Tema 2: Gestión de procesos

Septiembre de 2023

Introducción

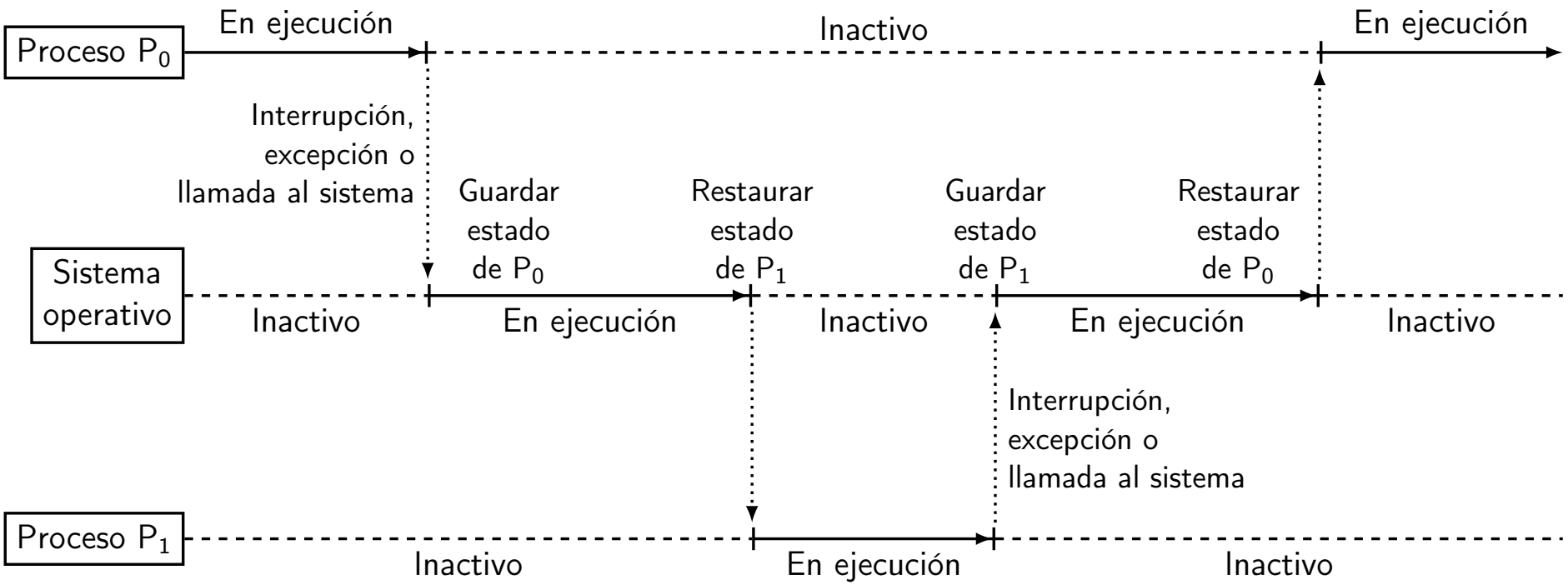
- La CPU es uno de los recursos más importantes a administrar
  - Ningún programa se puede ejecutar si no se le concede su uso
- La *planificación de la CPU*, es decir, la forma en la que se reparte entre los distintos procesos, será fundamental:
  - Seguirá un determinado *algoritmo de planificación*
  - Tendrá en cuenta distintas metas: eficiencia de uso de la CPU, ejecución de varios programas a la vez, etc.
  - La simplificaremos suponiendo una única CPU o núcleo
  - Necesaria porque suele haber muchos más procesos que CPUs o núcleos disponibles
- *Modelo de procesos*:
  - Modelo que organiza el software ejecutable de un sistema en torno al concepto de proceso
  - Facilita el uso del paralelismo

Concepto de proceso

- Un proceso (o proceso secuencial) es un *programa en ejecución*:
  - Programa: estático, contenido de un fichero en disco
  - Proceso: dinámico. En cada instante tiene un estado determinado por el contador de programa, contenido registros, valor de las variables definidas en el programa, etc.
- Un proceso necesita recursos para su funcionamiento: memoria RAM para código y datos, tiempo de CPU, espacio en disco para ficheros, etc.
- Un proceso puede comprender varios programas (sistemas Unix)
- Un mismo programa que se ejecuta varias veces puede dar lugar a varios procesos
- Un proceso nunca debe programarse con hipótesis implícitas de tiempo (al dejar la CPU, no sabe cuándo volverá a ella)

Pseudoparalelismo

- Sensación de que varias cosas se ejecutan al mismo tiempo a pesar de haber una CPU
  - Se obtiene cuando se producen frecuentes cambios de procesos
  - El paralelismo real se da cuando hay varias CPUs o cores
  - En los sistemas actuales conviven ambos tipos de paralelismo



Ventajas de tener procesos

- Compartir recursos físicos. Muchas veces recursos hardware caros o limitados (ordenador con varias CPUs, GPUs muy potentes, etc.)
- Compartir recursos lógicos: bases de datos, programas, etc.
- Acelerar los cálculos. Podemos acelerar una tarea dividiéndola en subtareas ejecutadas por procesos distintos. Especialmente útil cuando hay varias CPUs y/o núcleos
- Modularidad. A veces es más fácil construir un sistema si se diseña como un conjunto de procesos separados y no como un único proceso
- Comodidad. Simplemente, permitir que los usuarios hagan varias cosas a la vez

Cambio de proceso, de contexto y de modo

- Un *cambio de proceso* supone que la CPU deja de ejecutar código del proceso en curso para pasar al código de otro proceso
  - Debe ser posible continuar después la ejecución del proceso interrumpido $\rightarrow$ Guardar contexto
  - Realizado de forma automática y transparente por el sistema operativo cuando toma el control de la CPU
- Aunque muchas veces se confunden, un cambio de proceso, de contexto y de modo son cosas *distintas*:
  - Cambio de proceso: la CPU pasa de un proceso a otro
  - Cambio de contexto: la CPU pasa de un código a otro, pudiendo volver al primero como si nunca lo hubiera dejado; en otras palabras, pasa de un contexto de ejecución a otro
  - Cambio de modo: la CPU cambia entre modo usuario y núcleo
- Por ejemplo, en Unix, una llamada al sistema supone un cambio de contexto y modo, pero no de proceso

Creación de procesos

- En Unix los procesos se crean con `fork()`:
  - Crea copia idéntica del proceso que hace la llamada
  - Padre e hijo continúan ejecución tras la instrucción `fork()`
  - `fork()` devuelve 0 en el hijo y el PID del hijo en el padre → Sabiendo quién es, un proceso puede hacer una cosa u otra
- En Unix, `fork()` se complementa con la llamada `execve`:
  - Permite a un proceso cambiar de programa conservando:
    - Ficheros abiertos (se puede programar el cierre automático)
    - Tratamiento de señales (con algunas salvedades)
    - El PID y casi todas las demás propiedades del proceso
  - Exiten envoltorios por comodidad (`execl`, `execlp`, ...)
  - Importante: `execve` *no crea un nuevo proceso*
- Mediante `fork()`+`execve` se ejecutan todos los programas
- En Windows: `CreateProcess()` ≡ `fork()` y `exec()` a la vez

Ejemplo de fork+exec(): nanoshell.c

Padre (PID=1450)

```
1  #include <stdio.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main(void)
6  {
     pid_t pidhijo;
     char orden[80];

     while(1) {
11     printf("$ ");
       gets(orden);

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

     return 0;
   }
```

Ejemplo de fork+exec(): nanoshell.c

Padre (PID=1450)

```
1  #include <stdio.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main(void)
6  {
     pid_t pidhijo;
     char orden[80];

     while(1) {
11     printf("$ ");
       gets(orden);

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

     return 0;
   }
```

Ejemplo de fork+exec(): nanoshell.c

Padre (PID=1450)

```
1  #include <stdio.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main(void)
6  {
     pid_t pidhijo;
     char orden[80];

     while(1) {
11     printf("$ ");
       gets(orden);

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

     return 0;
   }
```

```
$ ls
```

Ejemplo de fork+exec(): nanoshell.c

Padre (PID=1450)

```
1  #include <stdio.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main(void)
6  {
     pid_t pidhijo;
     char orden[80];

     while(1) {
11     printf("$ ");
       gets(orden);

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

     return 0;
   }
```

Ejemplo de fork+exec(): nanoshell.c

Padre (PID=1450)

```
1  #include <stdio.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main(void)
6  {
     pid_t pidhijo;
     char orden[80];

     while(1) {
11     printf("$ ");
       gets(orden);

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

     return 0;
   }
```

pidhijo=2270

Hijo (PID=2270)

```
1  #include <stdio.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main(void)
6  {
     pid_t pidhijo;
     char orden[80];

     while(1) {
11     printf("$ ");
       gets(orden);

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

     return 0;
   }
```

pidhijo=0

Ejemplo de fork+exec(): nanoshell.c

Padre (PID=1450)

```
1  #include <stdio.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main(void)
6  {
     pid_t pidhijo;
     char orden[80];

     while(1) {
11     printf("$ ");
       gets(orden);

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

     return 0;
   }
```

Hijo (PID=2270)

```
1  #include <stdio.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main(void)
6  {
     pid_t pidhijo;
     char orden[80];

     while(1) {
11     printf("$ ");
       gets(orden);

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

     return 0;
   }
```

pidhijo=0

Ejemplo de fork+exec(): nanoshell.c

Padre (PID=1450)

```
1  #include <stdio.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main(void)
6  {
     pid_t pidhijo;
     char orden[80];

     while(1) {
11     printf("$ ");
       gets(orden);

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

     return 0;
   }
```

Hijo (PID=2270)

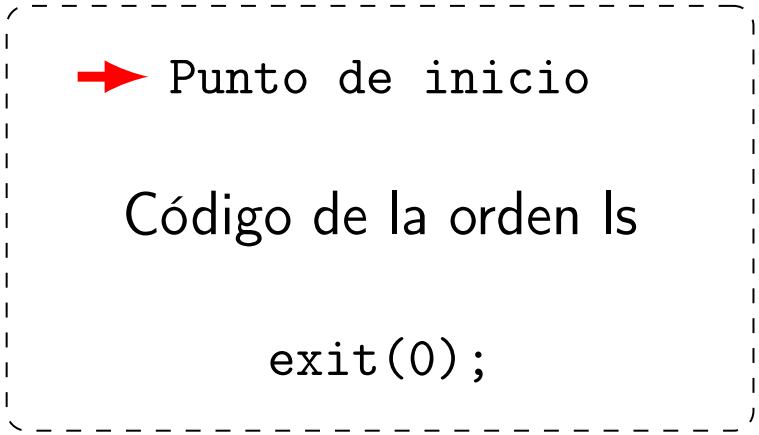
```
1  #include <stdio.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main(void)
6  {
     pid_t pidhijo;
     char orden[80];

     while(1) {
11     printf("$ ");
       gets(orden);

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

     return 0;
   }
```

Ejemplo de fork+exec(): nanoshell.c

Padre (PID=1450)

```
1  #include <stdio.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main(void)
6  {
     pid_t pidhijo;
     char orden[80];

     while(1) {
11     printf("$ ");
       gets(orden);

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

     return 0;
   }
```

Hijo (PID=2270)



Ejemplo de fork+exec(): nanoshell.c

Padre (PID=1450)

```
1  #include <stdio.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main(void)
6  {
     pid_t pidhijo;
     char orden[80];

     while(1) {
11     printf("$ ");
       gets(orden);

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

     return 0;
   }
```

Hijo (PID=2270)

Punto de inicio

Código de la orden ls

→ exit(0);

