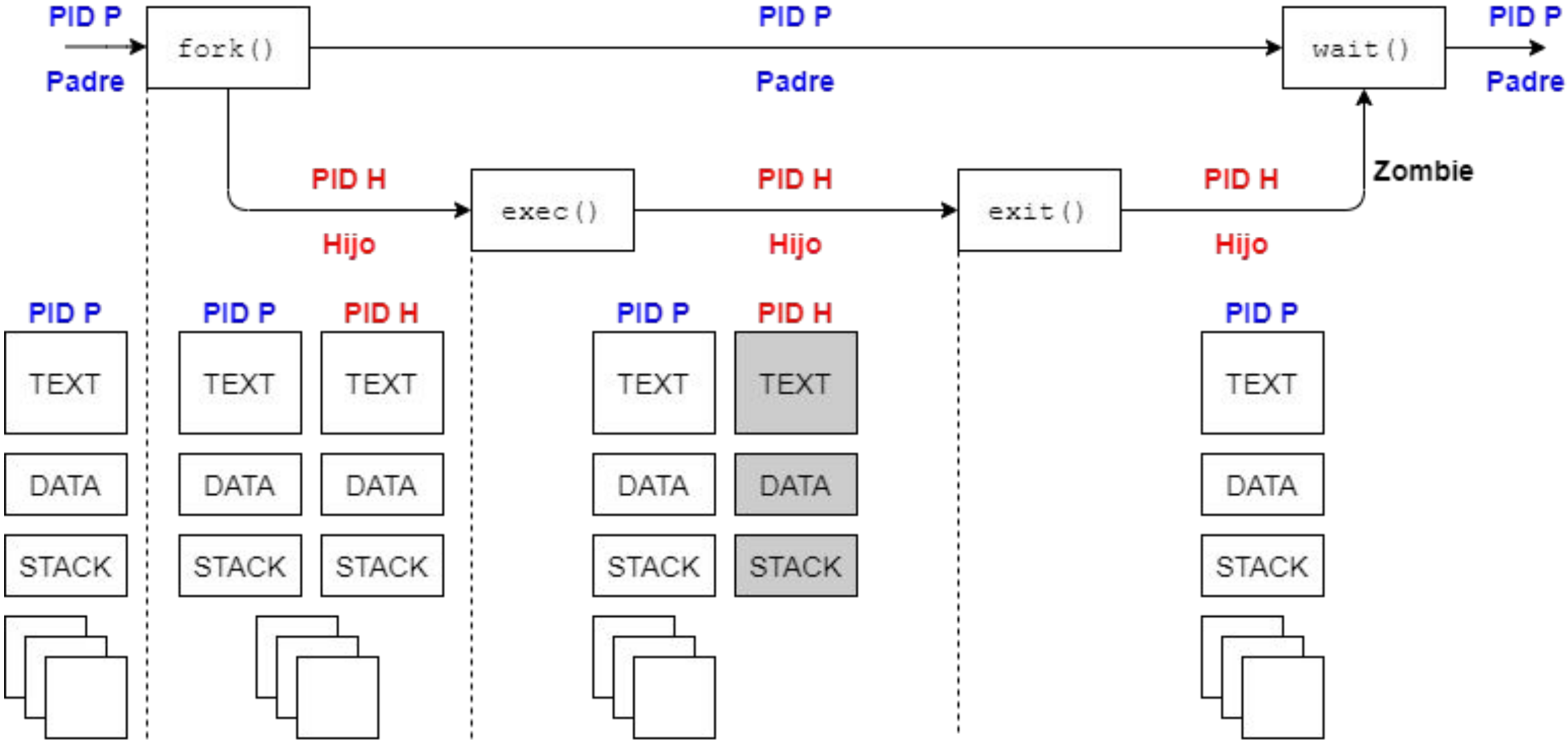
```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main(){
    pid_t pid; /* Se crea nuevo proceso */
    pid = fork();
    if (pid < 0) { /* error en fork() */
        printf("Falló fork()");
        return 1;
    }
    else if (pid == 0) { /* Proceso hijo */
        execlp("/bin/ls","ls","-l",NULL);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

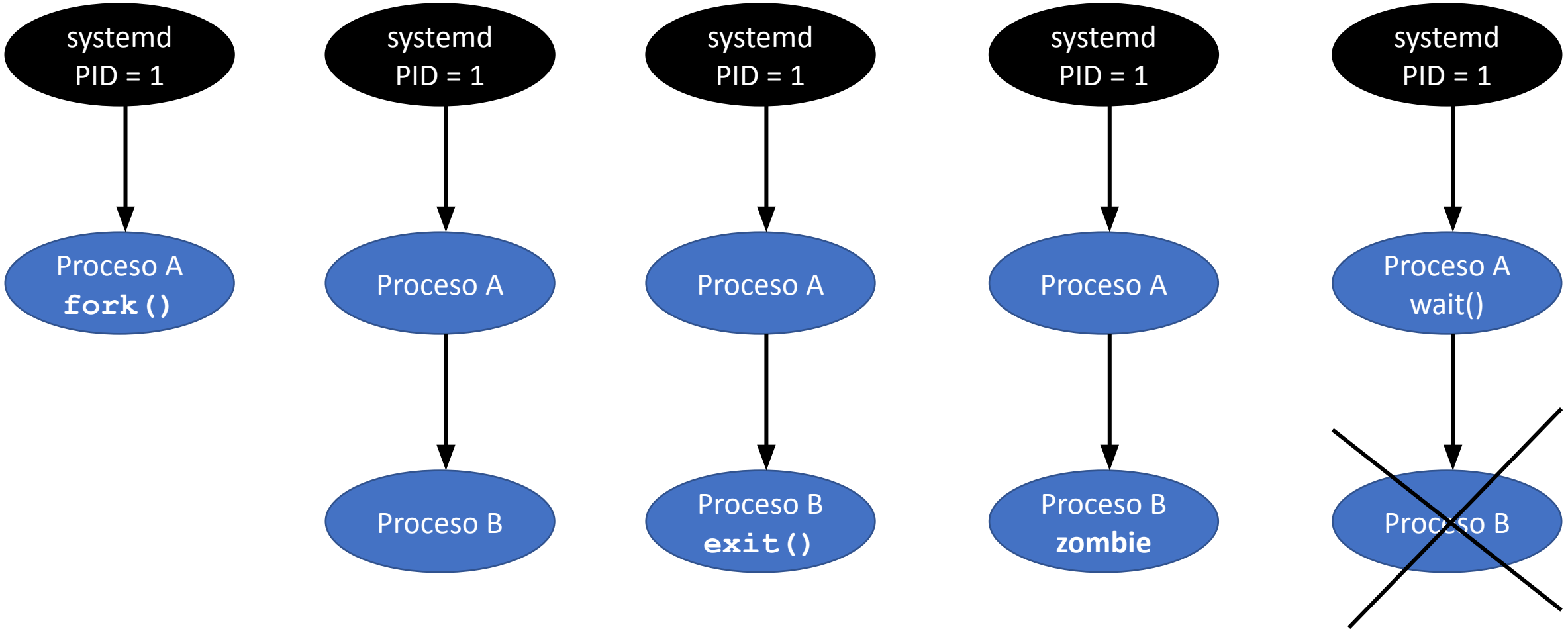
Terminación de un proceso

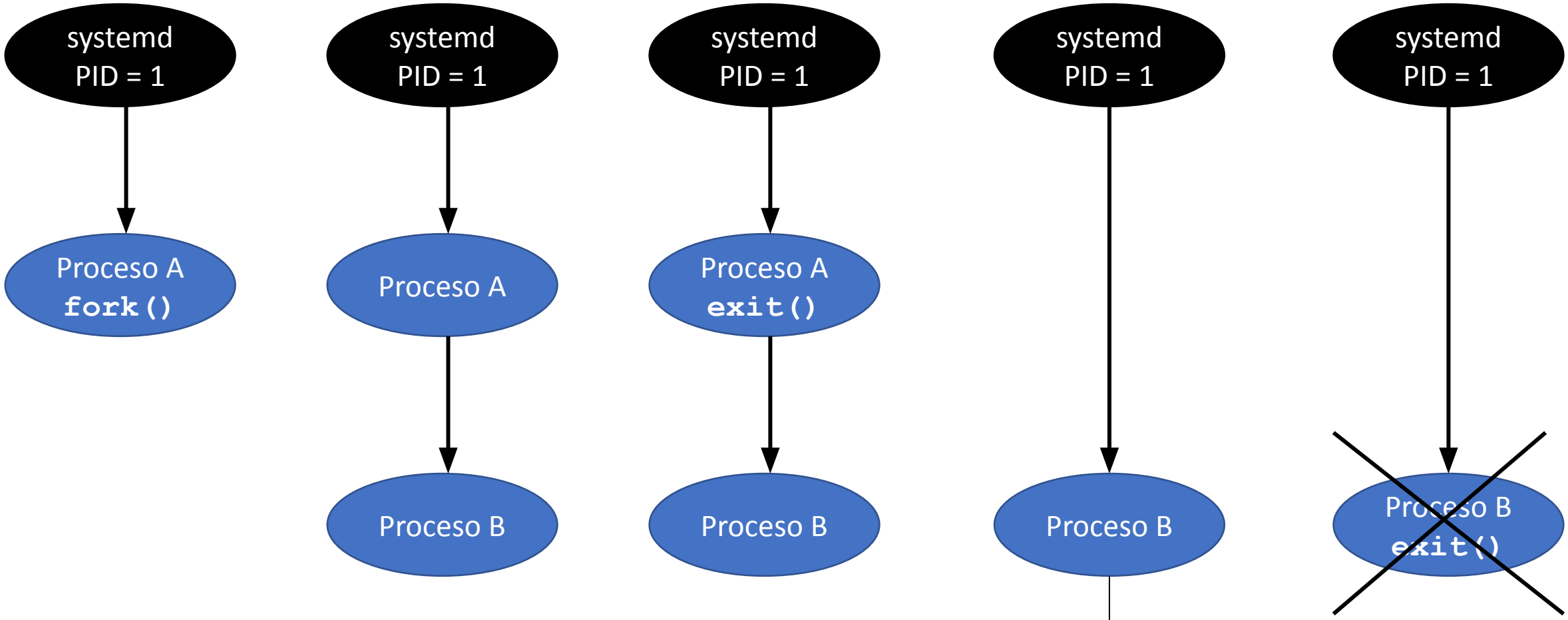
echo %errorlevel% → Windows
echo $? → Linux
Retorno último proceso ejecutado

- 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.

Hilos

Adaptación de múltiples referencias bibliográficas

Juan Felipe Muñoz Fernández

Definición

- Unidad básica de utilización de CPU. Información propia de cada hilo:
  - ID del hilo (thread)
  - Registro IP
  - Conjunto de registros
  - Pila
- Tradicionalmente: un proceso un hilo de ejecución: `main()`.
- Un proceso puede crear más de un hilo, los hilos comparten
  - Sección de código
  - Sección de datos
