

Fundamentos de los Sistemas Operativos

Tema 2. Procesos

© 1998-2015 José Miguel Santos - Alexis Quesada - Francisco Santana

Contenidos del Tema 2

- Qué es un proceso
- Estructuras de datos para gestionar procesos
- API para trabajar con procesos
- Hilos (*threads*)
- Algoritmos de planificación de la CPU

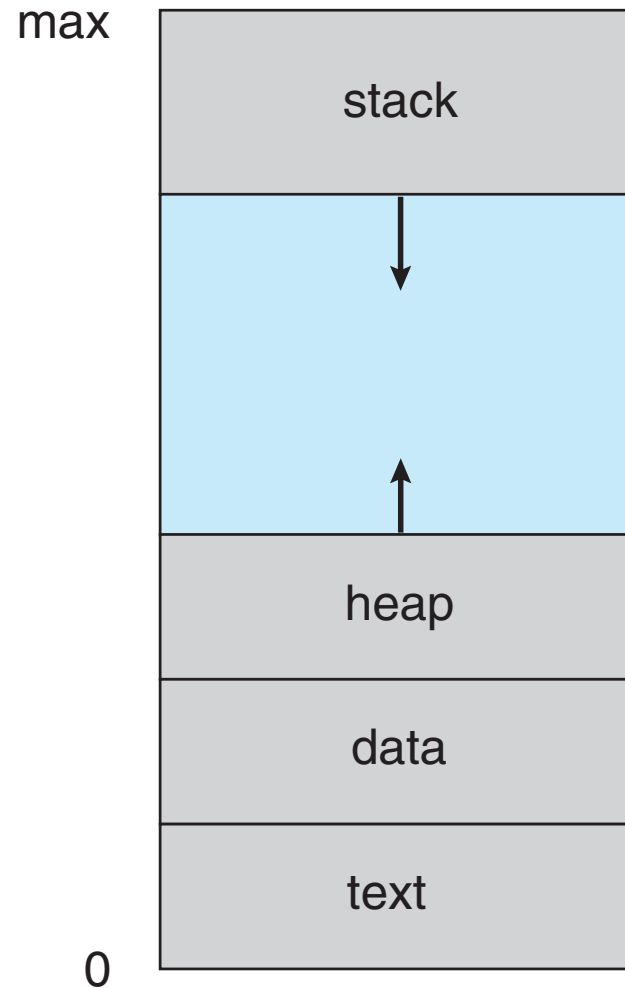
Bibliografía para el Tema 2

- Silberschatz, 7ª edición en español:
 - **Capítulo 3 (procesos):**
3.1, 3.2, 3.3
(3.4 en adelante tratan IPC, no entra en FSO)
 - **Capítulo 4 (hebras/hilos):**
entero, especial atención a 4.1 y 4.3
 - **Capítulo 5 (planificación de la CPU):**
entero, excepto 5.7

¿Qué es un proceso?

- Un **proceso** es un programa en ejecución.
- Un proceso necesita recursos para poder ejecutarse: memoria, dispositivos de E/S, la CPU, etc.
- Áreas típicas de la memoria:
 - código (también llamado «texto»)
 - datos (variables globales + memoria dinámica)
 - pila (datos locales de subrutinas)