Ejemplo de fork+exec(): nanoshell.c

Padre (PID=1450)

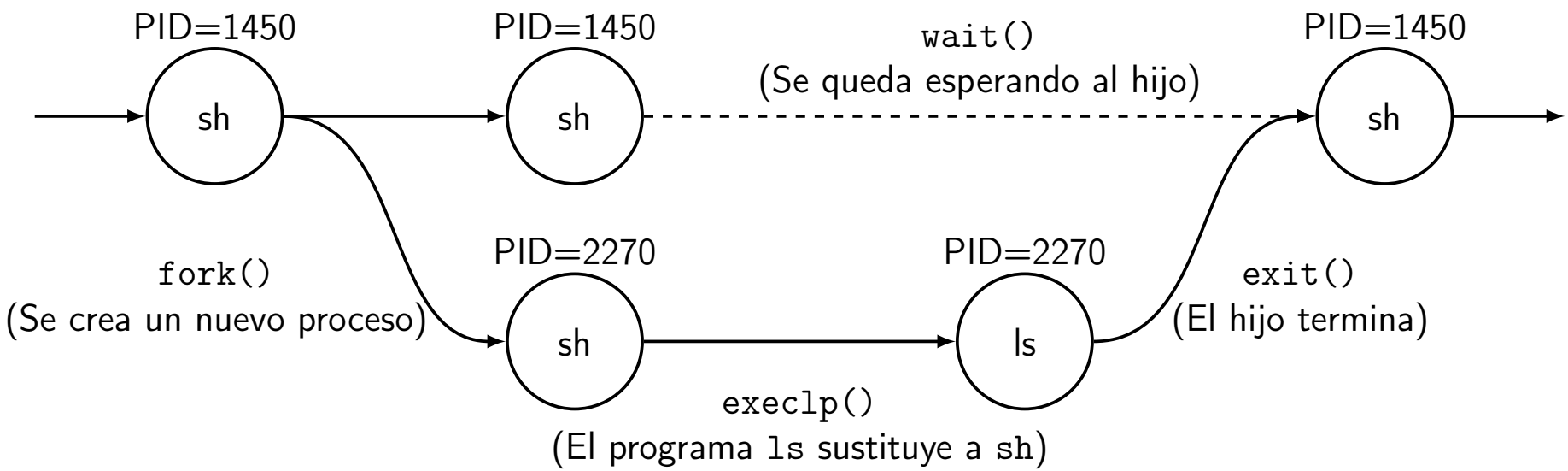
```
1  #include <stdio.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main(void)
6  {
     pid_t pidhijo;
     char orden[80];

     while(1) {
11     printf("$ ");
       gets(orden);

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

     return 0;
   }
```

Ejemplo de fork+exec(): nanoshell.c

Padre (PID=1450)

```
1  #include <stdio.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main(void)
6  {
     pid_t pidhijo;
     char orden[80];

     while(1) {
11     printf("$ ");
       gets(orden);

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

     return 0;
   }
```

Ejecución de la orden `ls` en `nanoshell.c`



Jerarquía de procesos

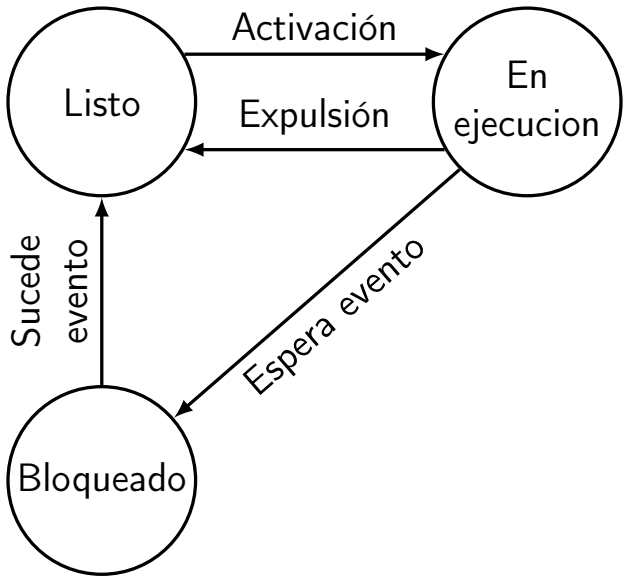
- En Unix, el SO mantiene una jerarquía de procesos padres e hijos en forma de árbol genealógico:
  - Podemos ver esa jerarquía con la orden `pstree`
  - La raíz es el proceso con PID 1, creado durante el arranque del sistema operativo y a partir del cual se crean todos los demás
- Al guardar las relaciones entre padres e hijos, Unix permite ciertas operaciones solo entre ellos
- En Windows, un proceso puede pasar a ser hijo de un proceso que no es su padre original
  - El concepto de jerarquía de procesos al estilo de Unix se pierde

Finalización de procesos

- Permite liberar los recursos usados por un proceso
- Puede ser:
  - Voluntaria: el propio proceso finaliza su ejecución por haber terminado o por haber encontrado un error
    - Unix: `exit()`
    - Windows: `ExitProcess()`
  - Involuntaria:
    - Por el propio sistema operativo ante un error fatal del proceso (división por cero, acceso no autorizado a una zona de memoria, ...)
    - Por otro proceso, con autorización suficiente. Unix: `kill()`, Windows: `TerminateProcess()`

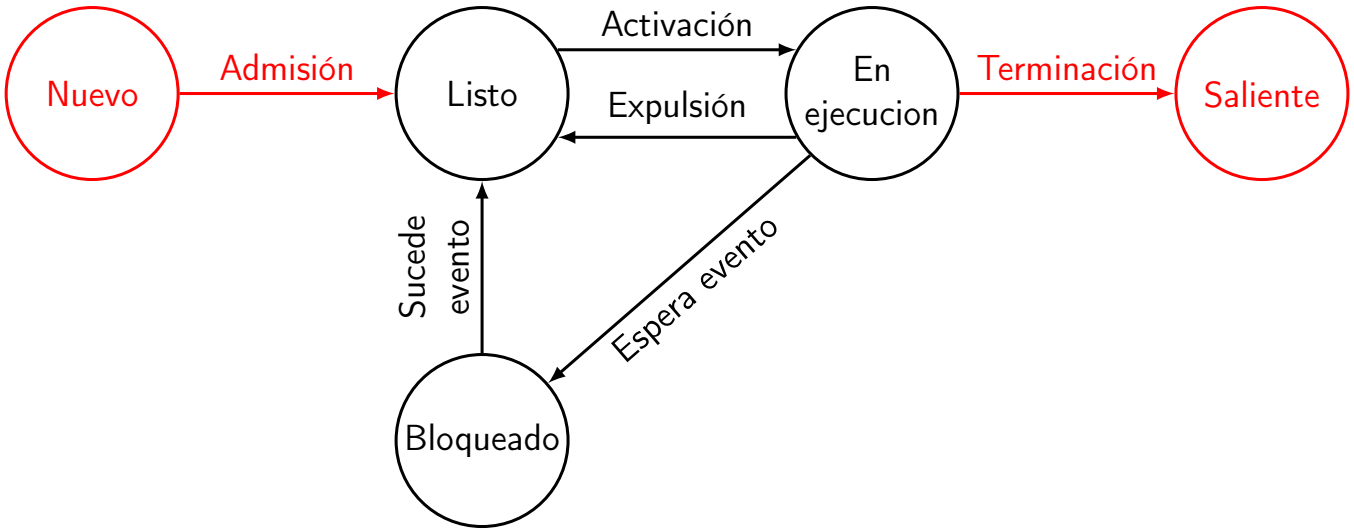
Estados de un proceso

- En su concepción más simple, un proceso puede estar en 3 estados posibles:
  - *En ejecución*: usa la CPU. Solo un proceso en este estado si hay una única CPU
  - *Listo*: puede ejecutarse, pero no tiene la CPU
  - *Bloqueado*: no puede ejecutarse porque está esperando algo
- 4 posibles transiciones, mostradas en la figura



Estados de un proceso

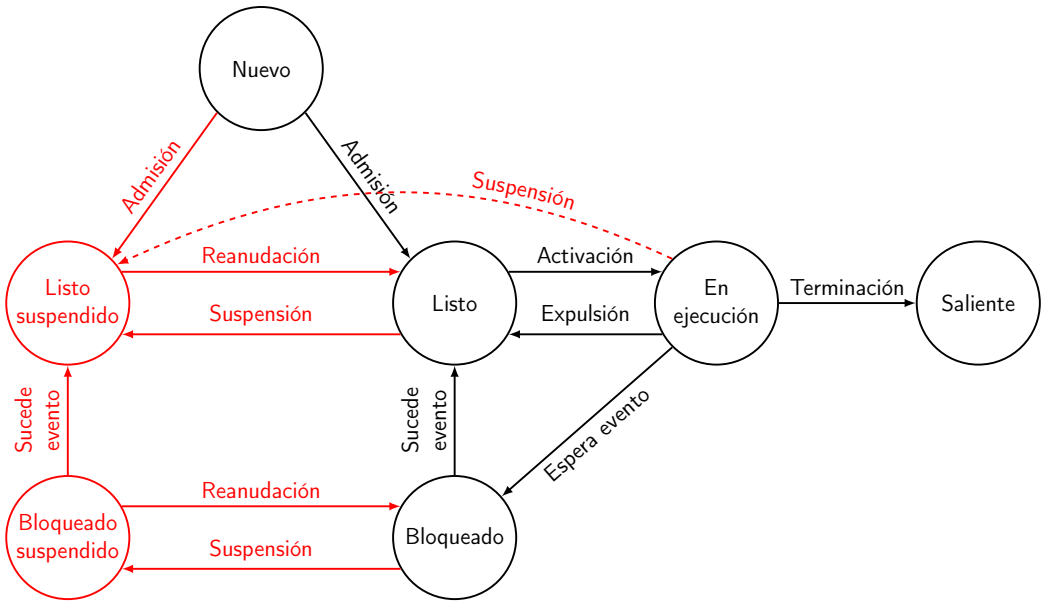
- Sistemas modernos son más complejos. Dos nuevos estados:
  - Nuevo: proceso recién creado, pero sin recursos suficientes
  - Saliente: proceso terminado, pero todavía con información útil
    - Caso de Unix: un proceso termina con `exit()`, pero su padre todavía no ha recogido su código de terminación con `wait()`



- Transiciones de «listo» y «bloqueado» a «saliente» también posibles, por ejemplo, cuando un proceso «mata» a otro

Estados incluyendo la suspensión y la reanudación

- Un proceso puede ser *suspendido* temporalmente (guardándose en disco) y después ser *reanudado*. Razones:
  - El sistema está funcionando mal
  - Depuración
  - Sistema muy cargado
- Nuevos estados: «listo suspendido» y «bloqueado suspendido»



Estados incluyendo la suspensión y la reanudación

- Nuevas transiciones:
  - De «bloqueado» a «bloqueado suspendido»: liberar memoria
  - De «bloqueado suspendido» a «listo suspendido»: sucede el evento por el que se bloqueó el proceso mientras está suspendido
  - De «listo suspendido» a «listo»: no hay procesos listos, un proceso listo-suspendido tiene prioridad alta, memoria libre, ...
  - De «listo» a «listo suspendido»: en ocasiones, se prefiere suspender un proceso listo de baja prioridad
  - De «nuevo» a «listo suspendido»: creamos un proceso con poca prioridad y no hay memoria disponible
  - De «bloqueado suspendido» a «bloqueado»: se libera memoria, el proceso tiene prioridad alta y se estima que pronto estará listo
  - De «en ejecución» a «listo suspendido»: pasa a listo un proceso bloqueado-suspendido de alta prioridad y no hay memoria libre
  - De cualquier estado a «saliente»: se mata un proceso