  - Archivos abiertos
  - Procesos hijos

Definición

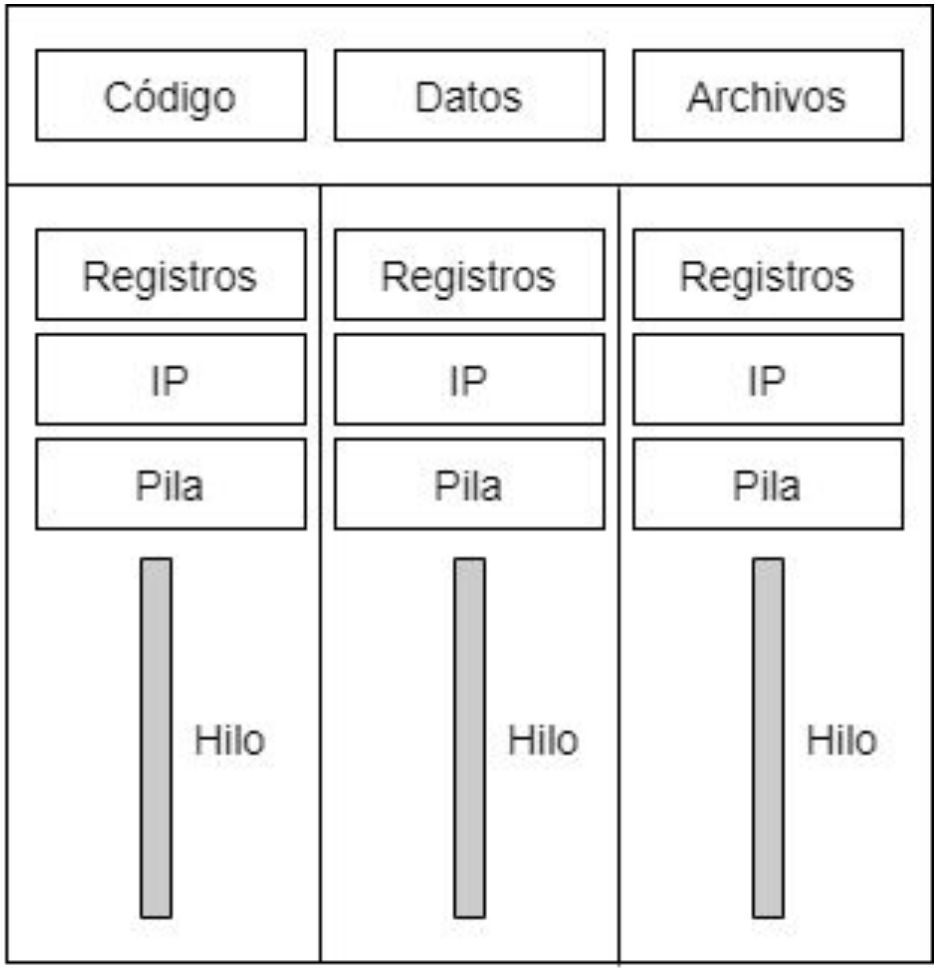
- Los hilos están dentro del mapa de memoria del proceso.
- Cada hilo es un flujo de ejecución.
- Todos los hilos comparten la imagen de memoria.
- Un hilo es una función que se puede ejecutar en paralelo
- Un hilo puede estar ejecutando, listo o bloqueado.

Motivación

- Aplicaciones modernas se implementan como procesos separados con varios hilos de control.
- Procesador de texto multihilo
  - Hilo que muestra documento
  - Hilo que revisa ortografía
  - Hilo que realiza operaciones de autoguardado cada cierto tiempo (en
  - Hilo que procesa las pulsaciones de teclado
- Procesador de texto un solo hilo
  - Todas las funciones anteriores pero frente a la ejecución de una de ellas, las demás deben esperar

Motivación


Código
Datos
Archivos
Registros
IP
Pila
Hilo principal


Código
Datos
Archivos
Registros
IP
Pila
Hilo
Registros
IP
Pila
Hilo
Registros
IP
Pila
Hilo

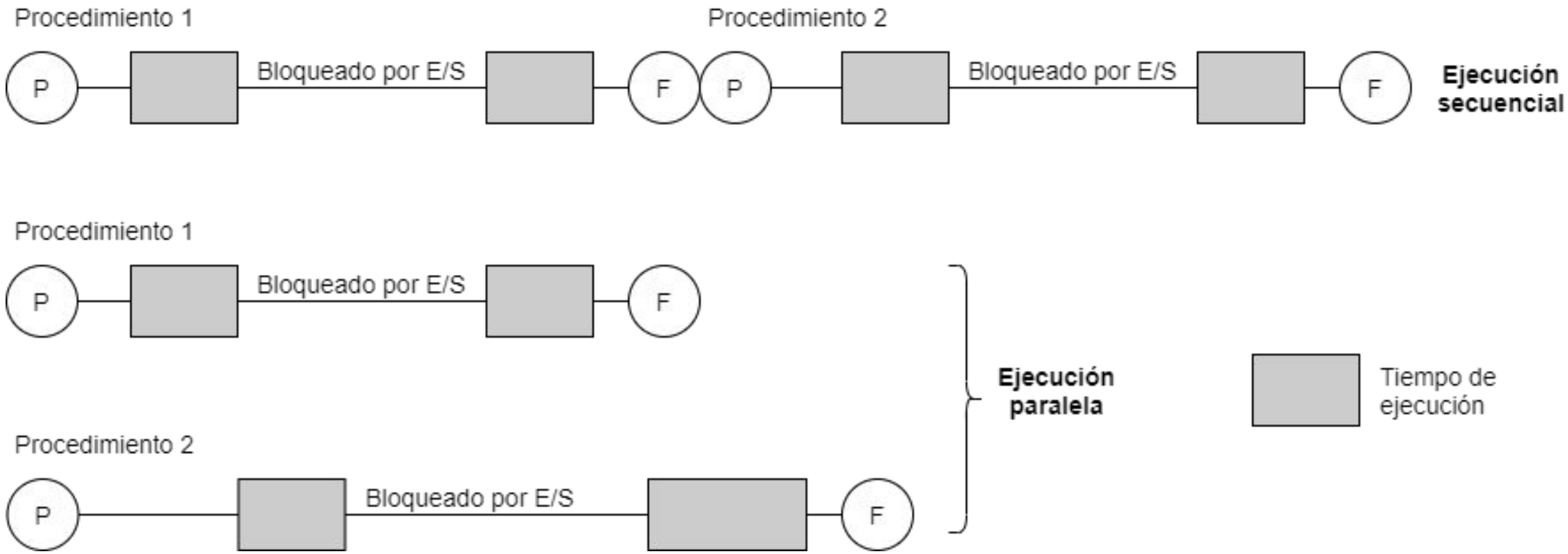
Motivación

- Aprovechar las capacidades de sistemas multinúcleo
