

# Tema 2: Gestión de Procesos

Grado en Ing. Informática - Grupo 2  
Curso 2024/25

Profesor: Manuel E. Acacio Sánchez

2.1 Introducción

2.2 Estados de un proceso

2.3 Implementación de procesos

2.4 Hilos (*threads*)

2.5 Planificación de procesos

## 2.1 Introducción

- La CPU es uno de los recursos más importantes
  - Para ejecutarse los programas necesitan acceso a la CPU
- Más procesos que CPUs (o *cores*)
  - ¿Qué procesos la usan? → Necesidad de **planificar** la CPU
  - Ocurre igual con otros recursos limitados (p.ej. la memoria)
- En los SSOO actuales distintos usuarios, e incluso un mismo usuario, realizan tareas concurrentemente:
  - Todo el software ejecutable se organiza en torno al concepto de proceso (*modelo de procesos*)
  - El SO es el encargado de gestionar la CPU como un recurso

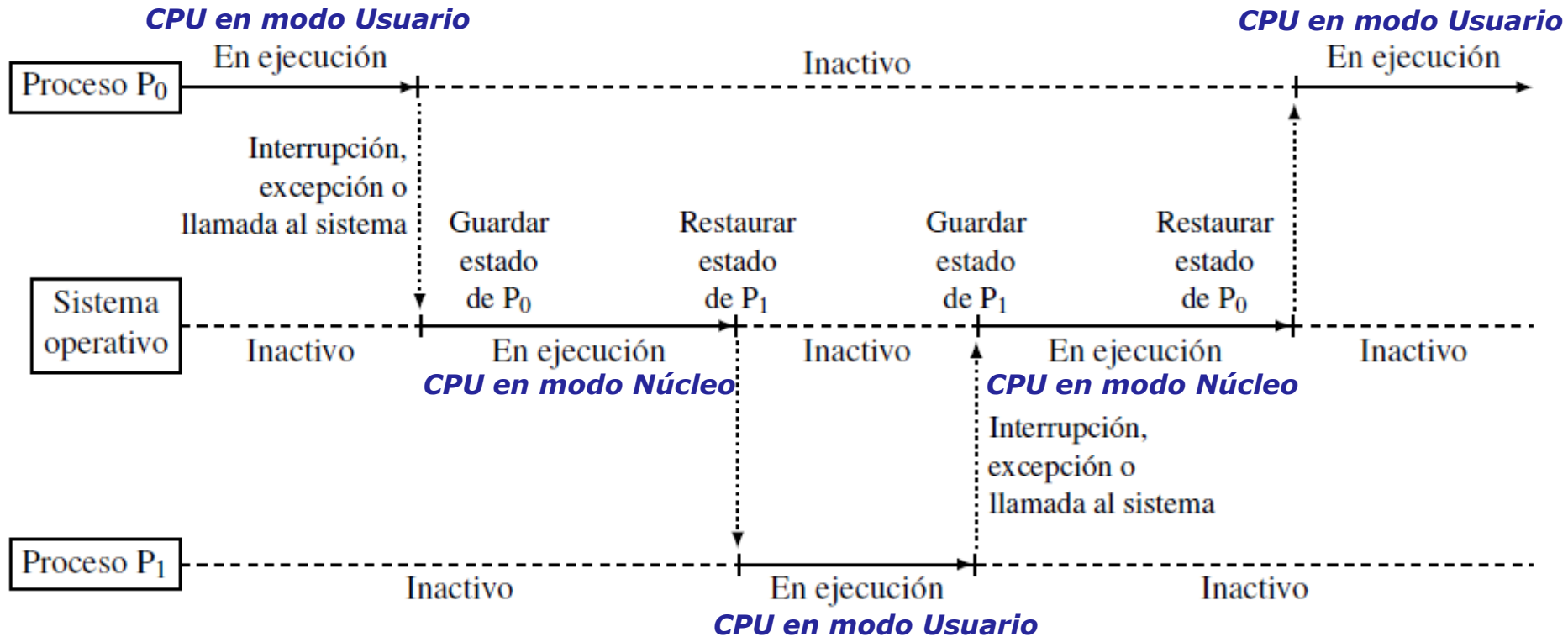
# 2.1 Introducción

- **Concepto de proceso:**

- Proceso == Programa en ejecución
  - es la unidad para describir las tareas de cada usuario
- Recordad: programa == estático ; proceso == dinámico
- Un proceso necesita recursos: memoria, CPU, espacio en disco...
- Un mismo proceso puede comprender varios programas (como sucede en UNIX) y un programa puede dar lugar a varios procesos
- Cuando hay una sola CPU y esta pasa rápidamente de ejecutar un proceso a otro:
  - Sensación de que todos los procesos se ejecutan a la vez: Multitarea o Pseudoparalelismo (la CPU se reparte entre los procesos)
  - Cambio de proceso  $\neq$  Cambio de contexto (pasar de ejecutar código de un programa a código de otro) En concreto, un cambio de contexto no necesariamente implica un cambio de proceso.

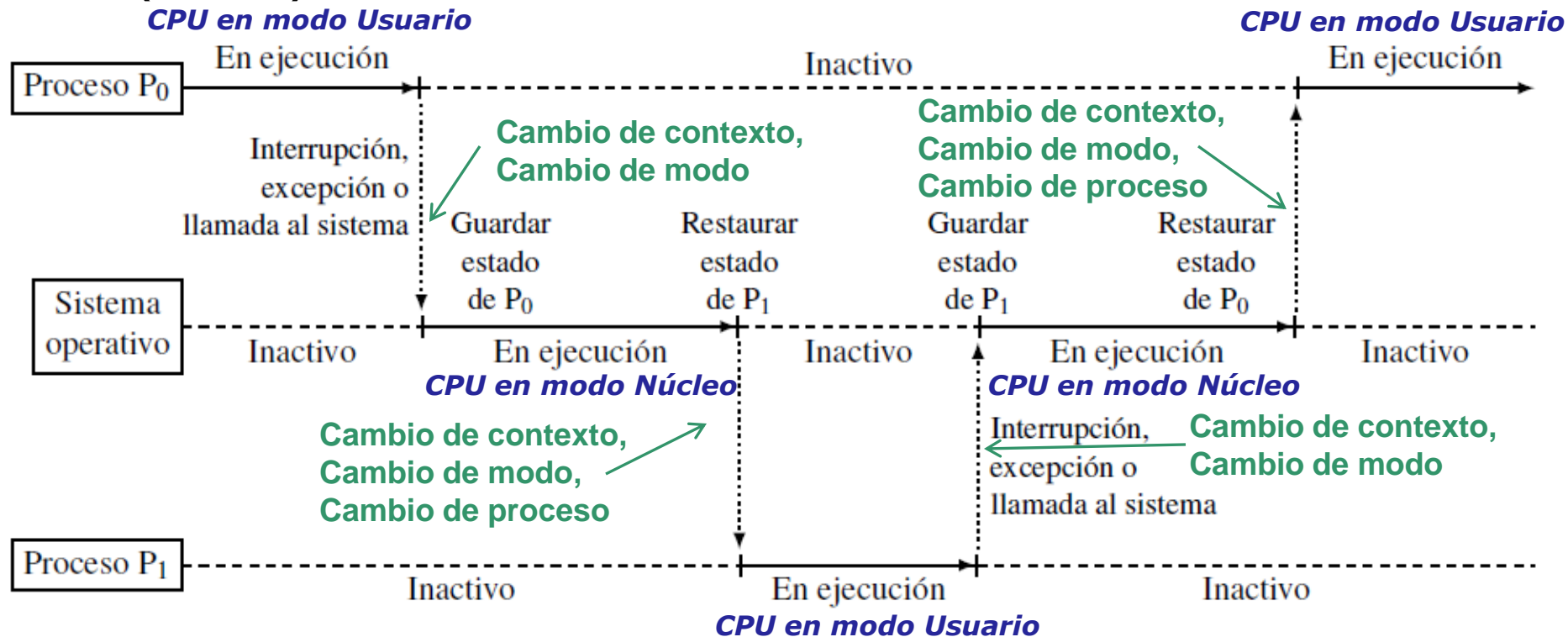
# 2.1 Introducción

- Cambio de proceso (1 CPU):



# 2.1 Introducción

- Cambio proceso UNIX vs Cambio contexto vs Cambio modo (1 CPU):



- Cambio de proceso implica uno o más cambios de contexto y de modo
- Cambio de modo implica cambio de contexto
- Cambio de contexto no implica siempre cambio de proceso
- Cambio de contexto no implica siempre cambio de modo

## 2.1 Introducción

- Permitir la ejecución concurrente de varios procesos complica la implementación del SO, pero tiene importantes ventajas:
  - Compartir recursos físicos (ej. CPU muy potente)
  - Compartir recursos lógicos (como una base de datos)
  - Acelerar la ejecución (sobre todo si tenemos varias CPUs)
  - Modularidad (facilidad de diseño)
  - Comodidad (permitir hacer varias cosas a la misma vez)
- Los procesos nunca deben diseñarse con hipótesis implícitas de tiempo
  - No se sabe con certeza cuándo se ejecutarán
- Cuando un proceso deja la CPU, ¿qué proceso pasa a usarla?
  - Tenemos más procesos que CPUs, por tanto necesitamos un **algoritmo de planificación (o planificador)** para decidir un proceso de entre todos los disponibles