Implementación de procesos

- Nos vamos a centrar en dos aspectos sin entrar en detalles muy concretos:
  - Creación de un proceso
  - Cambio transparente de un proceso a otro
- Para entender ambos mecanismos necesitamos describir:
  - La tabla de procesos y los PCB
  - La estructura de un proceso

Tabla de procesos y PCBs

- El sistema operativo mantiene una *tabla de procesos* para saber qué procesos existen y en qué estado se encuentran
- Una entrada por proceso. Cada entrada se llama PCB (*Process Control Block*), o BCP en español, y guarda toda la información relacionada con el proceso correspondiente
- Al menos, en el PCB se debe guardar todo lo necesario para que un proceso que pierde la CPU pueda recuperarla más tarde como si nada hubiera pasado

Información habitual en un PCB

Administración de procesos	Administración de memoria	Administración de ficheros
Registros	Dirección del segmento de texto	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		

Creación de procesos

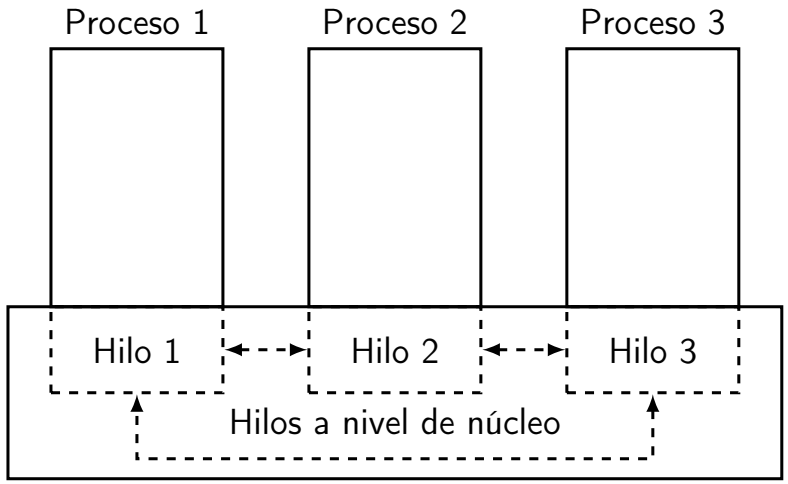
- Es una de las operaciones más complicadas
  - Hay que proveer al proceso de los recursos que necesita
- Al menos se debe:
  1. Dar nombre al proceso (PID en Unix)
  2. Insertarlo en la tabla de procesos, creando su PCB
  3. Determinar su prioridad inicial
  4. Asignarle recursos iniciales (principalmente, memoria)

Creación de procesos en Unix

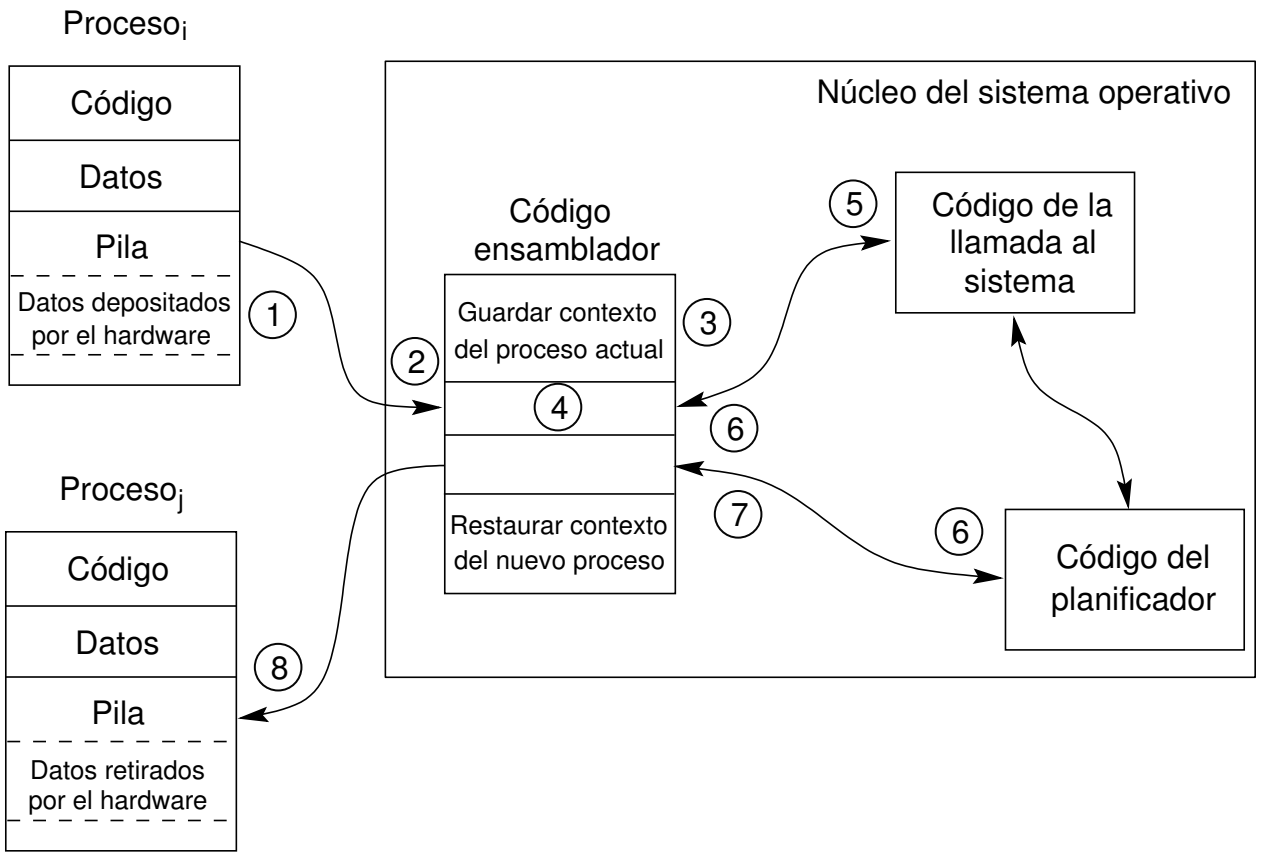
- Pasos durante una llamada al sistema `fork()`:
  1. El proceso padre realiza la llamada al sistema `fork()` $\rightarrow$ Paso al núcleo del sistema operativo
  2. El núcleo busca una entrada libre en la tabla de procesos para el proceso hijo y le asigna un PID
  3. Se copia el PCB del padre en el del hijo (con algunas salvedades)
  4. Se asigna memoria para los segmentos de datos y de pila del hijo y se copian los del padre en ellos. El segmento de código se comparte
  5. Se incrementan los contadores para cualquier fichero abierto por el padre ya que ahora también estarán abiertos en el hijo
  6. Se asigna al proceso hijo el estado «listo». Se devuelve el PID del hijo al padre y un valor de 0 al hijo como resultado de la llamada `fork()`

Estructura de un proceso

- En una llamada al sistema, al pasar al núcleo, ¿seguimos dentro del mismo proceso o hemos cambiado? → Depende del SO
- En Unix, al pasar al núcleo:
  - Seguimos en el mismo proceso
  - Por tanto, dos partes: la parte de usuario y la parte del núcleo
  - Parte del núcleo común a todos los procesos (misma copia)
  - Varios procesos en el núcleo → Cada uno con su PC y pila
  - Si bloqueo durante procesamiento llamada al sistema → Cambio de proceso. Al desbloquearse, continúa ejecución en el núcleo



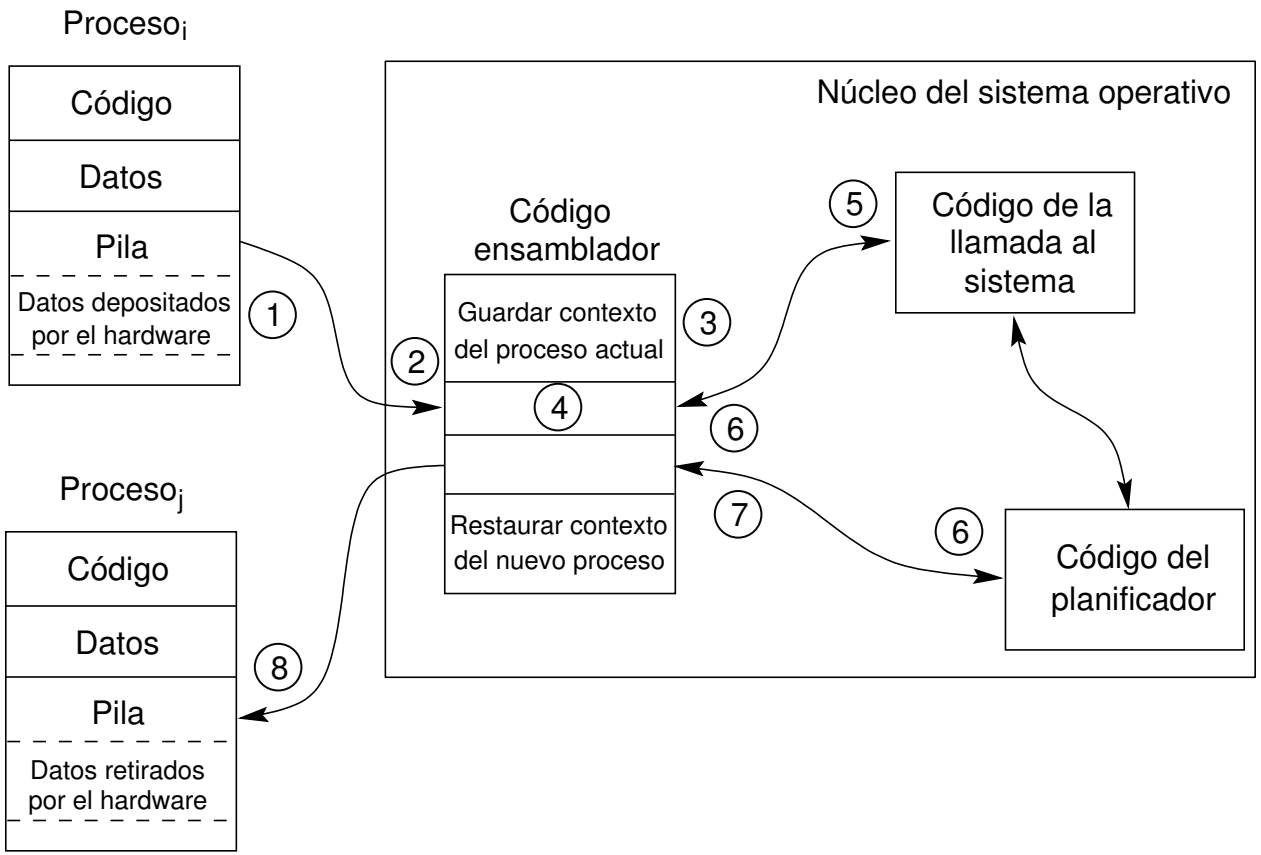
Ejemplo de cambio de proceso en una syscall