- Es más eficiente crear hilos que procesos
  - La creación y terminación de un proceso tiene más *overhead* que la de un hilo.
- Programas intensivos de CPU hay que diseñarlos con capacidades de procesamiento paralelo.
  - Problemas de ordenamiento
  - Árboles
  - Gráficos
  - Problemas de minería de datos
  - Problemas de IA

Estados de los hilos

- Tres estados posibles
  - Ejecutando
  - Listo para ejecutar
  - Bloqueado
- El estado del proceso será la combinación de los estados de sus hilos
  - Si un hilo en ejecución entonces estado del proceso en ejecución.
  - Si no hay hilos en ejecución pero hay un hilo en estado de listo, entonces proceso está en estado de listo.
  - Si todos los hilos están bloqueados, entonces estado del proceso es bloqueado.

Paralelismo



Paralelismo

- La base del paralelismo está en que mientras exista un hilo bloqueado, otro hilo puede estar ejecutándose.
- Concurrencia y paralelismo son dos cosas diferentes
- Hilos
  - Permiten variables compartidas y paralelismo
  - Usan llamadas bloqueantes
- Proceso convencional de un solo hilo
  - No hay paralelismo
  - Usa llamadas al sistema bloqueantes
- Varios procesos convencionales cooperativos
  - Permiten paralelismo
  - No comparten variables
  - Se requieren mecanismos de IPC para compartir información.

Desafíos desde la programación

- Sistemas multinúcleo requieren diseños de software que aprovechen dichas capacidades
- Diseñadores de S.Os deben escribir algoritmos de despacho de procesos e hilos que aprovechen todos los núcleos.
- Cada núcleo se ve como un procesador independiente para el S.O.

Desafíos desde la programación

- Identificar tareas
  - Identificar en un proceso cómo se puede dividir en tareas concurrentes
  - Idealmente tareas independientes
- Balance
  - Tareas que hagan un trabajo igual de igual valor para el proceso general
  - Algunas tareas en paralelo puede que no aporten mucho al proceso en general
- División de datos
  - Datos deben dividirse para cada una de las tareas que se ejecutan en núcleos independientes
- Dependencia de datos