# 2.1 Introducción

- Creación de procesos:
  - Si el SO soporta el concepto de proceso, requiere de mecanismos para su creación
  - ¿Cuándo se crean los procesos?
  - Inicio del sistema (creados por el núcleo al iniciarse el SO)
    - Procesos interactivos (usuario)
    - Procesos que están en segundo plano y se activan cuando sucede un evento. **Demonios** (*daemons* en UNIX)
  - Mediante llamadas al sistema
    - Llamada al sistema explícita para crear procesos
      - UNIX `fork+exec`, Win32 `CreateProcess`
    - El usuario inicia un proceso
      - Ya sea en modo gráfico o en algún *shell*
    - Inicio de un trabajo por lotes
      - Normalmente en sistemas de *mainframe*



## 2.1 Introducción

- Creación de procesos (llamadas al sistema)
  - Por ejemplo: en Unix se usa `fork()` para crear un proceso
    - Crea una copia idéntica (proceso hijo) del proceso llamador (proceso padre): mismo código, datos, mismos valores de pila, registros de la CPU...
    - En el hijo devuelve 0 y en el padre devuelve el *identificador de proceso* (PID) del hijo (permite diferenciar a padre e hijo)
    - El hijo podría usar llamadas al sistema tipo `exec()` y sustituir código y datos por los de otro programa (argumento) y ejecuta desde el principio (mantiene ficheros abiertos y tratamiento de las señales, salvo capturadas)
  - Jerarquía o árbol de procesos: proceso padre puede crear más procesos, procesos hijos también
    - En algunos SSOO como Linux, el kernel construye y gestiona esa jerarquía, permitiendo solo determinadas operaciones entre ellos (ej. un padre puede finalizar su procesos hijos)
    - En otros SSOO como Windows, el kernel no lo gestiona sino que un proceso que crea otro recibe un handler que le permite controlarlo, pero también pasárselo a otro proceso, lo que invalida el concepto de jerarquía


## 2.1 Introducción

```
7  int main(void)
   {
9   pid_t pidhijo;
   char orden[80];
11
   while(1) {
13     printf("$ ");
       fgets(orden, 80, stdin);
15     orden[strlen(orden)-1]='\0'; // Eliminamos \n.

17     pidhijo = fork();
       if (pidhijo == 0 ) {
19         execlp(orden, orden, NULL);
           printf("La orden %s falló\n", orden);
21         exit(1);
       }
23     wait(NULL);
   }
25
   return 0;
27 }
```

# 2.1 Introducción

## Proceso padre (PID=1450)


```
7  int main(void)
   {
9   pid_t pidhijo;
   char orden[80];
11
   while(1) {
13     printf("$ ");
    fgets(orden, 80, stdin);
15     orden[strlen(orden)-1]='\0'; // Eliminamos \n.

17     pidhijo = fork();
     if (pidhijo == 0 ) {
19         execlp(orden, orden, NULL);
         printf("La orden %s falló\n", orden);
21         exit(1);
     }
23     wait(NULL);
   }
25
   return 0;
27 }
```

\$ \_

# 2.1 Introducción

## Proceso padre (PID=1450)

```
7  int main(void)
   {
9    pid_t pidhijo;
    char orden[80];
11
    while(1) {
13     printf("$ ");
     fgets(orden, 80, stdin);
15     orden[strlen(orden)-1]='\0'; // Eliminamos \n.

17     pidhijo = fork();
    if (pidhijo == 0 ) {
19         execlp(orden, orden, NULL);
        printf("La orden %s falló\n", orden);
21         exit(1);
    }
23     wait(NULL);
    }
25
    return 0;
27 }
```

\$ ls

# 2.1 Introducción

## Proceso padre (PID=1450)

```
7  int main(void)
   {
9    pid_t pidhijo;
    char orden[80];
11
    while(1) {
13        printf("$ ");
        fgets(orden, 80, stdin);
15        orden[strlen(orden)-1]='\0'; // Eliminamos
17        pidhijo = fork();
        if (pidhijo == 0 ) {
19            execlp(orden, orden, NULL);
            printf("La orden %s falló\n", orden);
21            exit(1);
        }
23        wait(NULL);
    }
25
    return 0;
27 }
```

## Proceso hijo (PID=2270)

```
7  int main(void)
   {
9    pid_t pidhijo;
    char orden[80];
11
    while(1) {
13        printf("$ ");
        fgets(orden, 80, stdin);
15        orden[strlen(orden)-1]='\0'; // Eliminamos
17        pidhijo = fork();
        if (pidhijo == 0 ) {
19            execlp(orden, orden, NULL);
            printf("La orden %s falló\n", orden);
21            exit(1);
        }
23        wait(NULL);
    }
25
    return 0;
27 }
```

\$ ls

# 2.1 Introducción

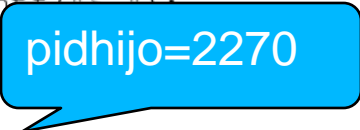
## Proceso padre (PID=1450)

```
7  int main(void)
   {
9   pid_t pidhijo;
   char orden[80];

11  while(1) {
13   printf("C: ");
   fgetc(stdin);
15   orden = " "; // Eliminamos

17  pidhijo = fork();
   if (pidhijo == 0 ) {
19   execlp(orden, orden, NULL);
   printf("La orden %s falló\n", orden);
21   exit(1);
   }
23   wait(NULL);
   }

25  return 0;
27 }
```



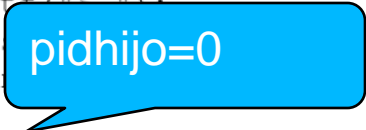
## Proceso hijo (PID=2270)

```
7  int main(void)
   {
9   pid_t pidhijo;
   char orden[80];

11  while(1) {
13   printf("C: ");
   fgetc(stdin);
15   orden = " "; // Eliminamos

17  pidhijo = fork();
   if (pidhijo == 0 ) {
19   execlp(orden, orden, NULL);
   printf("La orden %s falló\n", orden);
21   exit(1);
   }
23   wait(NULL);
   }

25  return 0;
27 }
```




\$ ls

# 2.1 Introducción

## Proceso padre (PID=1450)

```
7  int main(void)
   {
9   pid_t pidhijo;
   char orden[80];

11
   while(1) {
13     printf("$ ");
     fgets(orden, 80, stdin);
15     orden[strlen(orden)-1]='\0'; // Eliminamos

17     pidhijo = fork();
     if (pidhijo == 0 ) {
19       execlp(orden, orden, NULL);
       printf("La orden %s falló\n", orden);
21       exit(1);
     }
2  wait(NULL);
   }


25
   return 0;

27 }
```

## Proceso hijo (PID=2270)

```
7  int main(void)
   {
9   pid_t pidhijo;
   char orden[80];

11
   while(1) {
13     printf("$ ");
     fgets(orden, 80, stdin);
15     orden[strlen(orden)-1]='\0'; // Eliminamos

17     pidhijo = fork();
     if (pidhijo == 0 ) {
19  execlp(orden, orden, NULL);
       printf("La orden %s falló\n", orden);
21       exit(1);
     }
23     wait(NULL);
   }

25
   return 0;

27 }
```

\$ ls

# 2.1 Introducción

## Proceso padre (PID=1450)

```
7  int main(void)
   {
9   pid_t pidhijo;
   char orden[80];
11
   while(1) {
13     printf("$ ");
        fgets(orden, 80, stdin);
15     orden[strlen(orden)-1]='\0'; // Eliminamos
17     pidhijo = fork();
        if (pidhijo == 0 ) {
19         execlp(orden, orden, NULL);
        printf("La orden %s falló\n", orden);
21         exit(1);
        }
23     wait(NULL);
25
        return 0;
27 }
```

## Proceso hijo (PID=2270)

Código de ls

...

...

...

...

...

...

...


exit

\$ ls



# 2.1 Introducción

## Proceso padre (PID=1450)

```
7  int main(void)
   {
9   pid_t pidhijo;
   char orden[80];
11
   while(1) {
13     printf("$ ");
    fgets(orden, 80, stdin);
15     orden[strlen(orden)-1]='\0'; // Eliminamos \n.

17     pidhijo = fork();
     if (pidhijo == 0 ) {
19         execlp(orden, orden, NULL);
         printf("La orden %s falló\n", orden);
21         exit(1);
     }
23     wait(NULL);
   }
25
   return 0;
27 }
```

\$


# 2.1 Introducción

## Proceso padre (PID=1450)

```
7  int main(void)
   {
9   pid_t pidhijo;
   char orden[80];

11  while(1) {
13   printf("$ ");
   fgets(orden, 80, stdin);
15   orden[strlen(orden)-1]='\0'; // Eliminamos el salto de línea

17   pidhijo = fork();
   if (pidhijo == 0 ) {
19       execlp(orden, orden, NULL);
       printf("La orden %s falló\n", orden);
21       exit(1);
   }
23   wait(NULL);
25
   return 0;
27 }
```




## Proceso hijo (PID=2270)

```
7  int main(void)
   {
9   pid_t pidhijo;
   char orden[80];

11  while(1) {
13   printf("$ ");
   fgets(orden, 80, stdin);
15   orden[strlen(orden)-1]='\0'; // Eliminamos el salto de línea

17   pidhijo = fork();
   if (pidhijo == 0 ) {
19       execlp(orden, orden, NULL);
       printf("La orden %s falló\n", orden);
21       exit(1);
   }
23   wait(NULL);
25
   return 0;
27 }
```



Si execlp falla...