Aspecto clave: cambio automático y transparente de procesos

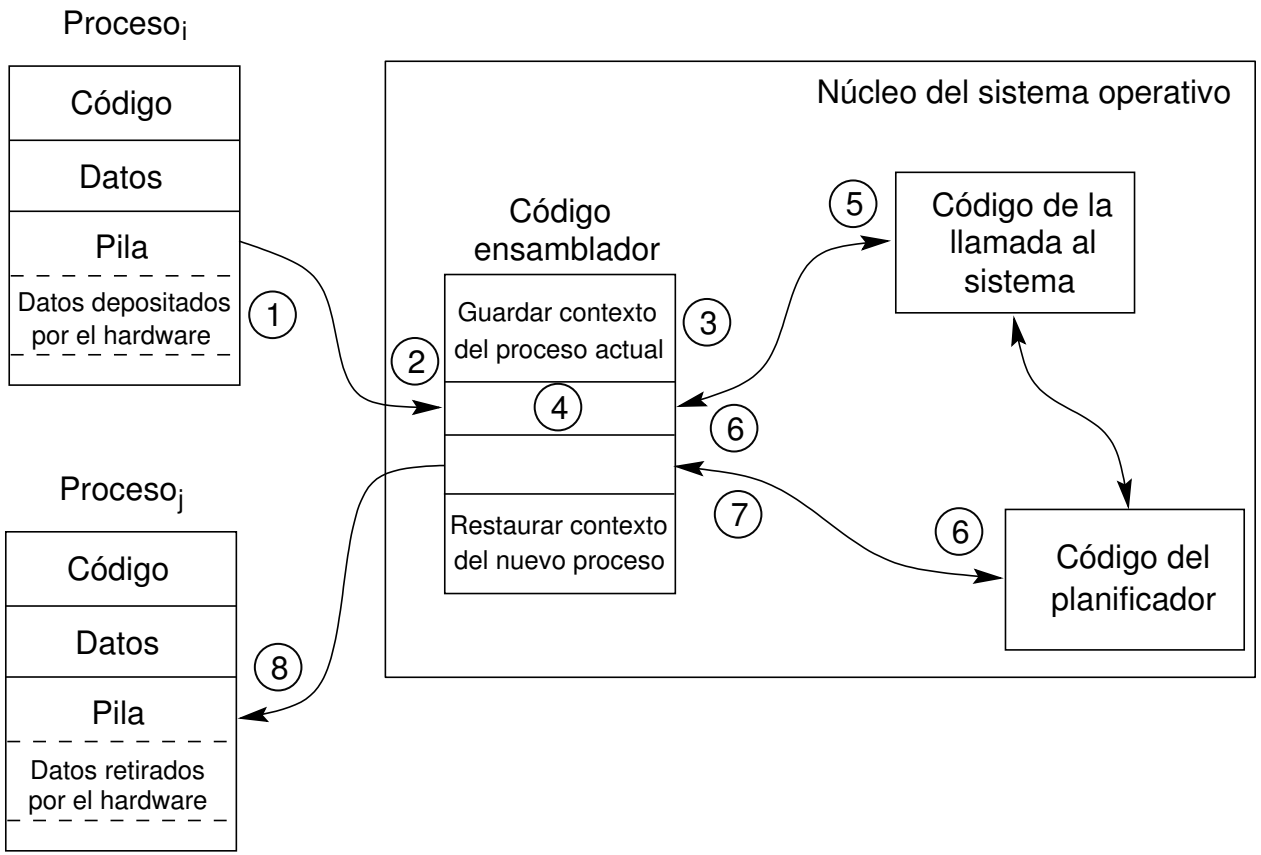
Realizado por el sistema operativo cuando toma el control

Ejemplo de cambio de proceso en una syscall



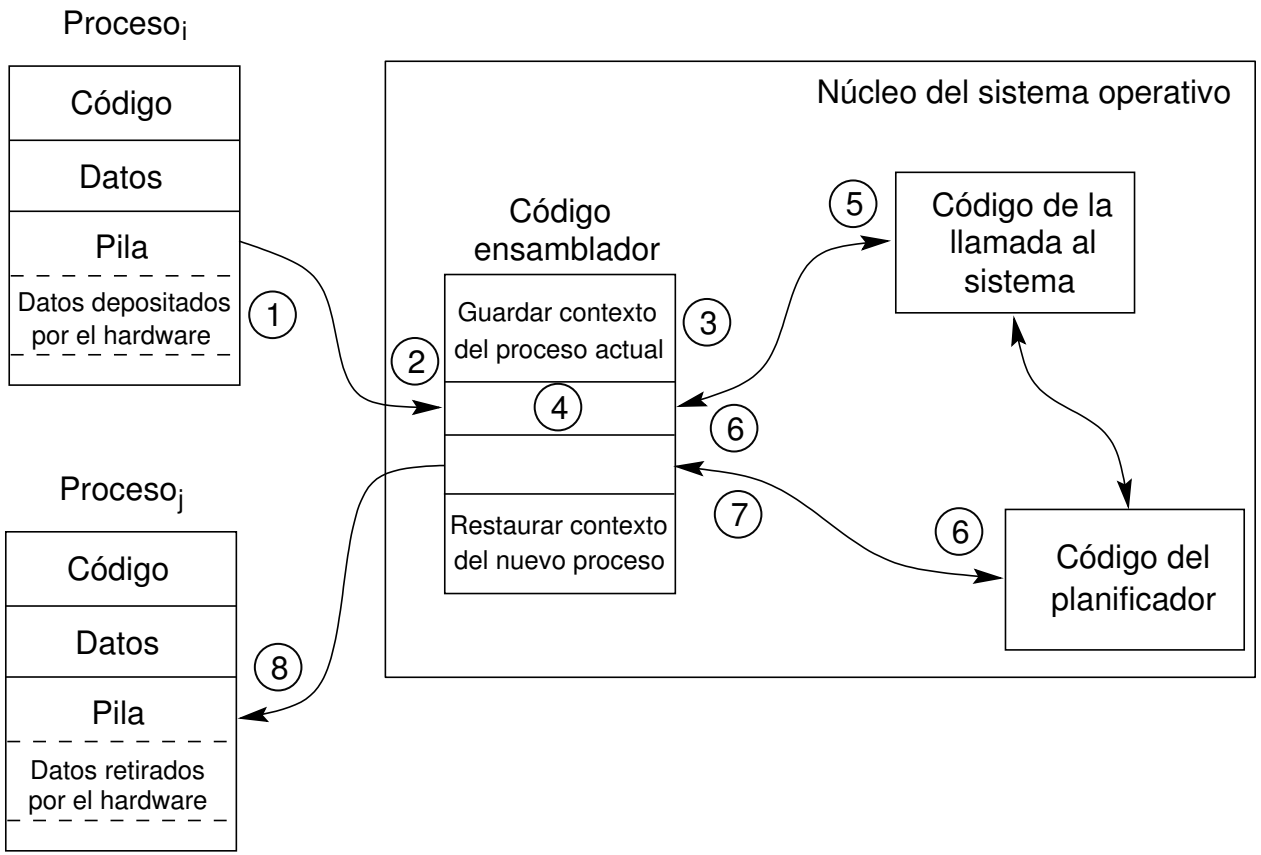
1. Hardware almacena el PC del proceso que solicita la llamada al sistema en su propia pila

Ejemplo de cambio de proceso en una syscall



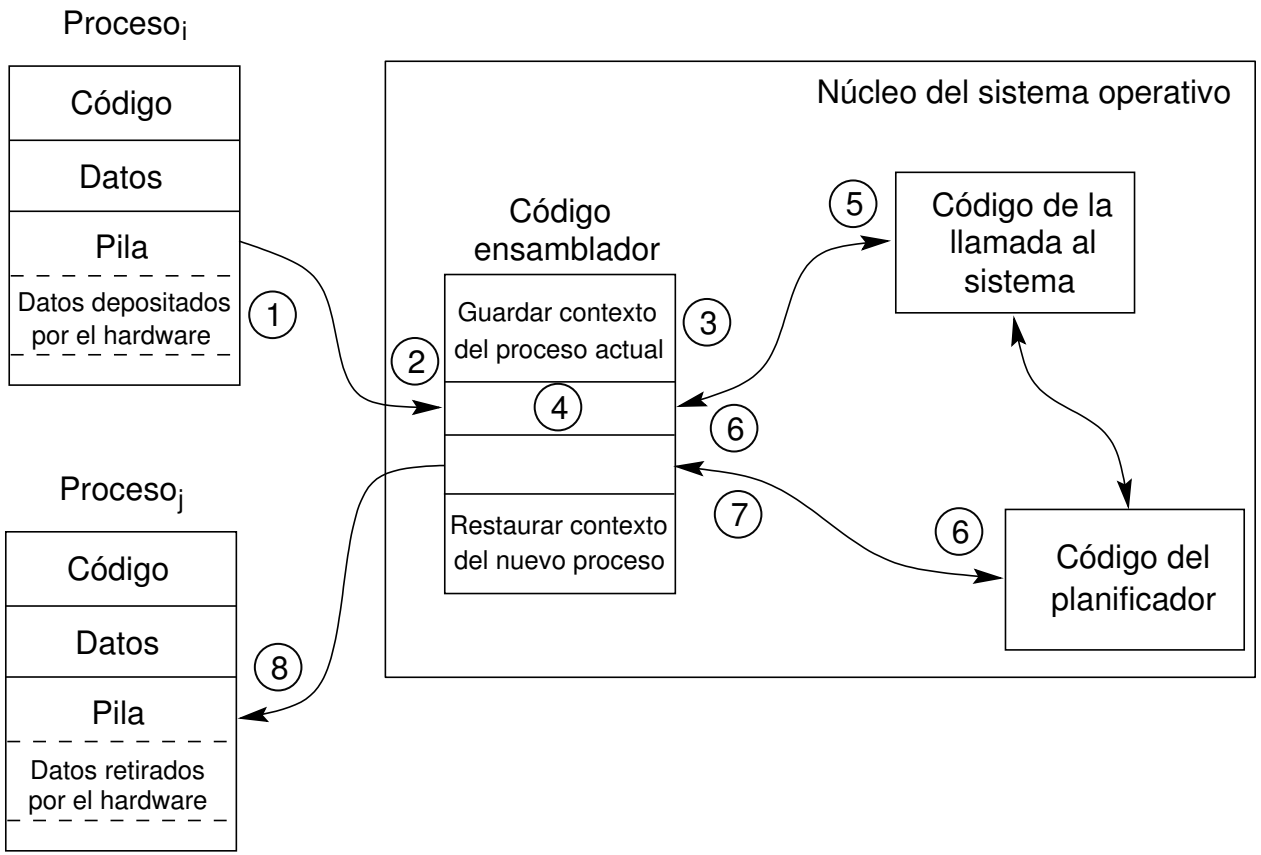
2. Hardware pasa a modo núcleo y carga el nuevo PC del vector de interrupciones (debe ser dirección inicio de rutina del SO)