  - Sincronizar ejecución cuando existe dependencia de datos

Desafíos desde la programación

- Pruebas y depuración
  - Tareas paralelas tienen diferentes caminos de ejecución.
  - Depuración y pruebas es mucho más difícil.
- Se requieren nuevos paradigmas de diseño de software
  - No seguir pensando en modelos secuenciales
  - Pensar en modelos de ejecución paralela
- Programación con alto nivel de dificultad
  - Acceso a datos compartidos se haga de forma correcta.
  - Accesos incorrectos a variables: se comparten variables globales.
  - Implementar mecanismos de sincronización de procesos/hilos.

Ejecución sincrónica y asincrónica

- **Ejecución sincrónica**
  - Hilo padre crea hilo(s) y continua su ejecución.
  - Papá e hijo se ejecutan concurrentemente e independientemente.
  - Hilos independientes hay pocos datos compartidos por ambos hilos.
- **Ejecución asincrónica**
  - Hilo padre crea uno o más hilos hijos.
  - Hilo padre espera a que hijos terminen para poder continuar.
  - Hilo terminado se une a hilo padre.
  - Hilo padre continua ejecución cuando todos los hijos terminen
  - Comparten muchos datos entre todos lo hilos
  - Hilo padre combina resultados de hilos hijos.

Modelos de multihilos

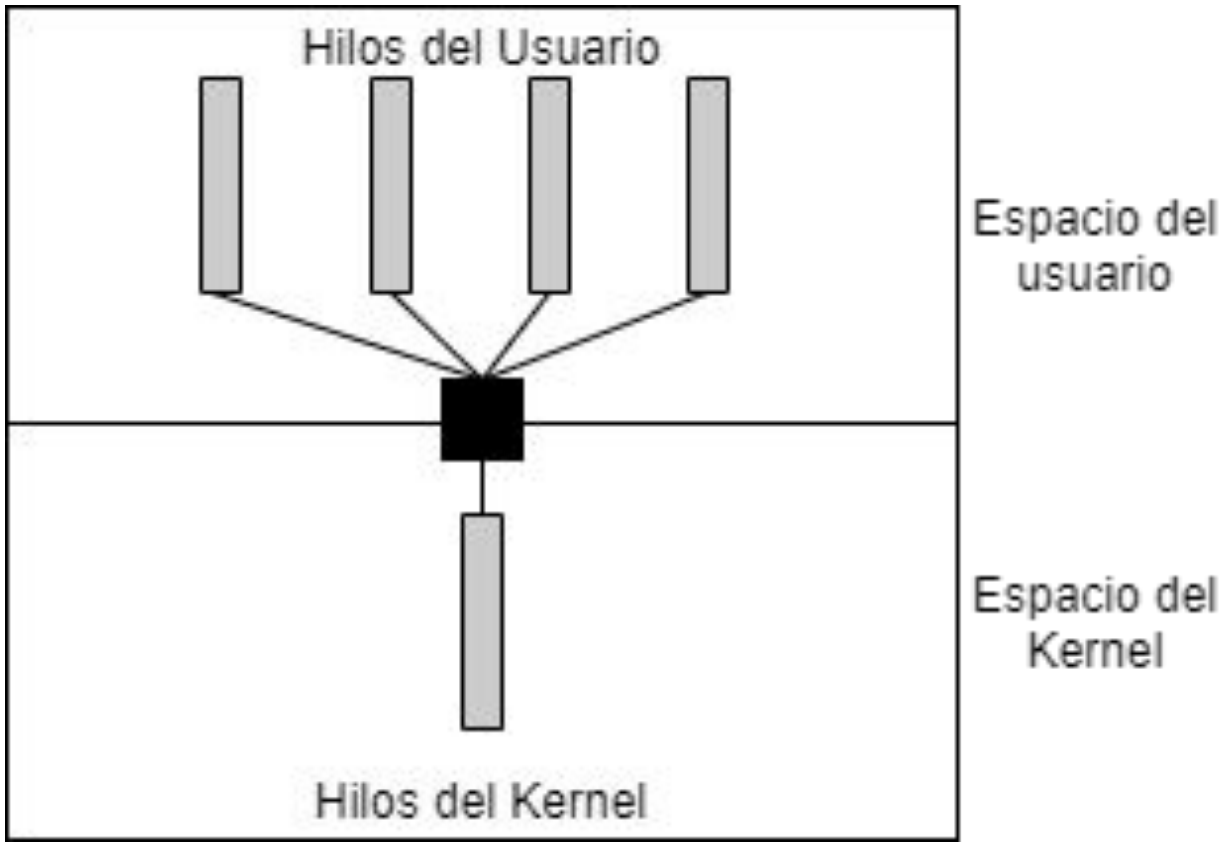
- Se deben suministrar mecanismos de soporte de hilos
  - A nivel del espacio del usuario
  - A nivel del núcleo (kernel) del sistema operativo
- Hilos a nivel de usuario
  - Se gestionen en el espacio del usuario, no requieren soporte del kernel.
- Hilos a nivel del kernel del sistema operativo
  - Se soportan y se administran directamente por el kernel del sistema operativo
- Debe existir una relación entre hilos en el espacio de usuario e hilos en el nivel del kernel.

Multithreading | Hyperthreading

- Capacidad de procesadores modernos de permitir paralelismo de instrucciones.
- Procesadores modernos tienen más de un núcleo
  - Cada núcleo es dividido en dos o más procesadores lógicos
  - Cada procesador lógico soporta un hilo de ejecución
- P. Ej.: Intel Core i5-6200u
  - Cantidad de núcleos: 2
  - Cantidad de subprocesos (hilos): 2
  - Sistema operativo Windows considera 4 procesadores lógicos