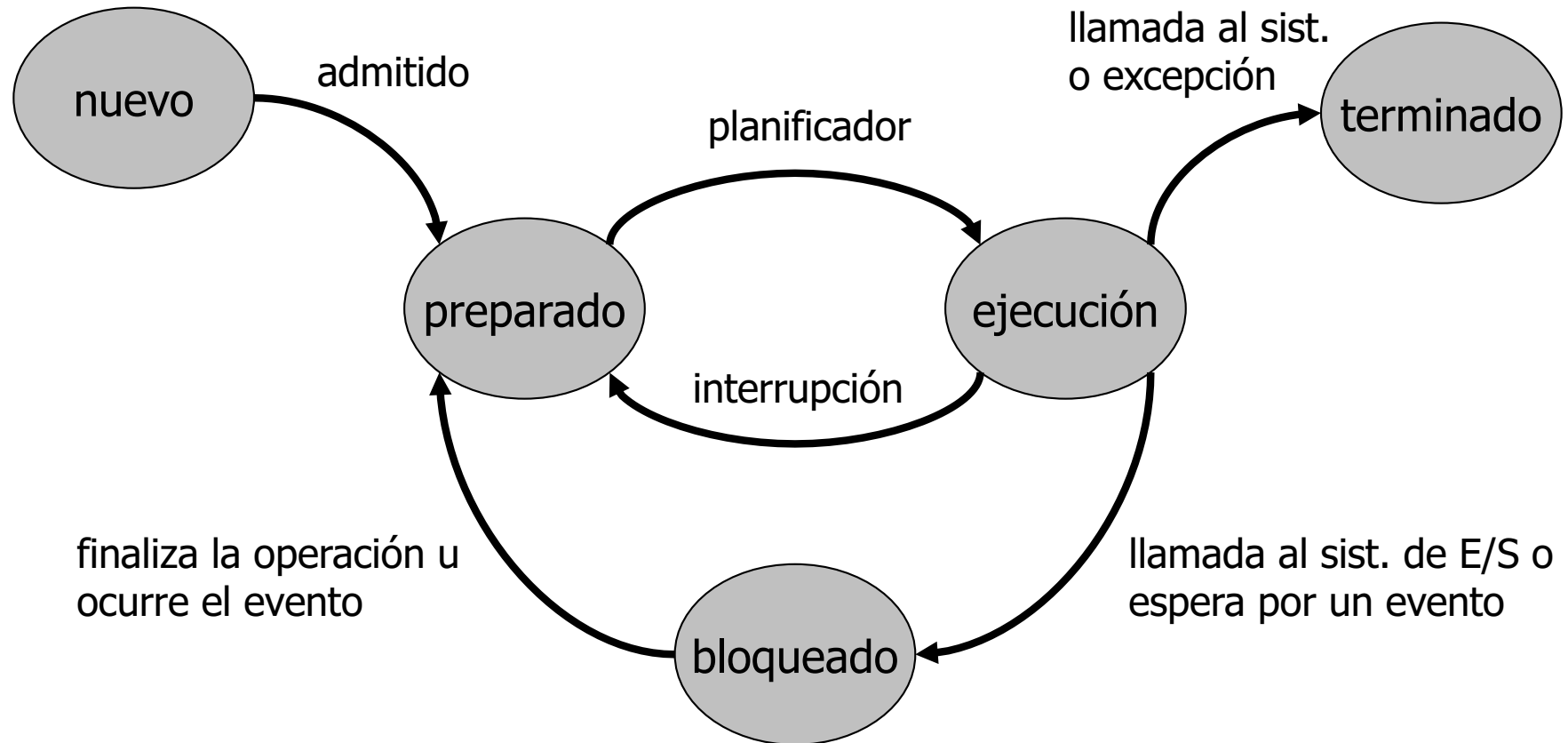
Organización de la memoria de un proceso (Unix, Windows)



Estados de un proceso

- A medida que un proceso se ejecuta, cambia de estado:
 - **nuevo**: recién creado por el S.O.
 - **en ejecución**: está en la CPU ejecutando instrucciones
 - **bloqueado**: esperando a que ocurra algún evento (ej. una operación de E/S)
 - **preparado**: esperando a que le asignen un procesador
 - **terminado**: no ejecutará más instrucciones y el S.O. le retirará los recursos que consume

Estados de un proceso



Bloque de control de proceso (BCP)

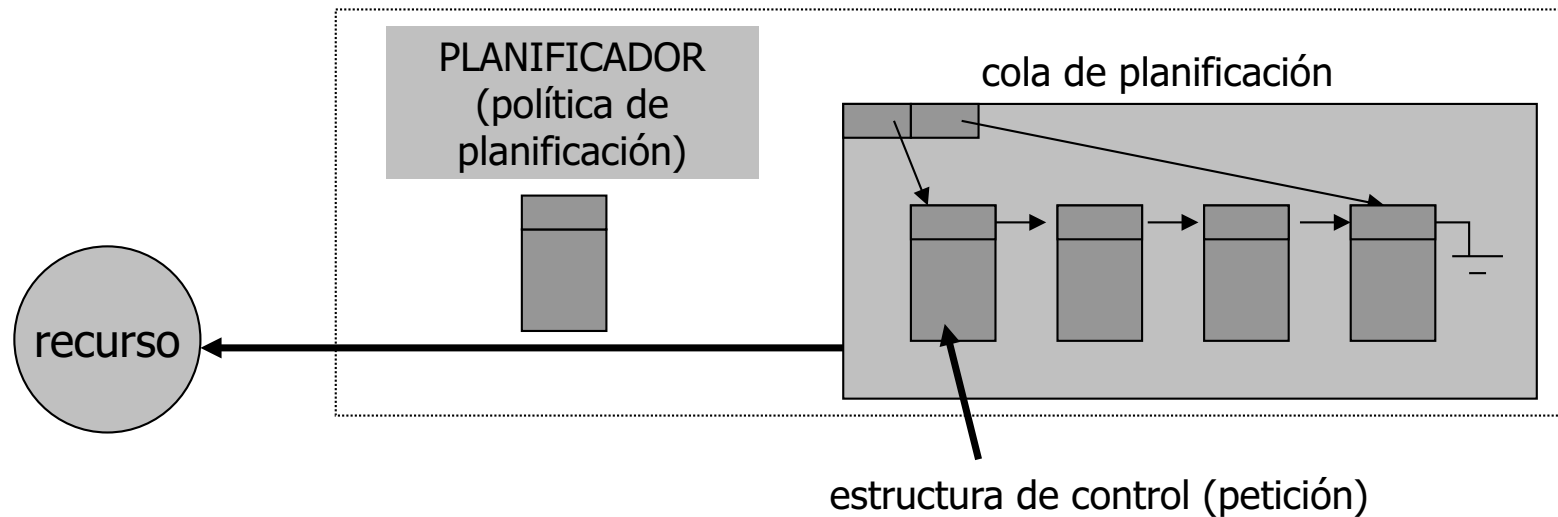
- Cada proceso debe tener una estructura de datos que almacena su estado y otra información de control:
 - Valores de los registros de la CPU
 - Estado actual (preparado, bloqueado, etc.)
 - Información para el planificador (prioridad, tiempo de espera...)
 - apuntadores a los recursos de memoria, E/S, archivos etc. que tiene concedidos o abiertos
 - Información de contabilidad (tiempo consumido...)