Ejemplo de cambio de proceso en una syscall



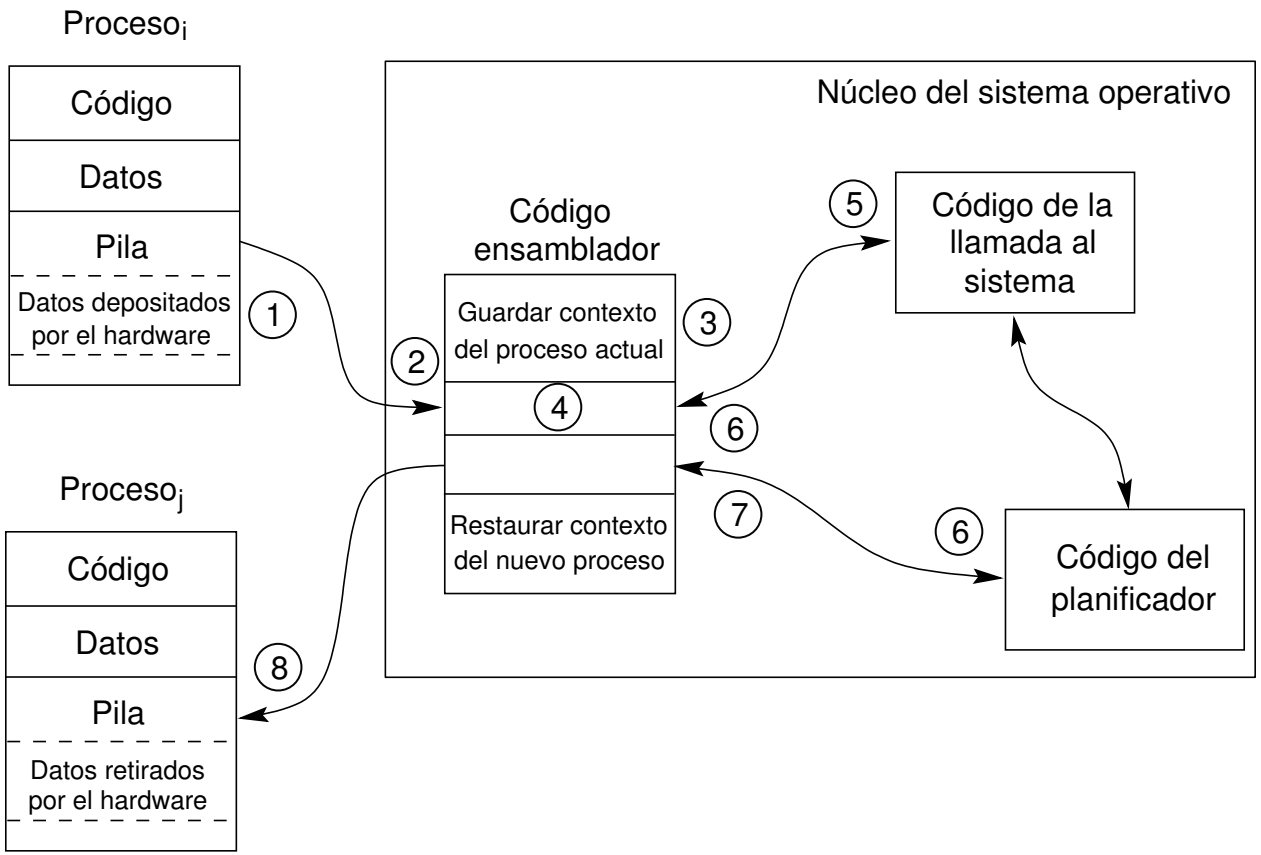
3. Proc. en ensamblador guarda «contexto» en PCB del proceso que solicita llamada al sistema

Ejemplo de cambio de proceso en una syscall



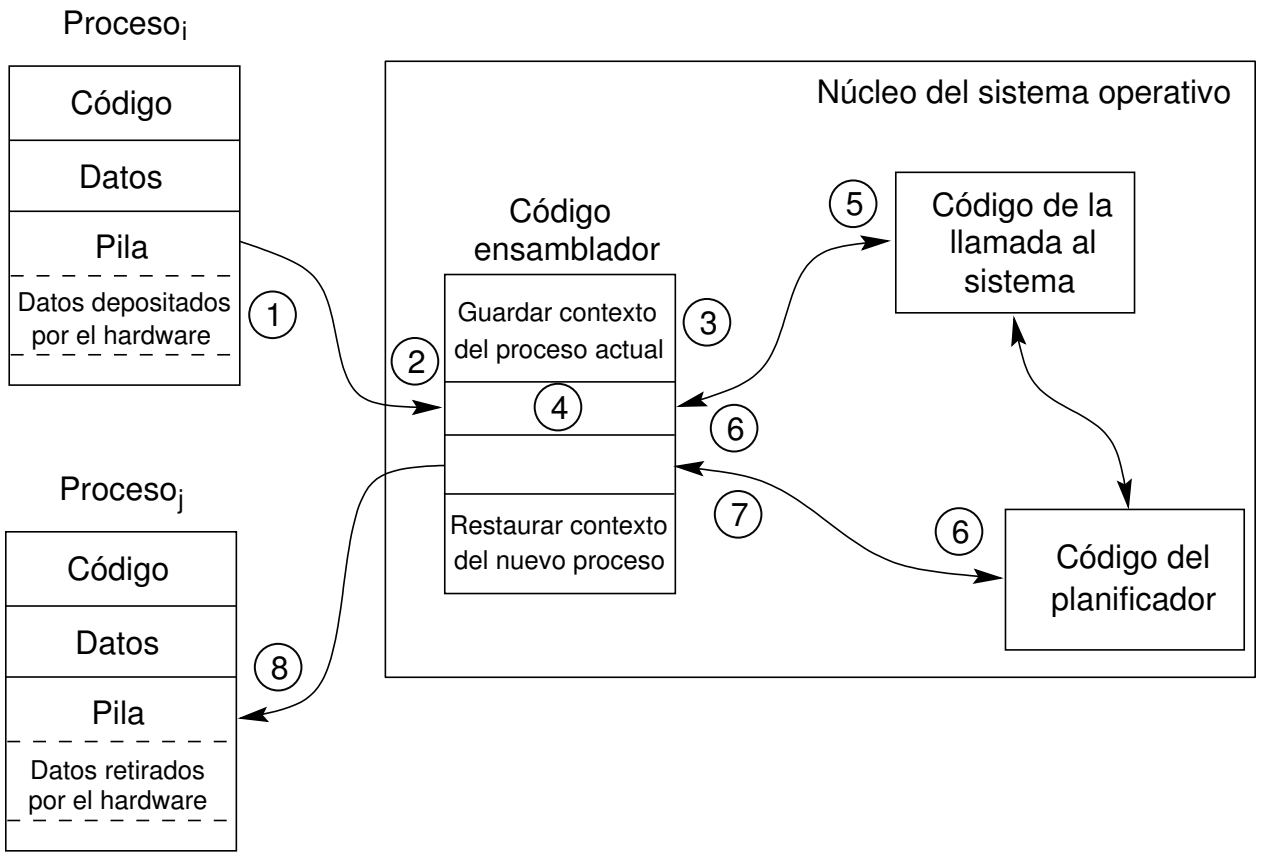
4. Proc. en ensamblador configura pila para núcleo. Hay una para la parte del núcleo de cada proceso

Ejemplo de cambio de proceso en una syscall



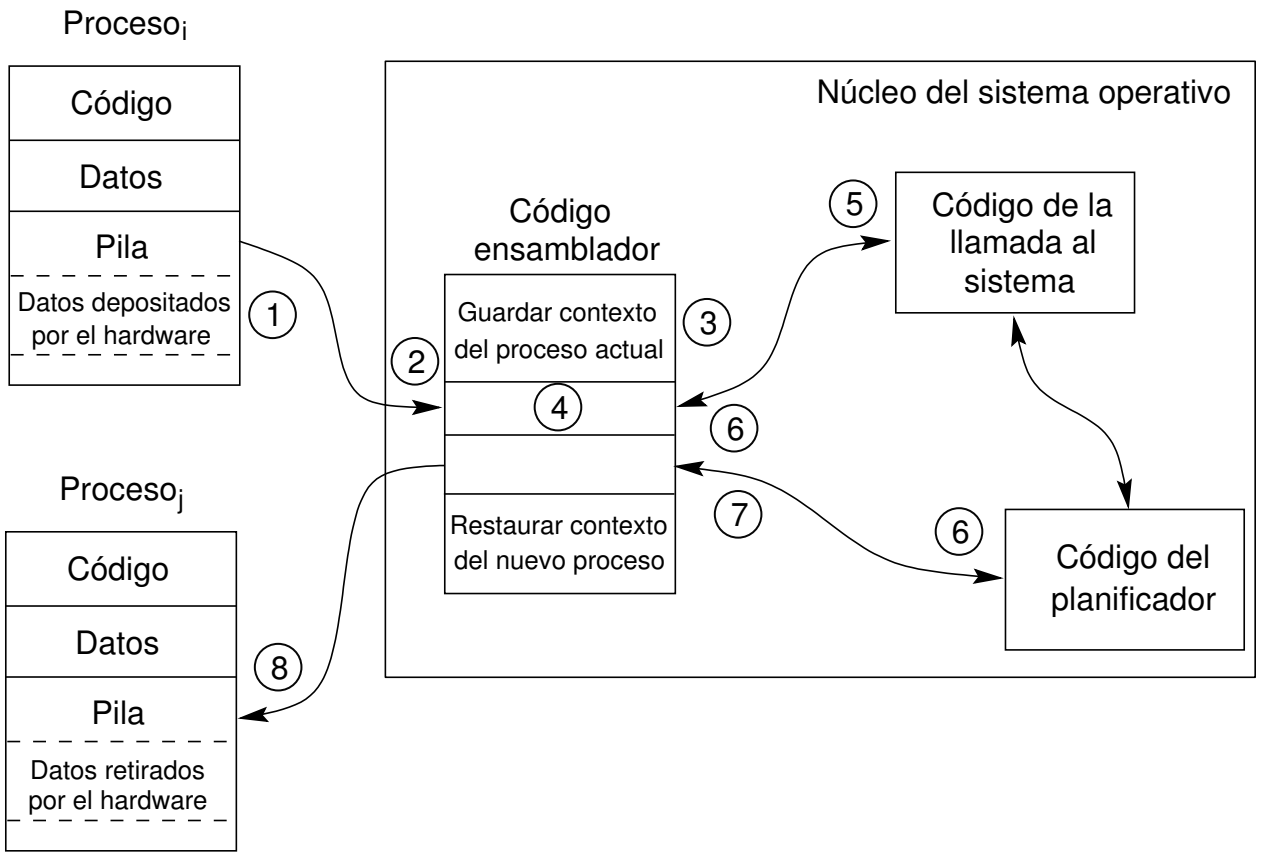
5. Proc. en ensamblador llama a proc. en C que implementa llamada al sistema. Si bloqueo $\rightarrow$ planificador