\$ ls

## 2.1 Introducción

- Creación de un proceso en UNIX:
  - Uso de `fork+exec`

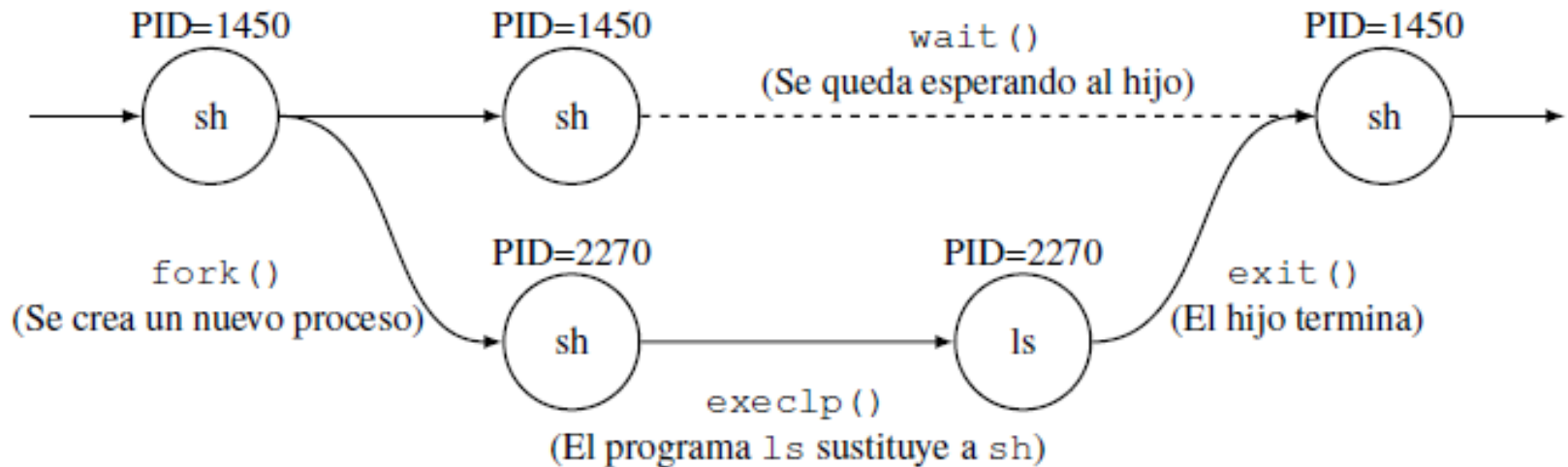


Figura 2.2: Uso conjunto de las llamadas al sistema `fork()` y `exec()` para ejecutar la orden `ls` en el programa 2.1.

# 2.1 Introducción

- Creación de un proceso en UNIX:
  - **exec.** Importante observar:
    - No crea un nuevo proceso
    - Cambia código y datos dentro del proceso
    - Conserva:
      - Ficheros abiertos
      - Tratamiento de señales ignoradas (señales = “interrupciones” que se pueden enviar a un proceso)
      - PID
  - Padre e hijo tiene espacios de direcciones distintos (en UNIX el espacio de direcciones del padre se copia inicialmente al del hijo) → No se comparte la memoria

## 2.1 Introducción

- Destrucción de procesos:
  - También se precisa de un mecanismo para terminar procesos
  - ¿Cuándo puede terminar un proceso?
  - Terminación normal (voluntaria): una vez concluye su trabajo
    - Se ejecuta llamada al sistema (UNIX `exit` / Win32 `ExitProcess`)
  - Terminación por error (voluntaria)
    - También usando esas llamadas al sistema, pero devolviendo un código de error indicando el fallo
  - Error fatal (involuntaria)
    - Instrucción ilegal, violación de memoria, ...
  - Terminado por otro proceso (involuntaria)
    - Llamada al sistema para terminar procesos (UNIX `kill`, Win32 `TerminateProcess`)

2.1 Introducción

2.2 Estados de un proceso

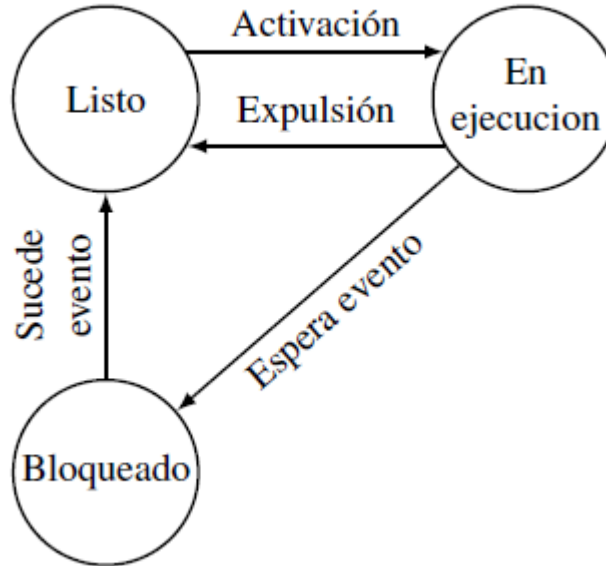
2.3 Implementación de procesos

2.4 Hilos (*threads*)

2.5 Planificación de procesos

## 2.2 Estados de un proceso

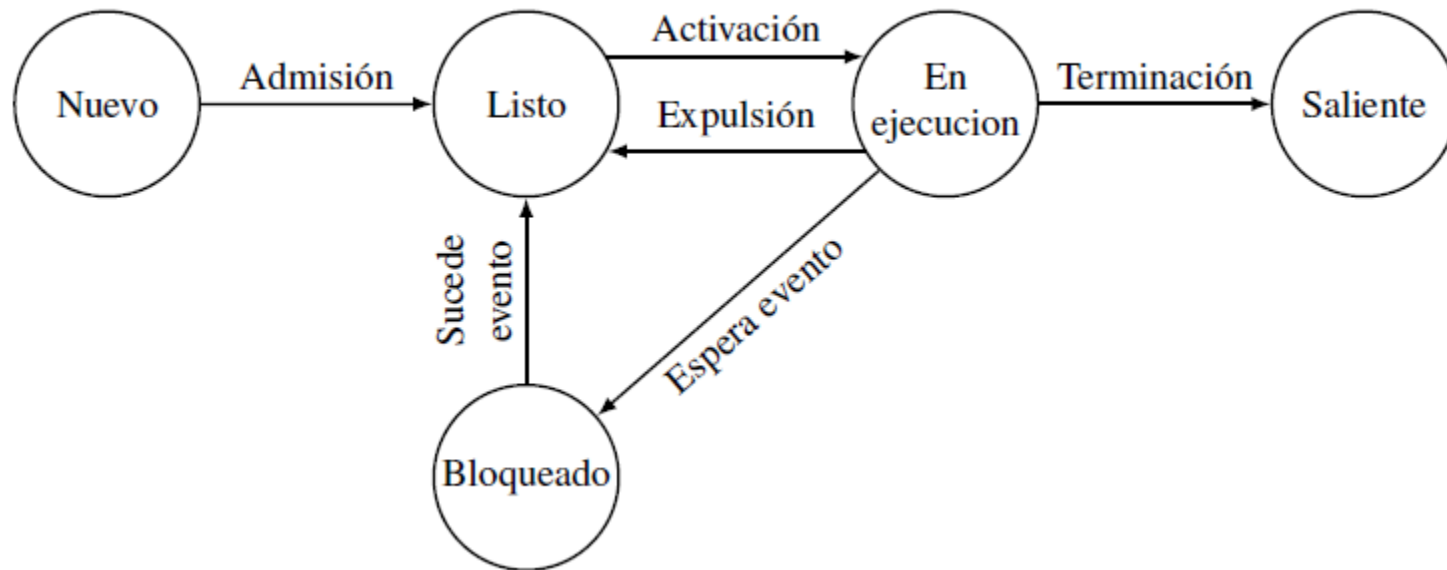
- Estados de un proceso:
  - **En ejecución:** tiene la CPU
  - **Listo:** detenido, otro tiene la CPU
  - **Bloqueado:** no se puede ejecutar, espera algún evento



- 1 CPU → 1 proceso *En ejecución*, varios *Listo* o *Bloqueado*

## 2.2 Estados de un proceso

- Hacen falta más estados para reflejar situaciones adicionales que podrían darse:
  - **Nuevo.** Se acaba de crear y aunque tiene el PCB, no ha sido cargado en memoria
  - **Saliente.** Se ha quitado del grupo de procesos ejecutables porque ha terminado (puede haber sido abortado por alguna razón)