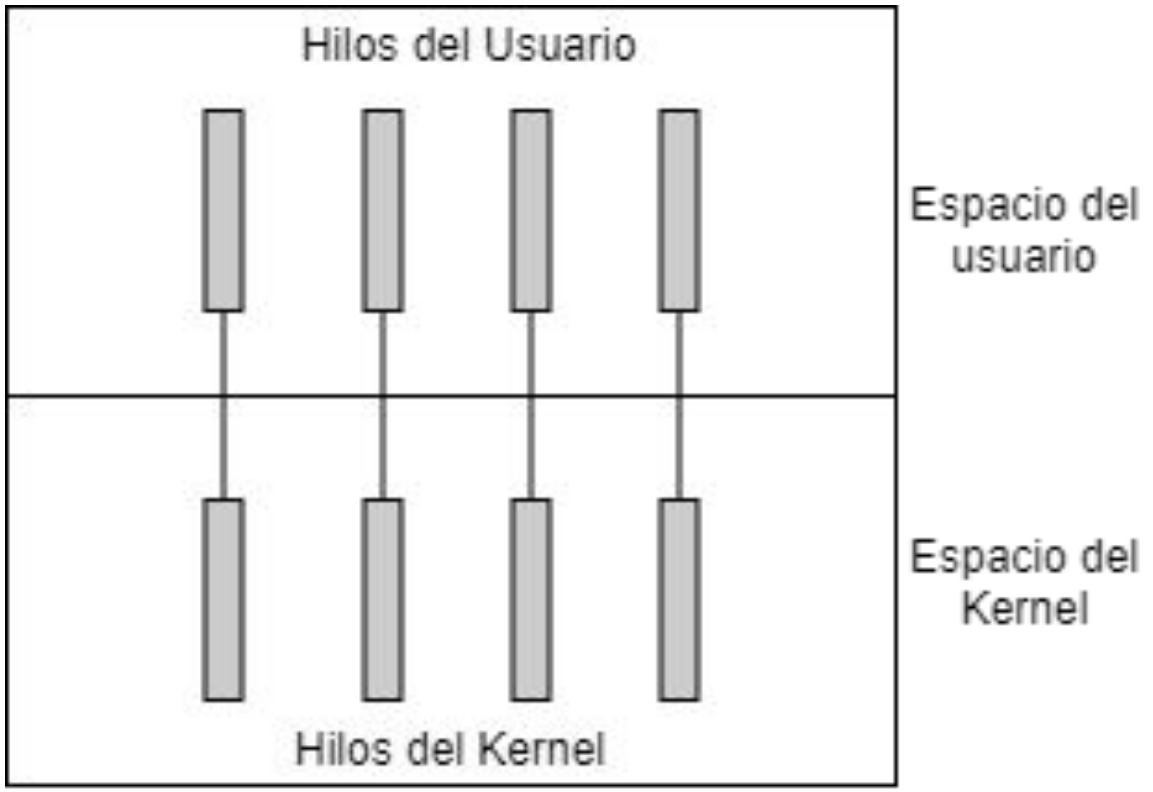
Modelos de multihilos: muchos a uno

- Hilos los gestionan librería en espacio del usuario
- Un hilo hace una *system call* bloqueante
  - Se bloquea todo el proceso
- Un solo hilo puede acceder al Kernel
  - No hay paralelismo en sistemas multi núcleo.
- Muy pocos sistemas ofrecen este modelo



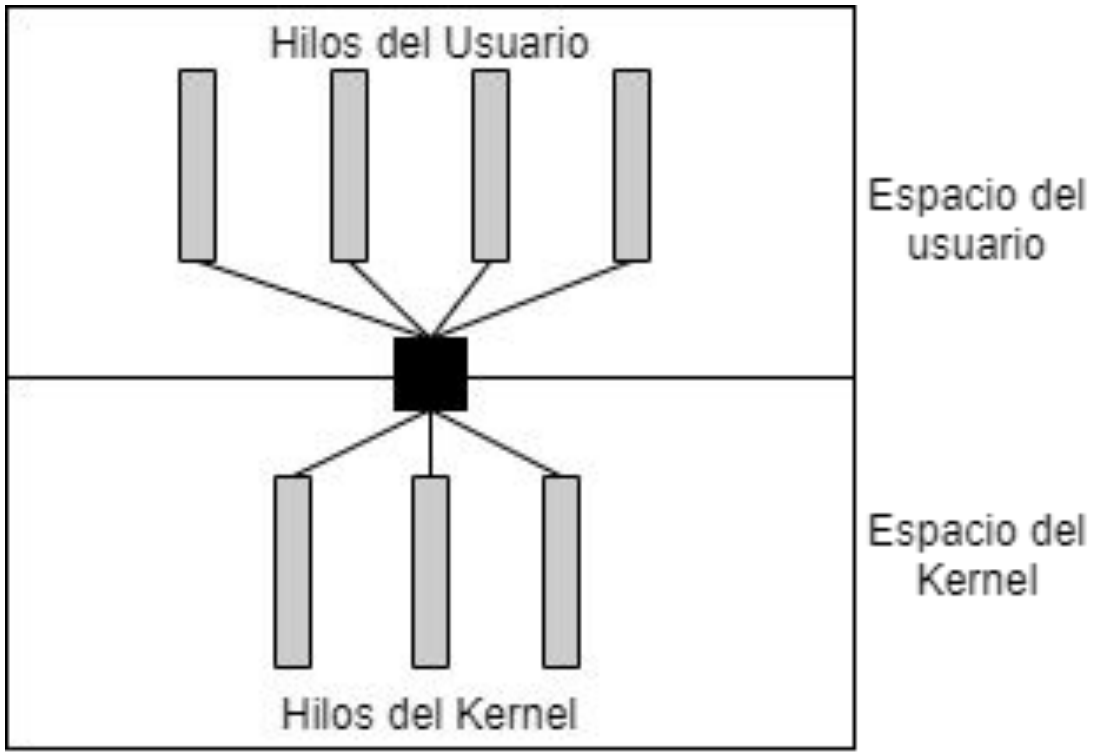
Modelos de multihilos: uno a uno

- Asociación entre un hilo del usuario contra un hilo del kernel.
- Un hilo bloqueado NO bloquea al resto.
- Se permite paralelismo en sistemas multinúcleo.
- Castiga desempeño:
  - Por cada hilo de usuario se crea un hilo en el kernel.
- Linux y Windows implementan este modelo.



Modelos de multihilos: muchos a muchos

- Se multiplexan los hilos del espacio de usuario con un número menor o igual de hilos en Kernel.
- Permite paralelismo
- Número de hilos en Kernel deben ser específicos para situaciones particulares.
  - Más hilos de Kernel en máquina con 8 núcleos que en máquina de 4.
- Difícil de implementar