Ejemplo de cambio de proceso en una syscall



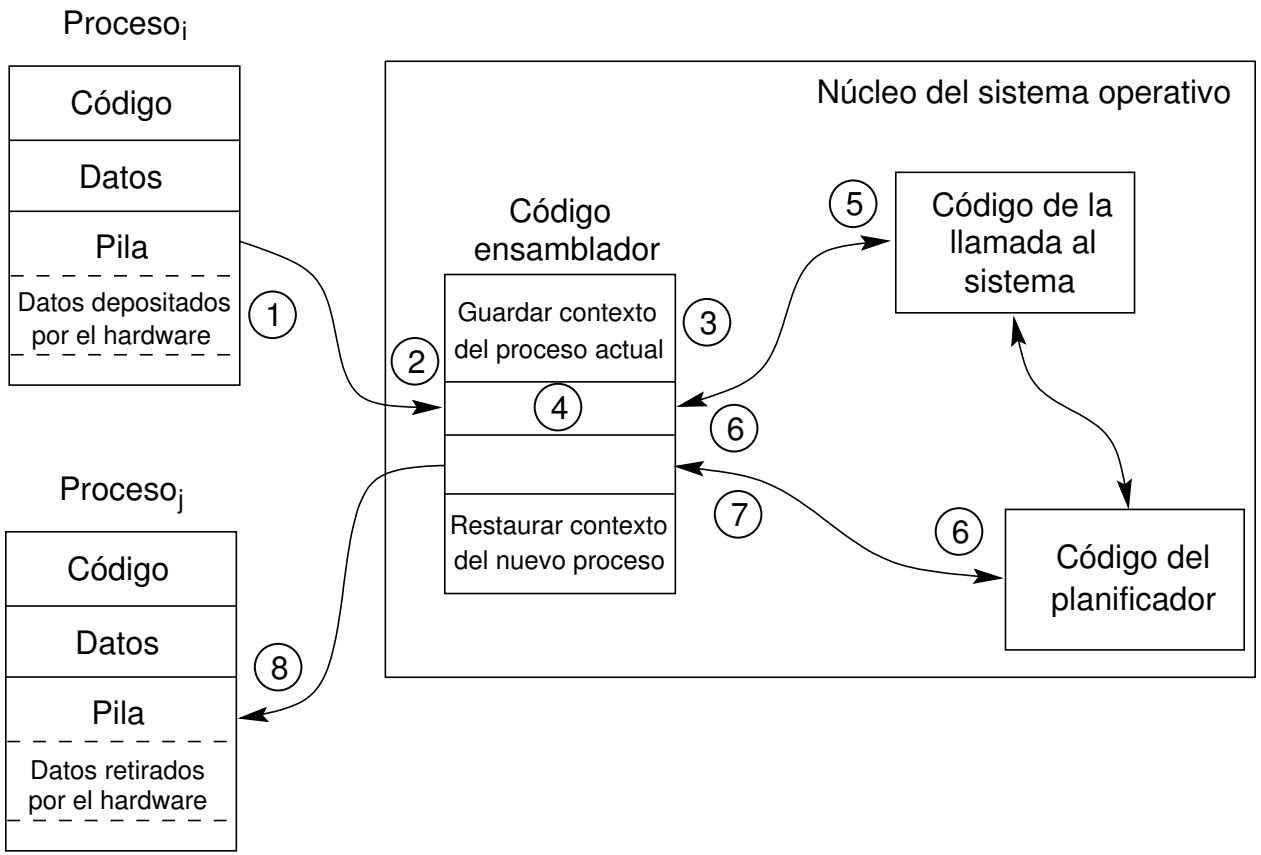
6. Se vuelve a proc. en ensamblador. Si necesario, llama a planificador (proc. en C que selecciona un proceso listo)

Ejemplo de cambio de proceso en una syscall



7. Proc. ensamblador restaura «contexto» del proceso seleccionado y prepara pila que el proceso usará en modo usuario

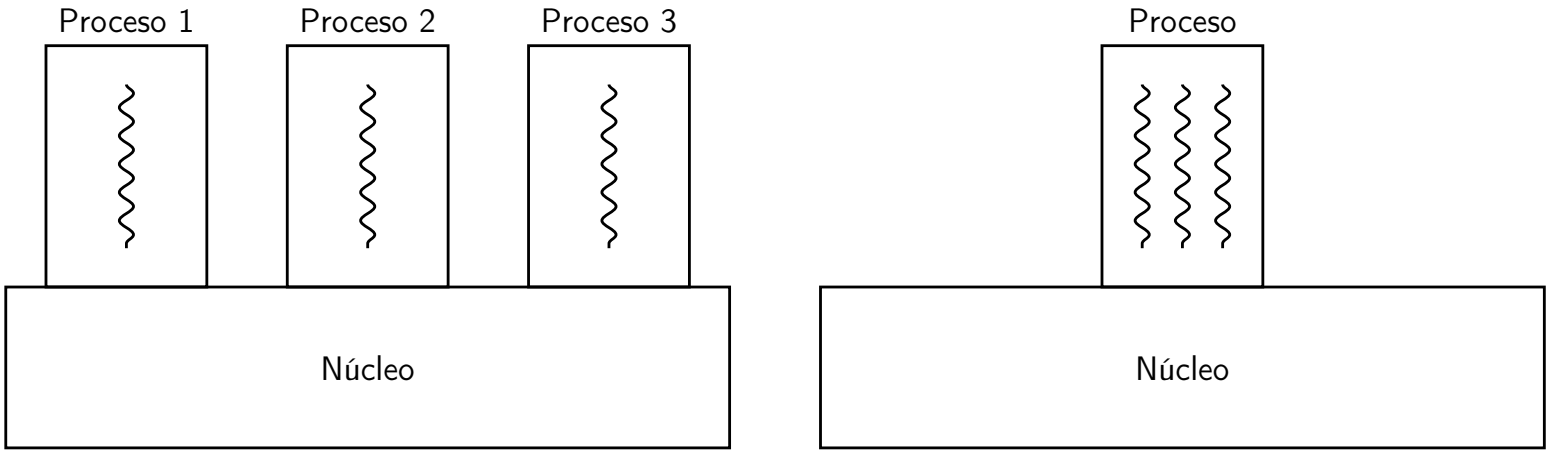
Ejemplo de cambio de proceso en una syscall



8. Cambio a modo usuario y regreso de la llamada al sistema (carga PC desde pila)

Hilos

- Hasta ahora, un proceso es algo que engloba 2 características:
  - Unidad de propiedad y asignación de recursos
  - Unidad de planificación y ejecución
- Sin embargo, características independientes:
  - Hilo (*thread*): unidad de planificación y ejecución
  - Proceso: unidad de propiedad de recursos
- Un proceso puede tener varios hilos



Ejemplo programa con hilos

```c
void * hilo_main (void * arg)
{
  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));

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

  return NULL;
}

int main (void)
{
  pthread_t thread;

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

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

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

  return 0;
}
```

Ejemplo programa con hilos

```c
void * hilo_main (void * arg)
{
  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));

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

  return NULL;
}

int main (void)
{
  pthread_t thread;

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

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

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

  return 0;
}
```

Ejemplo programa con hilos

```
void * hilo_main (void * arg)
{
  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));

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

  return NULL;
}

int main (void)
{
  pthread_t thread;

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

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

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

  return 0;
}
```

Ejemplo programa con hilos

```c
void * hilo_main (void * arg)
{
  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));

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

  return NULL;
}

int main (void)
{
  pthread_t thread;

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

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

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

  return 0;
}
```

Ejemplo programa con hilos

```
void * hilo_main (void * arg)
{
  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));

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

  return NULL;
}

int main (void)
{
  pthread_t thread;

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

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

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

  return 0;
}
```

Ejemplo programa con hilos

```
void * hilo_main (void * arg)
{
  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));

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

  return NULL;
}

int main (void)
{
  pthread_t thread;

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

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

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

  return 0;
}
```

Ejemplo programa con hilos

```
void * hilo_main (void * arg)
{
  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));

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

  return NULL;
}

int main (void)
{
  pthread_t thread;

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

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

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

  return 0;
}
```

Elementos por hilo y por proceso

Elementos por hilo	Elementos por proceso
Contador de programa	Espacio de direcciones
Registros de la CPU	Variables globales
Pila	Ficheros abiertos
Estado	Procesos hijos
	Alarmas
	Señales
	Información contable

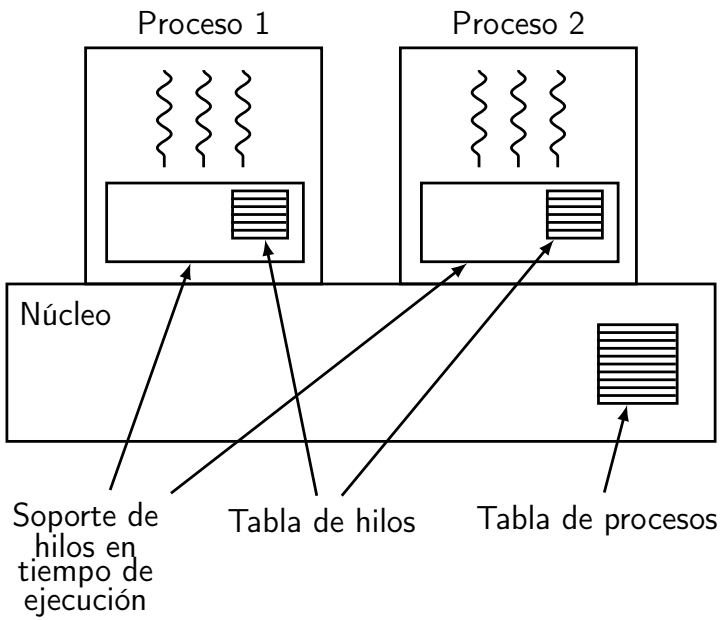
- Los distintos hilos de un proceso no son tan independientes como procesos distintos:
  - Comparten mismo espacio de direcciones de memoria
  - No hay protección entre hilos
  - Mecanismos de sincronización para datos globales

Aplicaciones de los hilos

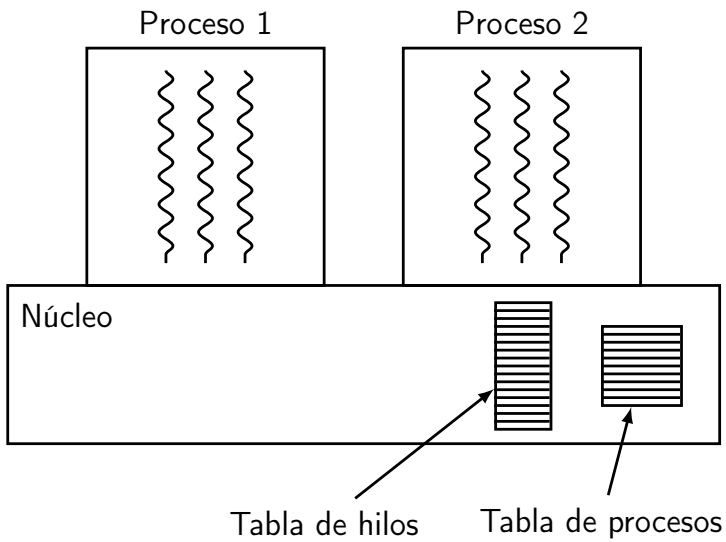
- Razones para preferir varios hilos en lugar de varios procesos:
  - Comunicación rápida entre hilos de un mismo proceso
  - Posible crear hilos hijos para que sean ellos lo que se bloqueen $\rightarrow$ Solapamiento E/S de un proceso con su cómputo $\rightarrow$ Se acelera ejecución
  - Mejor rendimiento: más rápido crear un hilo que un proceso, más rápido terminarlo y menos tiempo cambiar entre hilos de un mismo proceso
  - Con varias CPUs/núcleos, paralelismo real dentro de un proceso
  - Más fácil construcción de ciertos programa. Ejemplos:
    - Servidor de ficheros: hilos para atender peticiones
    - Programa gráfico: un hilo con entrada, otro ejecutando órdenes previas
    - Procesador de texto: hilo para guardar periódicamente

Implementación de hilos

- Los hilos pueden ser implementados:
  - Mediante funciones de biblioteca a nivel de usuario
  - Directamente por el núcleo del SO, mediante llamadas al sistema (Windows, Linux, ...). Lo más habitual actualmente



(a)



(b)

Hilos en modo usuario

- Ventajas:
  - Posibles en núcleos que no implementan hilos. Cada proceso tiene su tabla de hilos
  - Cambios de contexto entre hilos más rápidos (no interviene SO)
  - Cada proceso puede tener su algoritmo de planificación de hilos
- Inconvenientes:
  - Llamada al sistema bloqueante realizada por un hilo bloqueará a todo el proceso $\rightarrow$ Imposible solapar E/S y cómputo de un proceso. Posible solución: llamadas al sistema no bloqueantes, pero programación más difícil
  - Problema similar con los fallos de página
  - La CPU asignada al proceso tiene que repartirse entre sus hilos
  - Con varias CPUs/núcleos, no es posible paralelismo real dentro de un proceso

Hilos en modo núcleo

- Ventajas:
  - Al mantener el núcleo la tabla de hilos, puede repartir la CPU entre ellos. Con varias CPUs, varios hilos de un mismo proceso se pueden ejecutar en paralelo
  - Una llamada al sistema bloqueante solo bloquea al hilo que la realiza. Otros hilos del mismo proceso pueden continuar
  - Los fallos de página tampoco son un problema por lo mismo
- Inconvenientes:
  - Las funciones para sincronización entre hilos son llamadas al sistema $\rightarrow$ Más costosas
  - La creación y destrucción de hilos son más costosas también. Posible solución: reutilización de hilos
- Aunque planificación con hilos, todavía acciones que afectan a procesos por completo: suspensión, terminación, ...