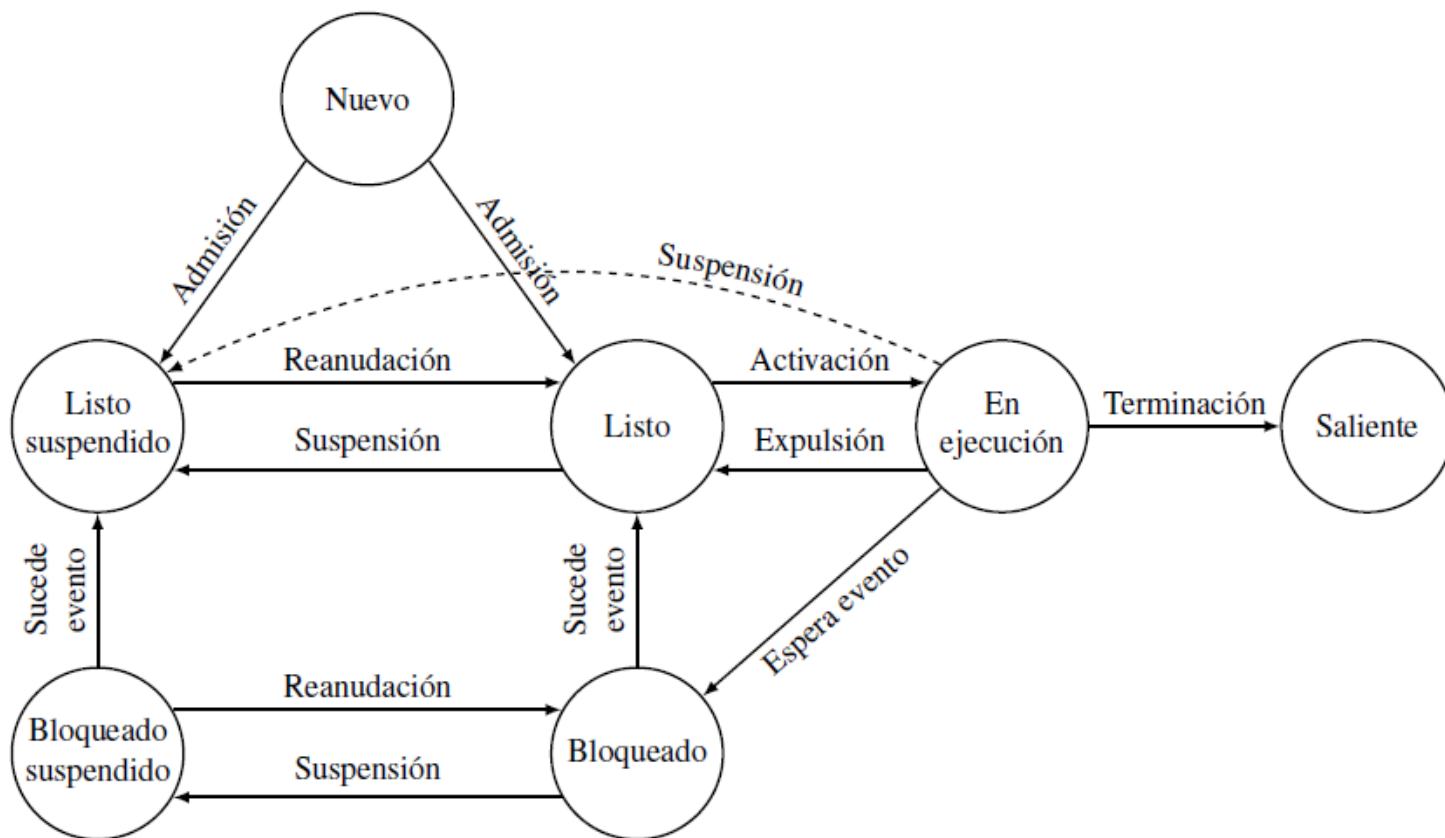
- **Listo/Bloqueado ⇒ Saliente.** No se muestra. Por ejemplo, un proceso padre puede terminar la ejecución de un hijo alguna razón



## 2.2 Estados de un proceso

- Además, se puede incluir la posibilidad de **suspender** y **reanudar** procesos:
  - Aliviar la carga temporalmente
  - Depuración
- Los procesos suspendidos se guardan en disco
- Estados adicionales:
  - **BloqueadoSuspendido** : está en disco, esperando un evento
  - **ListoSuspendido** : está en disco, podría pasar a ejecución

## 2.2 Estados de un proceso



- **Listo**. Está en memoria principal disponible para ejecución
- **Bloqueado**. Está en memoria principal esperando un evento
- **Listo suspendido**. Está en memoria secundaria disponible para ejecutarse
- **Bloqueado suspendido**. Está en memoria secundaria esperando un evento

## 2.2 Estados de un proceso

- **Transiciones:**

- **Bloqueado**  $\Rightarrow$  **Bloqueado suspendido**. Un proceso bloqueado se pasa a disco para liberar memoria, que puede ser usada por procesos otros listos
- **Bloqueado suspendido**  $\Rightarrow$  **Listo suspendido**. Llega el evento
- **Listo suspendido**  $\Rightarrow$  **Listo**. Si no hay procesos listos, necesitaremos traer uno para ejecutarlo. También puede que un proceso listo suspendido tenga la mayor prioridad, o que haya quedado memoria libre
- **Listo**  $\Rightarrow$  **Listo suspendido**. Normalmente se suspenden procesos bloqueados, pero se puede preferir suspender un proceso listo de baja prioridad ante uno bloqueado de alta prioridad. Se busca obtener más memoria libre
- **Nuevo**  $\Rightarrow$  **Listo suspendido/Listo**. Creamos el proceso en disco cuando no hay memoria disponible y no tiene prioridad como para expulsar a otro
- **Bloqueado suspendido**  $\Rightarrow$  **Bloqueado**. Un proceso termina liberando memoria, uno de los bloqueados suspendidos tiene la mayor prioridad y el SO sospecha que el evento que espera sucederá en breve (heurística)
- **Ejecutando**  $\Rightarrow$  **Listo suspendido**. Un proceso de mayor prioridad despierta de bloqueado suspendido
- **Cualquier estado**  $\Rightarrow$  **Saliente**

2.1 Introducción

2.2 Estados de un proceso

2.3 Implementación de procesos

2.4 Hilos (*threads*)

2.5 Planificación de procesos

## 2.3 Implementación de procesos

- Para implementar el modelo de procesos, el SO debe conocer qué procesos hay y su estado
- Para administrar los procesos se usa una **tabla de procesos**, con una entrada por cada uno de ellos:
  - Cada entrada se llama **PCB** (*Process Control Block*): guarda la información relacionada con el proceso

Administración de procesos	Administración de memoria	Administración de ficheros
Registros	Dirección del segmento de texto <sup>9</sup>	Directorio raíz
Contador del programa	Dirección del segmento de datos	Directorio de trabajo
Palabra de estado del programa	Dirección del segmento BSS	Descriptores de fichero
Puntero de pila	Dirección del segmento de pila	Identificación de usuario
Estado del proceso		Identificación de grupo
Prioridad		
Parámetros de planificación		
Identificador de proceso		
Proceso padre		
Hora de inicio del proceso		
Tiempo utilizado de CPU		

## 2.3 Implementación de procesos

- **Creación de un proceso:**
  1. Asignar un ID único al proceso (PID en UNIX)
  2. Insertarlo en la tabla de procesos, una vez inicializado su PCB
  3. Determinar prioridad inicial e incluir al proceso en los algoritmos de planificación
  4. Asignarle recursos iniciales
- **Creación de un proceso en UNIX (llamada `fork()` )**
  - Proceso padre realiza llamada `fork()`  $\Rightarrow$  interrupción  $\Rightarrow$  paso al núcleo
  - Núcleo busca entrada tabla de procesos para hijo. Asigna PID
  - Se copia la información de la entrada de la tabla procesos del padre en la entrada hijo
  - Asignación memoria segmentos datos y pila hijo, y se copia el contenido de los segmentos de los del padre
  - Compartición segmento código
  - Incremento contadores ficheros abiertos padre
  - Se le asigna al hijo el estado LISTO

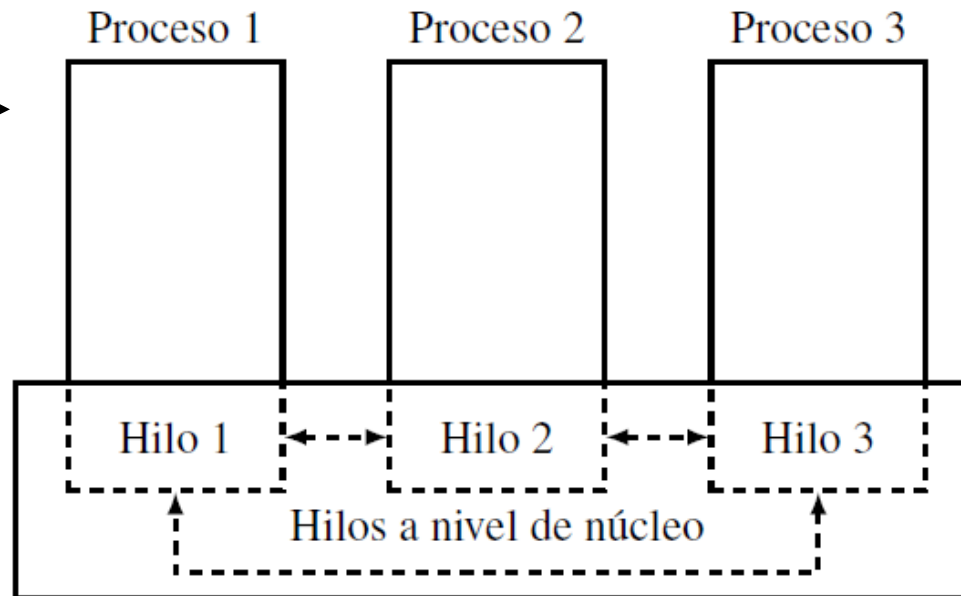
## 2.3 Implementación de procesos

- **Estructura de un proceso**

- Proceso = programa en ejecución (*modo usuario del procesador*)
- Llamadas al sistema para usar los servicios del SO (*modo núcleo*)
  - ¿Seguimos dentro del mismo proceso o se cambia de proceso?
- **En UNIX:**
  - El proceso está compuesto por la **parte usuario** (lo que el usuario implementa) y la **parte de núcleo** (las rutinas del núcleo que utiliza)