```c
#include <stdio.h>
#include <pthread.h>

void *mihilo(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main(): Inicia\n");
    pthread_create(&p1, NULL, mihilo, "Hilo A");
    pthread_create(&p2, NULL, mihilo, "Hilo B");

    /* Esperar a que hilos terminen */
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main(): Termina\n");
    return 0;
}
```

```
# ./hilos2
main(): Inicia
Hilo A
Hilo B
main(): Termina
```

main()	Hilo 1	Hilo 2
Inicia ejecución Imprime **`main(): Inicia`** Crea Hilo 1 Crea Hilo 2 Espera por Hilo 1		
	Se ejecuta Imprime **`Hilo A`** Retorna	
Espera por Hilo 2		
		Se ejecuta Imprime **`Hilo B`** Retorna
Imprime **`main(): Termina`**		

main()	Hilo 1	Hilo 2
Inicia ejecución Imprime **`main(): Inicia`** Crea Hilo 1		
	Se ejecuta Imprime **`Hilo A`** Retorna	
Crea Hilo 2		
		Se ejecuta Imprime **`Hilo B`** Retorna
Espera por Hilo1 *Retorna inmediatamente (ya terminó)*		
Espera por Hilo 2 *Retorna inmediatamente (ya terminó)*		
Imprime **`main(): Termina`**		

main()	Hilo 1	Hilo 2
Inicia ejecución Imprime **`main(): Inicia`** Crea Hilo 1 Crea Hilo 2		
		Se ejecuta Imprime **`Hilo B`** Retorna
Espera por Hilo 1		
	Se ejecuta Imprime **`Hilo A`** Retorna	
Espera por Hilo2 *Retorna inmediatamente (ya terminó)*		
Imprime **`main(): Termina`**		

Ejecución de hilos

- **No se puede asumir** que el hilo que se crea primero, es el primero en ejecutarse.
- La creación de hilo es una llamada a una función.
  - El sistema operativo crea un hilo de ejecución cuando se llama a la función.
  - El hilo se ejecuta de manera independiente del programa que llama la función.
  - Lo que se ejecuta después del llamado depende del planificador del S.O
    - Ya que los hilos también tiene diferentes estados pasan por el planificador del S.O.
  - Es difícil saber que se ejecutará en un momento dado.

```c
#include <pthread.h>
#include <stdio.h>

void * funcion(void *arg) {
    printf("Hilo %d \n", (int) pthread_self());
    pthread_exit(0);
}

int main(int argc, char *argv[]) {
    pthread_t h1, h2;

    pthread_create(&h1, NULL, funcion, NULL);
    pthread_create(&h2, NULL, funcion, NULL);

    printf("main() sigue su ejecución\n");
    printf("main() sigue su ejecución\n");
    printf("main() sigue su ejecución\n");

       /* Se espera a que terminen */
    pthread_join(h1, NULL);
    pthread_join(h2, NULL);

    printf("main() termina\n");
    return 0;
}
```

Hilado sincrónico:

- Hilo padre crea hilos hijos y sigue su ejecución.
- Hilos se ejecutan con independencia: padre e hijos.
- En algún punto hilo padre espera a hilos hijos: **`pthread_join()`**
- Se usa usualmente cuando hay muchos datos compartidos entre los hilos: hilo padre consolida la información de los hilos hijos cuando llama a **`pthread_join()`.**

```
# gcc -pthread -o hilos_sync hilos_sync.c
# ./hilos_sync
main() sigue su ejecución
main() sigue su ejecución
main() sigue su ejecución
Hilo 225703680
Hilo 234096384
main() termina
```

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void * funcion(void *arg) {
    printf("Hilo %d \n", (int) pthread_self());
    pthread_exit(0);
}

int main(int argc, char *argv[]) {
    int j;
    pthread_t hilos[10];
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    for (j = 0; j < 10; j++) {
        pthread_create(&hilos[j], &attr, funcion, NULL);
    }

    /* Se espera 5 segundos a que terminen hilos */
    sleep(5);

    printf("main() termina\n");
    return 0;
}
```

```
# gcc -pthread -o hilos_async hilos_async.c
# ./hilos_async
Hilo 208307968
Hilo 199915264
Hilo 241878784
Hilo 233486080
Hilo 225093376
Hilo 191522560
Hilo 183129856
Hilo 174737152
Hilo 216700672
Hilo 250271488
main() termina
```

Hilado asincrónico:

- Hilo padre crea hilos hijos y sigue su ejecución.
- Hilo padre e hilo(s) hijo(s) se ejecutan concurrentemente.
- No se comparten muchos datos entre los hilos.

Referencias

- Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). Concurrency: An Introduction. In *Operating Systems. Three Easy Pieces*. Arpaci-Dusseau Books.
- Carretero Pérez, J., García Carballeira, F., De Miguel Anasagasti, P., & Pérez Costoya, F. (2001). Procesos ligeros. In *Sistemas operativos. Una Visión Aplicada* (pp. 98–101). McGraw Hill.
- Silberschatz, A., Galvin B., P., & Gagne, G. (2018). Threads & Concurrency. In *Operating Systems Concepts* (10th ed., pp. 159–196). John Wiley & Sons, Inc.
- Tanenbaum, A. S. (2009). Hilos. In *Sistemas Operativos Modernos* (3rd ed., pp. 95–114). Pearson Educación S.A.