Planificadores y metas

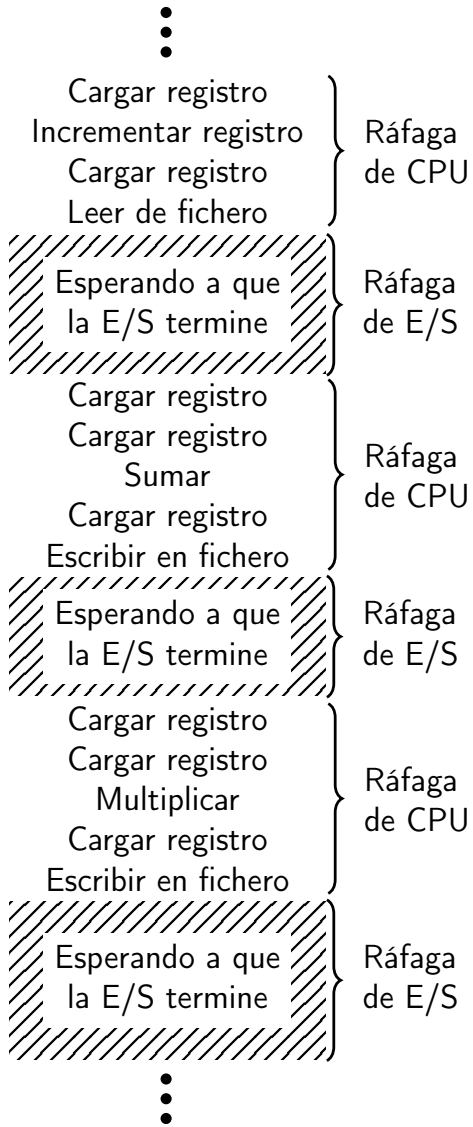
- *Planificador*: parte del sistema operativo que decide qué proceso listo ejecutar siguiendo un *algoritmo de planificación*
- Algunas de las posibles metas a conseguir:
  1. *Equidad*: reparto «justo» de la CPU
  2. *Eficiencia*: mantener CPU ocupada al máximo con trabajo útil
  3. *Minimizar el tiempo de espera*, que es el tiempo que pasa un proceso listo esperando que se le conceda la CPU
  4. *Minimizar el tiempo de respuesta*, que es el que va desde que se solicita una acción hasta obtener los resultados. Para usuarios interactivos
  5. *Minimizar el tiempo de regreso* o *de retorno*, que es el que va desde que se entrega un trabajo hasta obtener los resultados. Para trabajos por lotes
- Hay metas contradictorias, como la 4 y la 5 $\rightarrow$ No hay algoritmos óptimos

Planificación apropiativa y no apropiativa

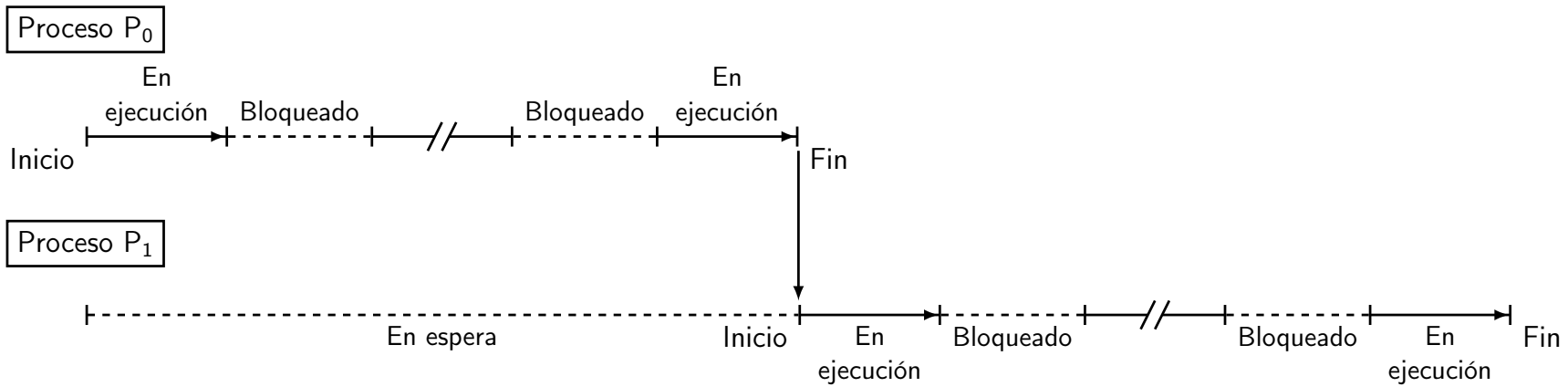
- Complicación adicional para los planificadores: procesos son impredecibles
  - Algunos esperan mucho en operaciones de E/S; otros utilizan mucha CPU
  - Por tanto, cuando se concede la CPU a un proceso, no se sabe cuándo este se bloqueará o terminará
- Dos estrategias posibles:
  - *Planificación apropiativa*: es posible expulsar de la CPU a procesos ejecutables. Necesaria para procesos interactivos y sistemas multitarea
  - *Planificación no apropiativa*: un proceso no deja la CPU hasta que se bloquea o termina. Útil para procesos por lotes

Ráfagas de CPU y E/S

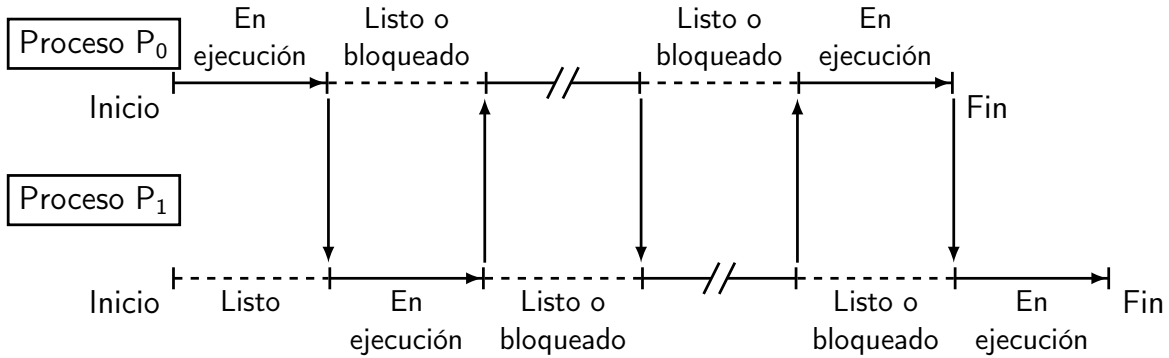
- Propiedad de los procesos: la ejecución de un proceso consiste en un ciclo de uso de la CPU (*ráfaga de CPU*) y un ciclo de espera de E/S (*ráfaga de E/S*), alternando entre estos dos ciclos
  - Se empieza y termina con una ráfaga de CPU
  - La duración de las ráfagas no se conoce a priori
  - Propiedad clave para la existencia de la multiprogramación y la multitarea



Ráfagas de CPU y E/S



(a) Sin multiprogramación



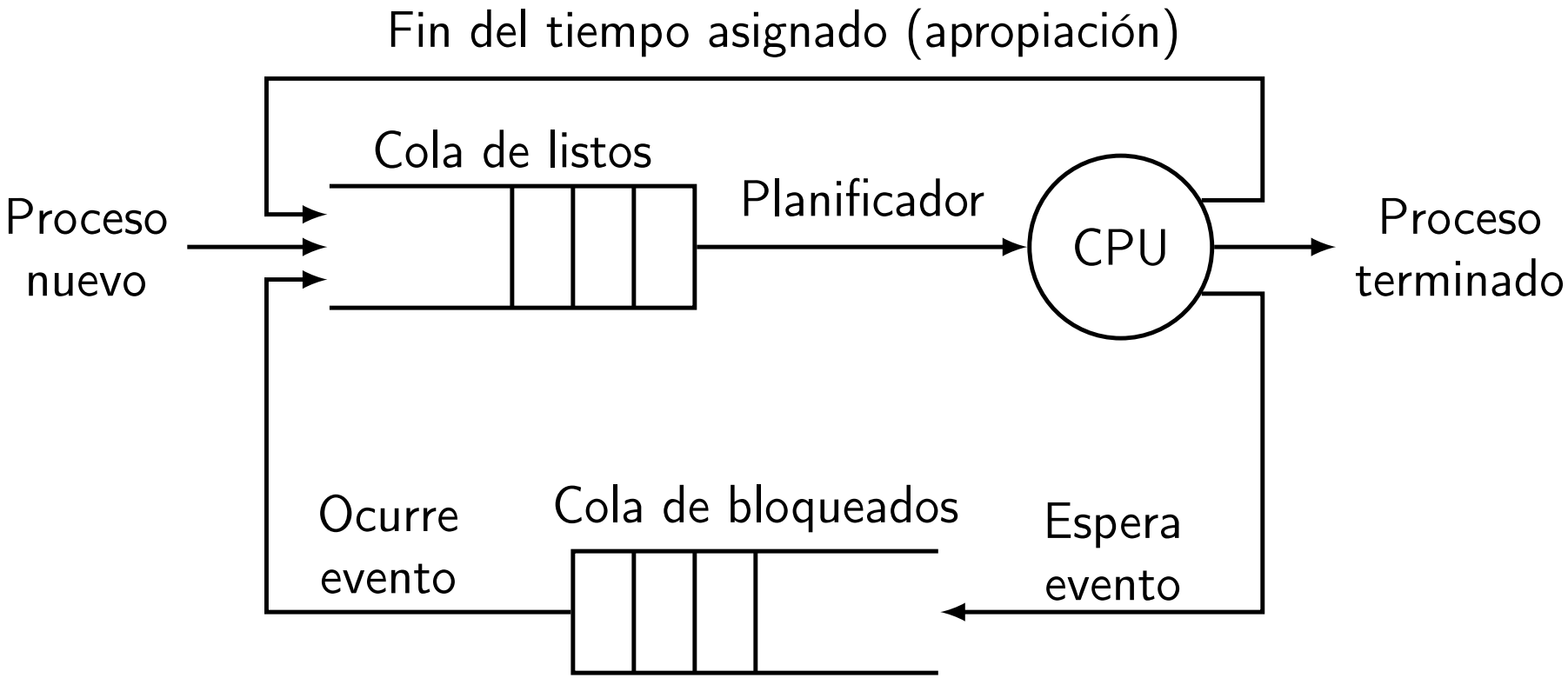
(b) Con multiprogramación

Proceso limitado por CPU o E/S

- *Proceso limitado por E/S*: pasa la mayor parte de su tiempo en espera de una operación de E/S. Normalmente, muchas ráfagas de CPU breves. Ejemplo: intérprete de órdenes
- *Proceso limitado por la CPU*: necesita usar la CPU la mayor parte de su tiempo. Normalmente, pocas ráfagas de CPU de muy larga duración. Ejemplo: proceso que resuelve ecuaciones