**Parte usuario:** →  
Distinta para cada proceso

**Parte núcleo:** →  
Común para todos



## 2.3 Implementación de procesos

- **Estructura de un proceso**

- Cuando un proceso hace llamada al sistema pasa a su parte núcleo (no sale del proceso, pero se pone a ejecutar código del SO)
- Varios procesos podrían hacer llamadas al sistema a la vez:
  - La parte del núcleo de cada proceso tiene su propio PC y pila, de forma que no hay interferencias
- Puede que un proceso se bloquee dentro del núcleo:
  - Entonces se cede la CPU a otro proceso. Cuando se desbloquee seguirá en el núcleo hasta finalizar procesamiento llamada



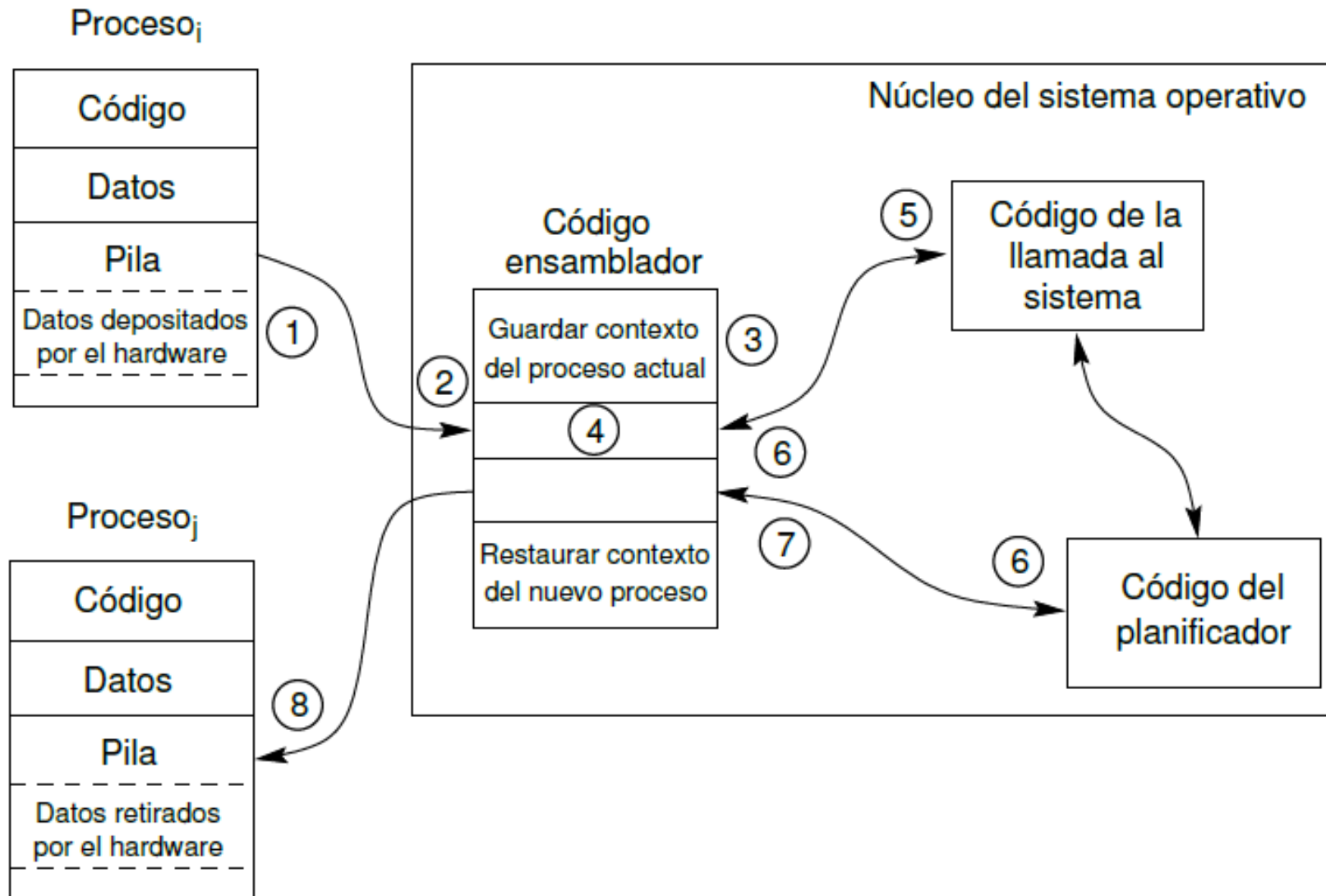
## 2.3 Implementación de procesos

- **Cambio de proceso**
  - Los realiza el kernel del SO cuando toma el control de la CPU
  - Eventos que lo motivan:
    - Interrupción hardware:
      - de reloj: Proceso en *ejecución* agota su tiempo → pasa a *listo*
      - de E/S: Evento por el que espera algún proceso → pasa a *listo*
    - Interrupción software (Llamada al sistema) → podría pasar a *bloqueado*
    - Excepción: por ejemplo
      - Fallo de página: proceso pasa a *bloqueado*
      - División por cero: irreversible, el proceso pasa a *saliente*
- No es lo mismo un cambio de proceso que un cambio de contexto
  - **Cambiar de contexto:** guarda registros, cambia pila,...
  - **Cambiar de proceso:** actualizar PCB, mover entre colas, seleccionar nuevo proceso, cambiar estructuras de gestión de memoria, activarlo...

## 2.3 Implementación de procesos

- **Mecanismo de cambio de proceso** (Suponemos una llamada al sistema)
  1. **Hw**: guarda en pila el contador de programa (PC) del llamador (pila que el proceso usa en modo usuario)
  2. **Hw**: pasa a modo núcleo y pone nuevo PC según vector interrupción (**Rutina ASM** del kernel asociada con las llamadas al sistema)
  3. **Rutina ASM**: guarda contexto de llamador en su PCB
  4. **Rutina ASM**: configurar nueva pila que usará el kernel del SO mientras atiende la llamada al sistema (el kernel es un programa y necesita pila)
  5. **Rutina ASM** comprueba llamada al sistema y llama a **Rutina en C** que implementa la llamada
    - 5.1. **Rutina en C** ejecuta código de la llamada (el proceso se podría bloquear, llamar al planificador y ceder la CPU a otro proceso)
  6. **Rutina ASM**: tras llamada ver si hay que llamar al planificador
    - 6bis. **Planificador**: código C que decide qué proceso se ejecutará a continuación (mirando tabla de procesos) y se reconfigura el sistema para el cambio de proceso (**Despachador**: parte del SO que entrega control CPU al nuevo proceso)
  7. **Rutina ASM**: restaura el contexto del proceso a ejecutar y la pila que el proceso que recibe la CPU va a usar en modo usuario
  8. **Hw**: Cambia a modo usuario, desapila dirección de siguiente instrucción (del nuevo proceso)  $\Rightarrow$  nuevo PC

## 2.3 Implementación de procesos



2.1 Introducción

2.2 Estados de un proceso

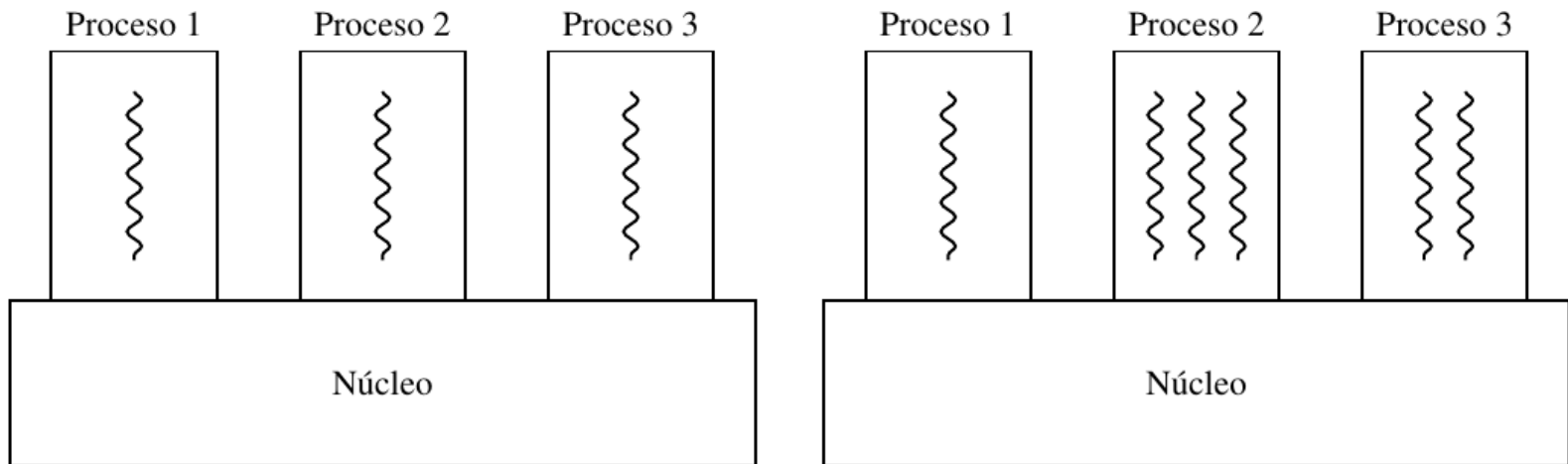
2.3 Implementación de procesos

2.4 Hilos (*threads*)

2.5 Planificación de procesos

## 2.4 Hilos (*threads*)

- Hasta ahora, un proceso es, a la vez:
  - **Unidad de asignación de recursos**
  - **Unidad de planificación y ejecución**
- Los SSOO modernos separan estas dos características independientes:
  - **Hilo** es la unidad de planificación
  - **Proceso** es la unidad de asignación de recursos
- Un proceso puede tener **varios hilos**



## 2.4 Hilos (*threads*)

```
#include <pthread.h>
2  #include <stdio.h>
   #include <unistd.h>
4  #include <sys/types.h>

6  #include <sys/syscall.h>

8  void * hilo_main (void * arg)
   {
10     fprintf (stderr, "\tEl PID del hilo hijo es: %d\n", (int)
        getpid());
        fprintf (stderr, "\tEl TID del hilo hijo es: %d\n", (int)
            syscall(SYS_gettid));

12     /* Bucle infinito del hilo hijo. */
14     while (1);

16     return NULL;
   }

18
```

## 2.4 Hilos (*threads*)

```
int main (void)
20 {
    pthread_t thread;

22     fprintf (stderr, "\nEl PID del hilo principal es: %d\n", (
        int) getpid());
24     fprintf (stderr, "El TID del hilo principal es: %d\n\n", (
        int) syscall(SYS_gettid));

26     /* Creamos un hilo cuya ejecución comienza en la función
        hilo_main. */
    pthread_create(&thread, NULL, &hilo_main, NULL);

28     /* Bucle infinito del hilo padre. */
30     while (1);

32     return 0;
}
```

## 2.4 Hilos (*threads*)

```
[alumno@localhost ~]$ gcc creahilo.c -o creahilo -lpthread  
[alumno@localhost ~]$ ./creahilo
```

El PID del hilo principal es: 8115

El TID del hilo principal es: 8115

El PID del hilo hijo es: 8115

El TID del hilo hijo es: 8116



## 2.4 Hilos (*threads*)

- Elementos asociados al proceso:
  - Asignación de memoria
  - Variables globales
  - Procesos hijos
  - Alarmas
  - Señales
  - Ficheros abiertos
  - Información contable
- Elementos asociados a cada hilo (proceso con varios hilos):
  - Estado (ejecución, listo, ...)
  - Contexto (valores de los registros de la CPU y PC)
  - Pila de ejecución

## 2.4 Hilos (*threads*)

- Los hilos de un proceso no son tan independientes como procesos distintos:
  - Comparten memoria: todos los hilos tienen el mismo espacio de memoria (comparten las mismas variables globales)
    - A través de mecanismos de sincronización (mutex y/o semáforos) se organiza el acceso a la memoria compartida
  - Un hilo puede acceder a la pila de otro hilo (no hay protección entre los hilos)
    - No existe protección (ni se necesita) entre los hilos de un mismo proceso
  - Los recursos del proceso son compartidos entre los hilos (ficheros abiertos, señales, procesos hijos, ...)
  - Aunque la planificación se realiza basándose en hilos, la terminación y suspensión de un proceso afecta a todos los hilos

## 2.4 Hilos (*threads*)

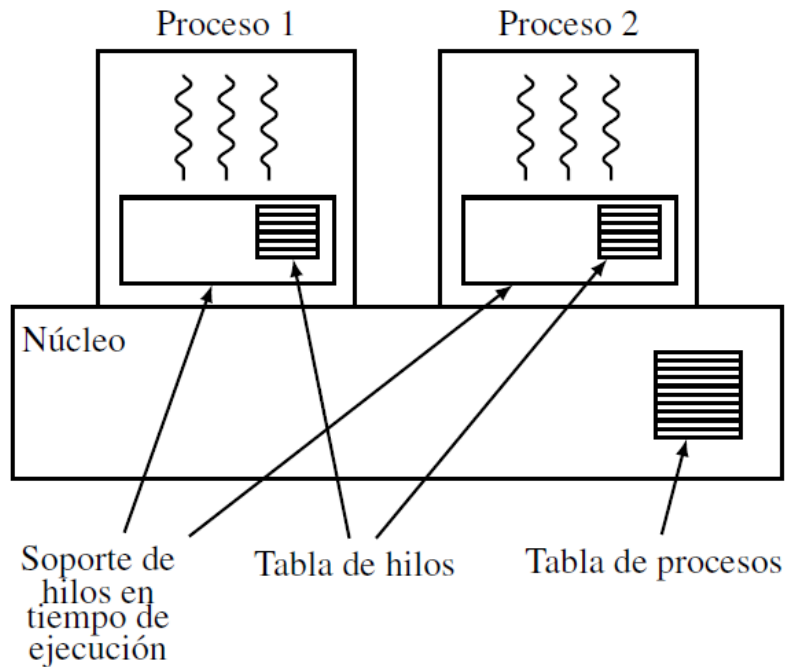
- **¿Para qué quiero tener varios hilos por proceso?**
  - Comparten memoria → Comunicación entre hilos sin intervención del núcleo (más rápida)
  - Descomponer una aplicación en varios hilos, cada uno a cargo de una función diferente, puede facilitar su construcción
  - Permiten solapar E/S y cómputo dentro de un mismo proceso (un hijo maneja la E/S y es quien se bloquea mientras otro calcula)
  - Con varias CPUs se puede explotar paralelismo real en un proceso
  - Menos tiempo para crear y destruir hilos, y conmutar entre ellos
  - ¿Cómo harías sin hilos...?
    - Procesador de textos: guardar automáticamente cambios cada cierto tiempo
    - Servidor web: Hilo despachador (principal) que va creando nuevos hilos conforme llegan peticiones (ejecutores). Los ejecutores se bloquean, el principal no
    - Aplicación gráfica que interactúa directamente con el usuario: un hilo puede llevar menús, otro ejecuta órdenes y otro actualiza la ventana

## 2.4 Hilos (*threads*)

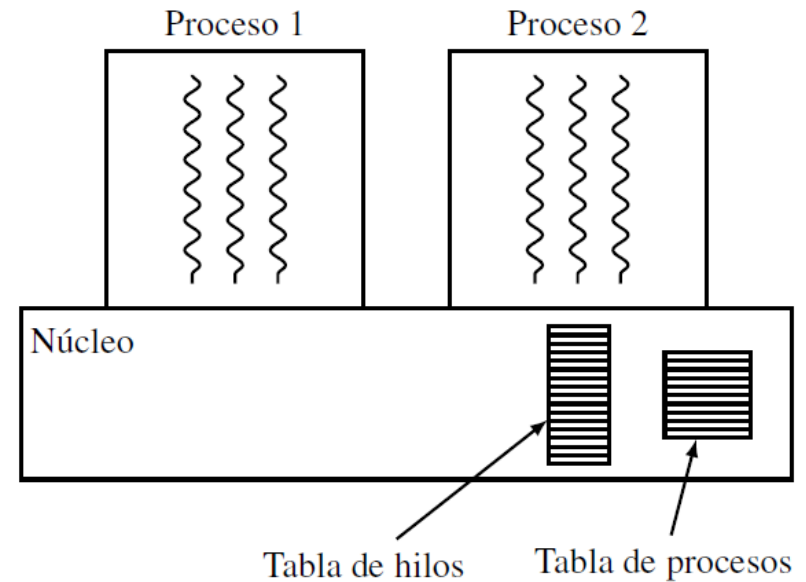
- **Soporte de los hilos**

- Los hilos pueden ser soportados directamente por el **núcleo** (llamadas al sistema) o a través de llamadas a librería a nivel de **usuario**

### HILOS EN ESPACIO USUARIO



### HILOS EN NÚCLEO



## 2.4 Hilos (*threads*)

- **Hilos implementados en modo usuario**

- **Ventajas**

- El núcleo del SO no sabe que existen: portabilidad
    - Tabla de hilos privada en el proceso para cambios de contexto: cada proceso puede tener su algoritmo de planificación interno
    - Cambio de contexto mucho más rápido entre hilos (no se pasa al kernel)