Algoritmos de planificación

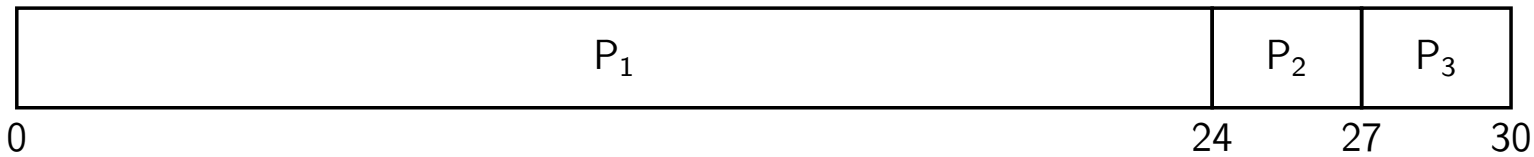
- Se pueden analizar de forma gráfica representando el sistema mediante un diagrama de colas



Primero en llegar, primero en ser servido (FCFS)

- Se asigna la CPU en orden de llegada a la cola de listos
- No apropiativo → Adecuado para procesos por lotes

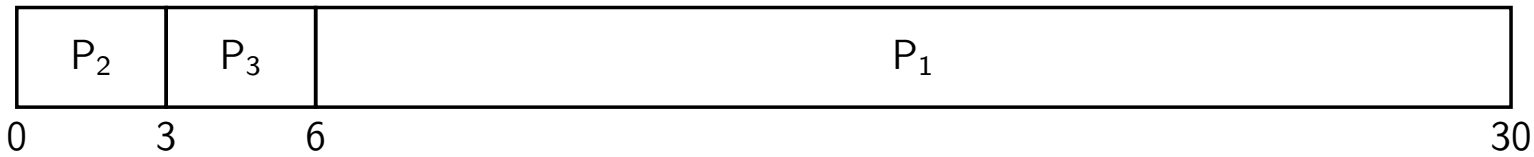
Proceso	Ráfaga de CPU
P1	24
P2	3
P3	3



- Problema 1: tiempo promedio de respuesta puede ser bastante largo. En el ejemplo es $\frac{24+27+30}{3} = 27$ (comparar con SJF)
- Problema 2: efecto convoy. Cuando proceso limitado por CPU va seguido de muchos limitados por E/S. Se desaprovechan recursos

Primero el trabajo más corto (SJF)

- Se le da la CPU al proceso listo que menos tiempo va a tardar en ejecutarse en total
- No apropiativo



Tiempo promedio de respuesta$=\frac{3+6+30}{3} = 13$

- Adecuado para procesos por lotes, en donde los tiempos de ejecución se suelen conocer de antemano. La selección se puede hacer teniendo en cuenta:
  - El tiempo total de ejecución (incluyendo E/S)
  - El tiempo de la siguiente ráfaga de CPU (si es que se conoce)

Primero el trabajo más corto (SJF)

- También aplicable a procesos interactivos que se comportan así:

  Esperar orden $\rightarrow$ Ejecutar orden $\rightarrow$ Esperar orden $\rightarrow$ Ejecutar orden ...

  - Cada orden se puede ver como un proceso por lotes cuyo tiempo de ejecución se conoce
  - Estimación del tiempo de ejecución de una orden según comportamiento pasado:

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

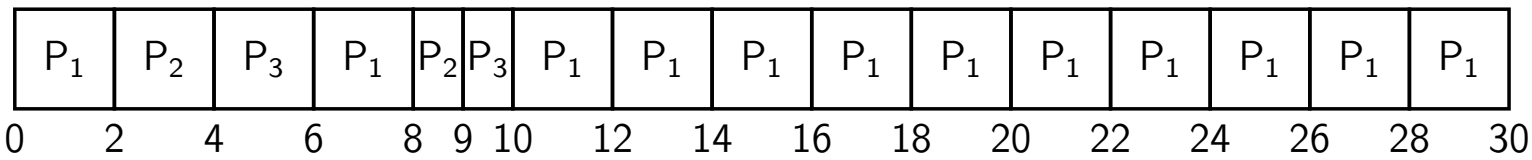
Estimación	Tiempo real
$E_1 = T_0$	T_1
$E_2 = \frac{1}{2} \cdot E_1 + (1 - \frac{1}{2}) \cdot T_1 = \frac{T_0}{2} + \frac{T_1}{2}$	T_2
$E_3 = \frac{1}{2} \cdot E_2 + (1 - \frac{1}{2}) \cdot T_2 = \frac{T_0}{4} + \frac{T_1}{4} + \frac{T_2}{2}$	T_3
$E_4 = \frac{1}{2} \cdot E_3 + (1 - \frac{1}{2}) \cdot T_3 = \frac{T_0}{8} + \frac{T_1}{8} + \frac{T_2}{4} + \frac{T_3}{2}$	T_4
...	...

Primero el del menor tiempo restante (SRTF)

- Es el SJF, pero apropiativo
- Un proceso se expulsa de la CPU si llega otro proceso listo con ráfaga de CPU menor que lo que resta del proceso que se está ejecutando
- También aplicable, pero tiene menos sentido, a procesos por lotes teniendo en cuenta la duración total

Round robin (RR) o planificación circular

- De las más antiguas, sencillas, justas y de uso más amplio
- Los procesos se atienden en orden de llegada a la cola de listos
- Cada proceso tiene asignado un *quantum* de ejecución. Por tanto, se cambia de proceso cuando el que tiene la CPU:
  - Consume su quantum → Se expulsa y vuelve al final de la cola de listos → Algoritmo apropiativo
  - Termina antes de consumir su quantum
  - Se queda bloqueado



- Importante longitud del quantum:
  - Si pequeño, muchos cambios de proceso → CPU se desperdicia
  - Si grande, últimos procesos listos tardan en ser atendidos. Malo para procesos interactivos

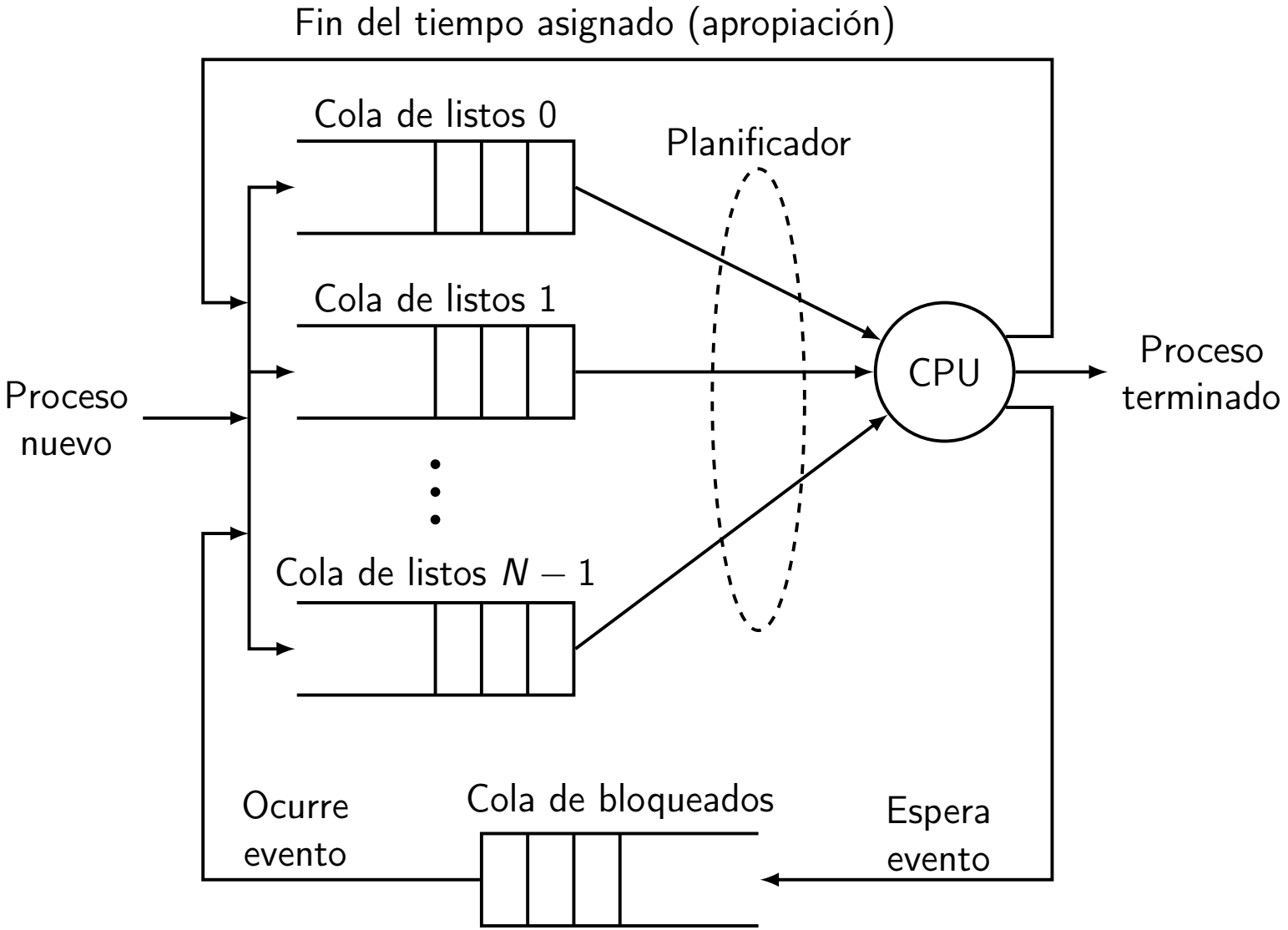
Planificación por prioridad

- Cada proceso tiene una prioridad y el proceso ejecutable de mayor prioridad toma la CPU
- La asignación de prioridades puede ser *estática* o *dinámica*
- La planificación puede ser *apropiativa* o *no apropiativa*
- Se puede dar *inanición*: procesos de baja prioridad nunca consiguen el control de la CPU. Posible solución:
  - ↑ periódicamente prioridad de procesos listos de baja prioridad

Múltiples colas con realimentación

- La más general, pero también la más compleja
- Existen N colas de procesos listos. Un proceso nuevo irá a una cola u otra en función de ciertos criterios:
  - Tipo de proceso (interactivo o por lotes)
  - Importancia del proceso (de sistema o de usuario)
  - Cola que utilizó la última vez
  - Consumo esperado de CPU del proceso, etc.
- Debe haber planificación entre colas y planificación dentro de cada cola:
  - Generalmente, planificación apropiativa por prioridad entre colas
  - Luego, round robin, FCFS, etc. en cada cola
- Si los procesos pueden cambiar de cola $\rightarrow$ *planificación de múltiples colas con realimentación*
  - Habrá que establecer criterios para cambiar de cola

Múltiples colas con realimentación



Planificación a corto, medio y largo plazo

- Hasta ahora hemos supuesto que todos los proceso ejecutables se encuentran en memoria
- ¿Qué pasa si los procesos se pueden suspender y reanudar?
- Solución: tener dos niveles de planificación
  - *Planificador a corto plazo* (PCP): lo visto hasta ahora
  - *Planificador a medio plazo* (PMP): para el intercambio de procesos entre la memoria principal y el disco. Se llama periódicamente e intercambia procesos en base a:
    - Tiempo transcurrido desde el último intercambio del proceso
    - Tiempo de CPU utilizado recientemente por el proceso
    - Tamaño del proceso (mejor intercambiar pequeños)
    - Prioridad del proceso
- También posibles tres niveles con un *planificador a largo plazo* (PLP): decide qué trabajo por lotes pasa a ejecutarse

Planificación a corto, medio y largo plazo