- **Inconvenientes**

- Llamadas al sistema bloqueantes bloquean al proceso completo  
**Solución:** funciones no bloqueantes, pero más difícil de programar
    - Fallos de página bloquean al proceso completo
    - Tienen que ceder la CPU entre ellos, lo que los limita a conmutar dentro del mismo proceso
    - No podemos explotar paralelismo real en arquitecturas con varias CPUs

## 2.4 Hilos (*threads*)

- **Hilos implementados en el núcleo**

- **Ventajas**

- El núcleo mantiene la tabla de hilos y reparte la CPU entre hilos
    - Las llamadas bloqueantes no introducen problemas: si un hilo se bloquea, otro hilo usa la CPU
    - Los fallos de página tampoco suponen un problema: se pasa a otro hilo
    - Al bloquearse un hilo, el núcleo puede conmutar a otro hilo de otro proceso
    - Con varias CPUs, podríamos ejecutar los hilos de un proceso a la vez usando varias CPUs (explotar paralelismo real)

- **Inconvenientes**

- Sincronización entre hilos con llamadas al sistema → más costosas
    - La creación y destrucción de hilos es más costoso → Reutilización de hilos

2.1 Introducción

2.2 Estados de un proceso

2.3 Implementación de procesos

2.4 Hilos (*threads*)

2.5 Planificación de procesos

## 2.5.1 Planificadores

- En un momento dado pueden haber varios procesos en estado listo:
  - La parte del SO que decide qué proceso obtiene la CPU es el **planificador** siguiendo un **algoritmo de planificación**
  - El **despachador** es la parte del SO que materializa un cambio de proceso (reiniciar el proceso adecuado en la posición en que quedó, cambiar contexto y modo)
  - **Metas generales de la planificación de procesos:**
    - Conseguir **equidad** en el uso de la CPU → Cada proceso reciba su “parte” correspondiente de uso de la CPU
    - **Eficacia** → %uso de CPU ejecutando procesos de usuario  $\approx 100\%$
    - Maximizar el **rendimiento o productividad** → nº de tareas procesadas por unidad de tiempo
    - Minimizar **tiempo de respuesta** (usuarios interactivos) o el **tiempo de retorno** (usuarios de trabajos por lotes)
    - Minimizar **tiempo de espera** en la cola de procesos listos



## 2.5.1 Planificadores

- **Planificación:**

- **Apropiativa:** Un proceso *en ejecución* puede ser expulsado por el SO para entregar la CPU a otro proceso
- **No Apropiativa:** El proceso se ejecuta hasta que cede el control al SO (acaba o se bloquea)

- **Los procesos pueden ser:**

- **Por lotes:** No hay usuario interactuando, por tanto se puede usar una planificación no apropiativa o apropiativa con largos periodos de CPU
- **Interactivos:** Cuando hay usuarios interactuando con el sistema, tanto en un teclado como vía red (por ejemplo en servidores). Necesitan una planificación apropiativa

## 2.5.1 Planificadores

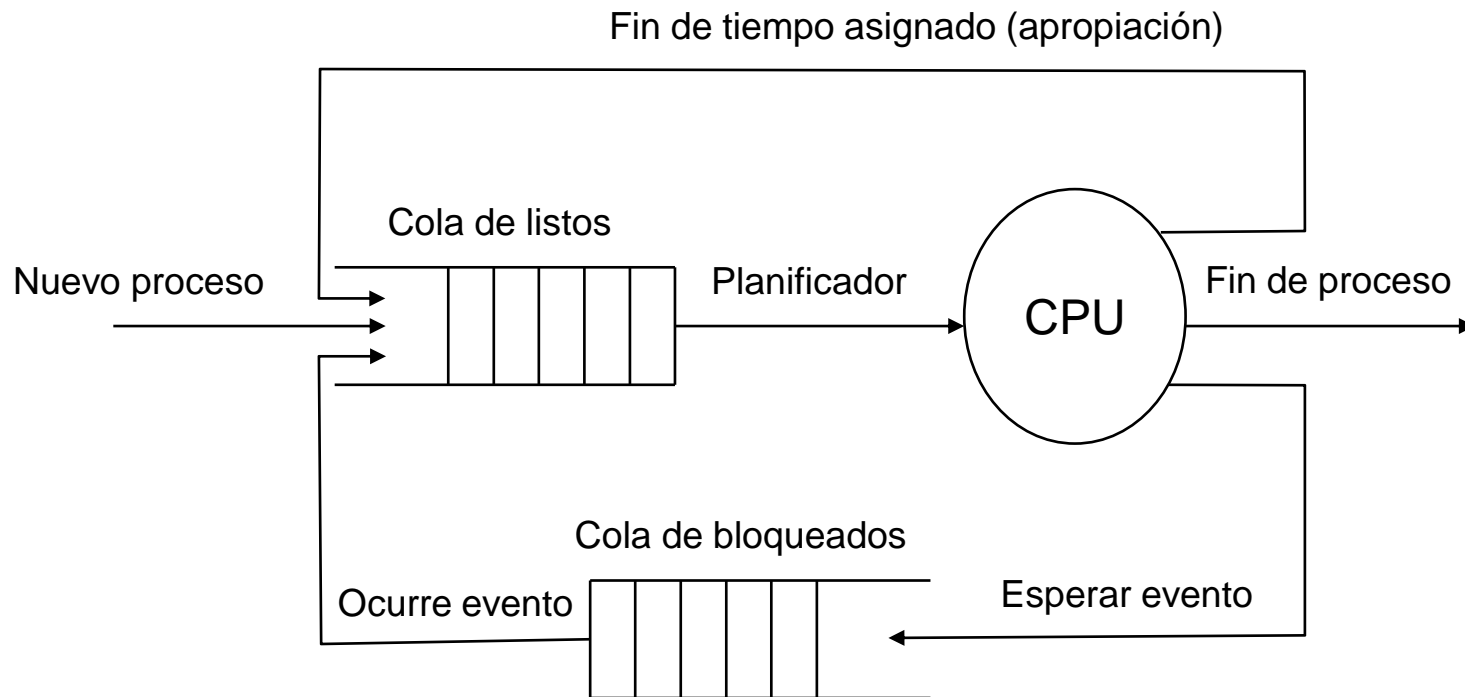
- Las necesidades de los procesos son muy distintas:
  - Aunque siempre siguen el patrón:
    - Se alternan Ráfaga de CPU y Ráfaga de E/S

Ráfaga CPU → Ráfaga E/S → Ráfaga CPU → Ráfaga E/S → Ráfaga CPU...

- La duración de las ráfagas **no se sabe a priori**, aunque se puede inferir a partir de la historia de la ejecución del proceso
- Esta alternancia de ráfagas es la que hace que tenga sentido la multiprogramación: mientras procesos esperan, otro usa la CPU
- **Procesos limitados por CPU**
  - Hacen mucho de la CPU para cálculo y muy poca E/S
    - P.ej. Resolver un sistema de ecuaciones enorme y mostrar la solución
- **Procesos limitados por E/S**
  - Realizan procesos que requieren de poco cálculo y mucha E/S
    - P.ej. Un shell, un comprobador de consistencia de un sistema de ficheros...

## 2.5.1 Esquema de funcionamiento

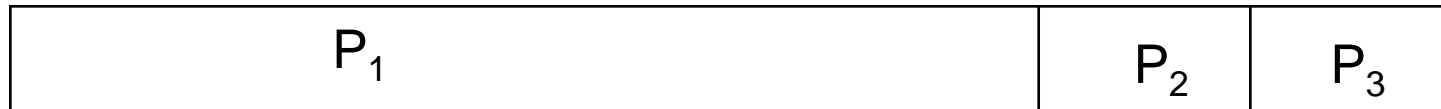
- **Representación gráfica de un sistema de planificación:**



## 2.5.2 Primero en llegar, primero servido

- FIFO o FCFS (*First Come, First Served*)
- No apropiativo
- **Para procesos por lotes.** Se le asigna la CPU al primer proceso que la requiere  $\Rightarrow$  puede implementarse con una cola FIFO
- El tiempo promedio de respuesta puede ser bastante largo

Proceso	Duración
P1	24
P2	3
P3	3



0

24

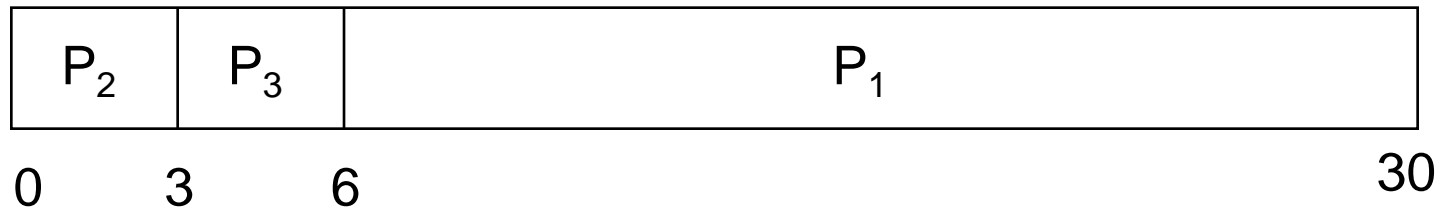
27

30

- Tiempo promedio de respuesta:  $(24+27+30)/3 = 27$  (cambiado a orden P2, P3, P1, el tiempo promedio de respuesta sería 13)
- **Problema:** Efecto convoy (los procesos que piden poca CPU y se bloquean rápidamente tienen que esperar a que todos los demás consuman su parte de CPU) → Desaprovechamiento de los recursos

## 2.5.3 El trabajo más corto primero (SJF)

- De entre todos los procesos listos, el de menor duración
- No apropiativo
- Para procesos por lotes (o de duración conocida)
- El caso anterior daría esta ordenación:



- Para procesos interactivos, podemos estimar el tiempo total de la ejecución (incluyendo E/S) o hacerlo para cada ráfaga de CPU
- Para que SJF sea óptimo es necesario disponer de todos los procesos de forma simultánea

	A	B	C	D	E
<b>Tiempo de ejecución</b>	<b>2</b>	<b>4</b>	<b>1</b>	<b>1</b>	<b>1</b>
<b>Tiempo de llegada</b>	<b>0</b>	<b>0</b>	<b>3</b>	<b>3</b>	<b>3</b>

Orden posible:            A, B, C, D, E

$$(2+6+4+5+6)/5 = 4.6$$

Orden mejor:            B, C, D, E, A

$$(4+2+3+4+9)/5 = 4.4$$

IMPOSIBLE:            C, D, E, A, B

i C, D y E llegan en el instante 3!

## 2.5.4 El menor tiempo restante primero (SRTF)

- Versión **apropiativa** de SJF:
  - El proceso que coge la CPU es aquél al que le queda menos para terminar
  - En versión interactiva sería aquél al que le queda menos para terminar su ráfaga de CPU actual
- En SRTF y en SJF un problema es saber a priori la duración del proceso → se usan **estimaciones**
- Para procesos interactivos necesitaremos una predicción de la duración de cada ráfaga de CPU

$$E_t = \alpha \cdot E_{t-1} + (1 - \alpha) T_{t-1} \quad \text{para } t = 2, 3, 4 \dots; \alpha \in [0, 1]$$

**ESTIMACIÓN ( $\alpha=0.5$ )**

**DURACIÓN REAL**

1ª Ráfaga CPU: →  $E_0$  (nos la dan) →  $T_0$

2ª Ráfaga CPU:  $E_1 = 0.5 \cdot E_0 + 0.5 \cdot T_0$  ←  $T_1$

3ª Ráfaga CPU:  $E_2 = 0.5 \cdot E_1 + 0.5 \cdot T_1$  ←  $T_2$

...

## 2.5.5 Planificación circular o Round-robin

- Uno de los más ampliamente utilizados
- Algoritmo **apropiado**, para sistemas interactivos
- Como FCFS pero a cada proceso se le asigna un **quantum** de tiempo. De esta manera el proceso:
  - Termina antes de consumir el quantum
  - Queda bloqueado (E/S o fallo de página)
  - Consume su tiempo
- El tiempo en coger CPU está acotado → Un proceso espera a lo sumo  **$(n-1) \times q$**  unidades de tiempo
- Es importante decidir la **longitud del quantum**
  - Quantum pequeño: Ineficiente →  
Tiempo de gestión > Tiempo ejecutando procesos
  - Quantum grande: Los procesos esperan mucho

## 2.5.6 Planificación por prioridad

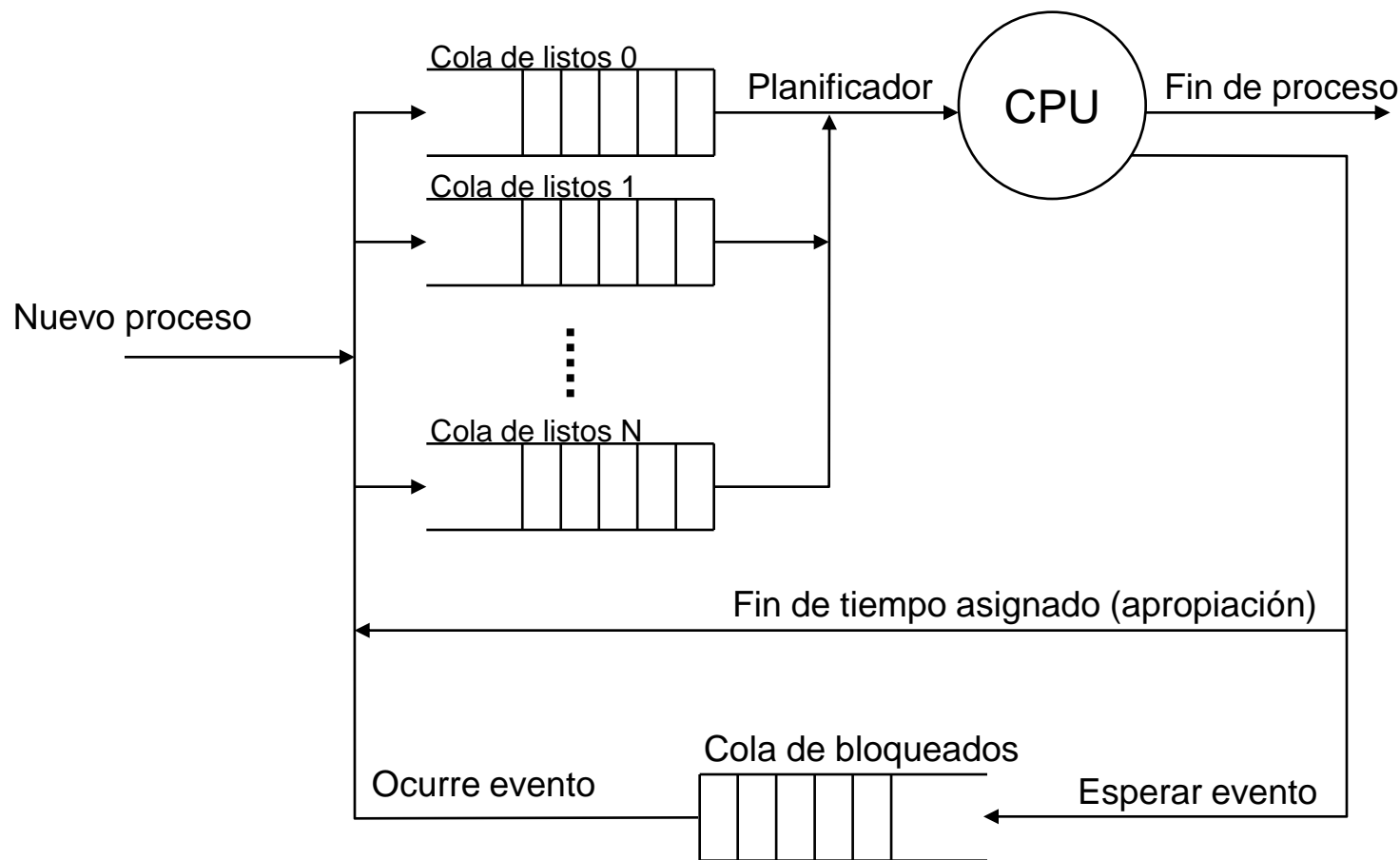
- Cada proceso tiene asignada una prioridad y se elige al proceso *listo* de mayor prioridad
- La prioridad (número) se asigna cuando se crea el proceso
  - Prioridad estática: no cambia a lo largo de la vida del proceso
  - Prioridad dinámica: puede cambiar durante la vida del proceso
- Por ejemplo: para favorecer a procesos limitados por E/S:
  - Prioridad  $1/F$  , donde  $F \Rightarrow$  fracción de quantum que utilizó
  - $Q = 100\text{ms}$  (quantum)
  - $F = 1/2$  (50ms)  $\Rightarrow$  prioridad= 2
  - $F = 1/50$  (2ms)  $\Rightarrow$  prioridad= 50
- Puede ser **apropiativo** (quita la CPU cuando hay un proceso *listo* de mayor prioridad) **o no**
- Se puede dar **inanición**. Solución:
  - Ir disminuyendo la prioridad de los que se ejecutan
  - Ir aumentando la prioridad de los que no se ejecutan



## 2.5.7 Planificación por niveles con realimentación

- **Varias colas de procesos listos pero con distintos niveles de prioridad:**
  - Cada cola tiene algoritmo de planificación propio
  - También planificación entre las distintas colas (suele ser planificación apropiativa por prioridad)
- Para una mayor flexibilidad: **realimentación**
  - Paso de colas de mayor a menor prioridad
  - Paso de colas de menor a mayor prioridad
- **Parámetros de diseño:**
  - Número de colas
  - Algoritmos de planificación
  - Criterio de ascenso y descenso
  - Cola inicial de un proceso nuevo, etc.

## 2.5.7 Esquema de funcionamiento



## 2.5.8 Planificación a corto, medio y largo plazo

- Hasta ahora nos hemos olvidado de los procesos *listos* que no están en memoria (*suspendidos*)
- Planificación de **varios** niveles
  - **Planificador a corto plazo (PCP)**
    - Selecciona de entre todos los procesos listos en memoria cuál pasar a la CPU
  - **Planificador a medio plazo (PMP)**
    - Selecciona qué procesos pasarán de memoria a disco (se suspenderán) y viceversa
  - **Planificador a largo plazo (PLP)**
    - Selecciona qué trabajos por lotes se dejarán pasar (a memoria o a disco)

## 2.5.8 Planificación a corto, medio y largo plazo