Planificación de la CPU (*CPU scheduling*)

- ¿Qué proceso utiliza la CPU en cada momento?
- Objetivos diferentes según el tipo de sistema:
 - Multiprogramación básica → sacar rendimiento al procesador
 - Tiempo compartido → garantizar tiempos de respuesta cortos
 - Tiempo real → garantizar plazos de ejecución
 - ...

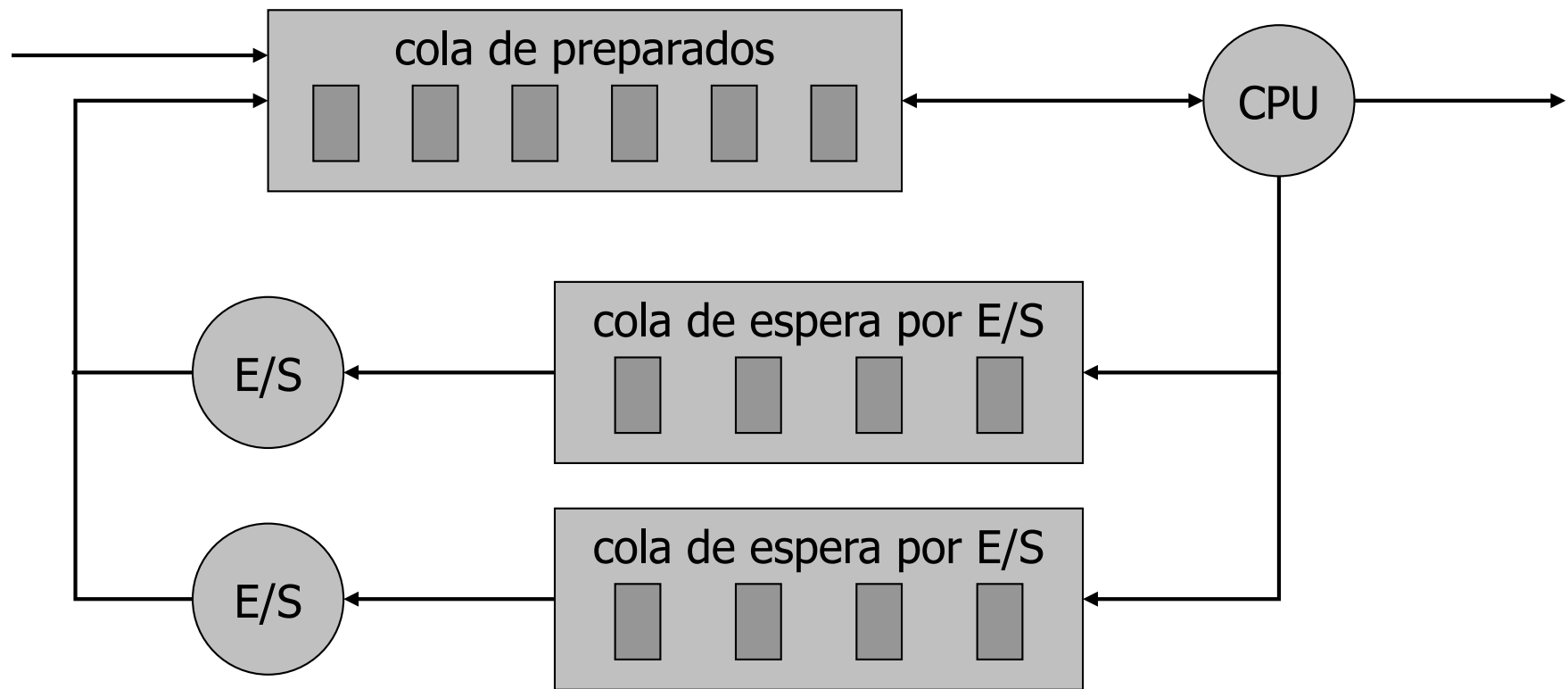
Cola de planificación

- Conjunto de procesos en espera por la utilización de un determinado recurso
- Implementación: tabla, lista enlazada, árbol...
- El planificador inserta y extrae procesos de la cola de acuerdo con la política implementada



Colas de procesos

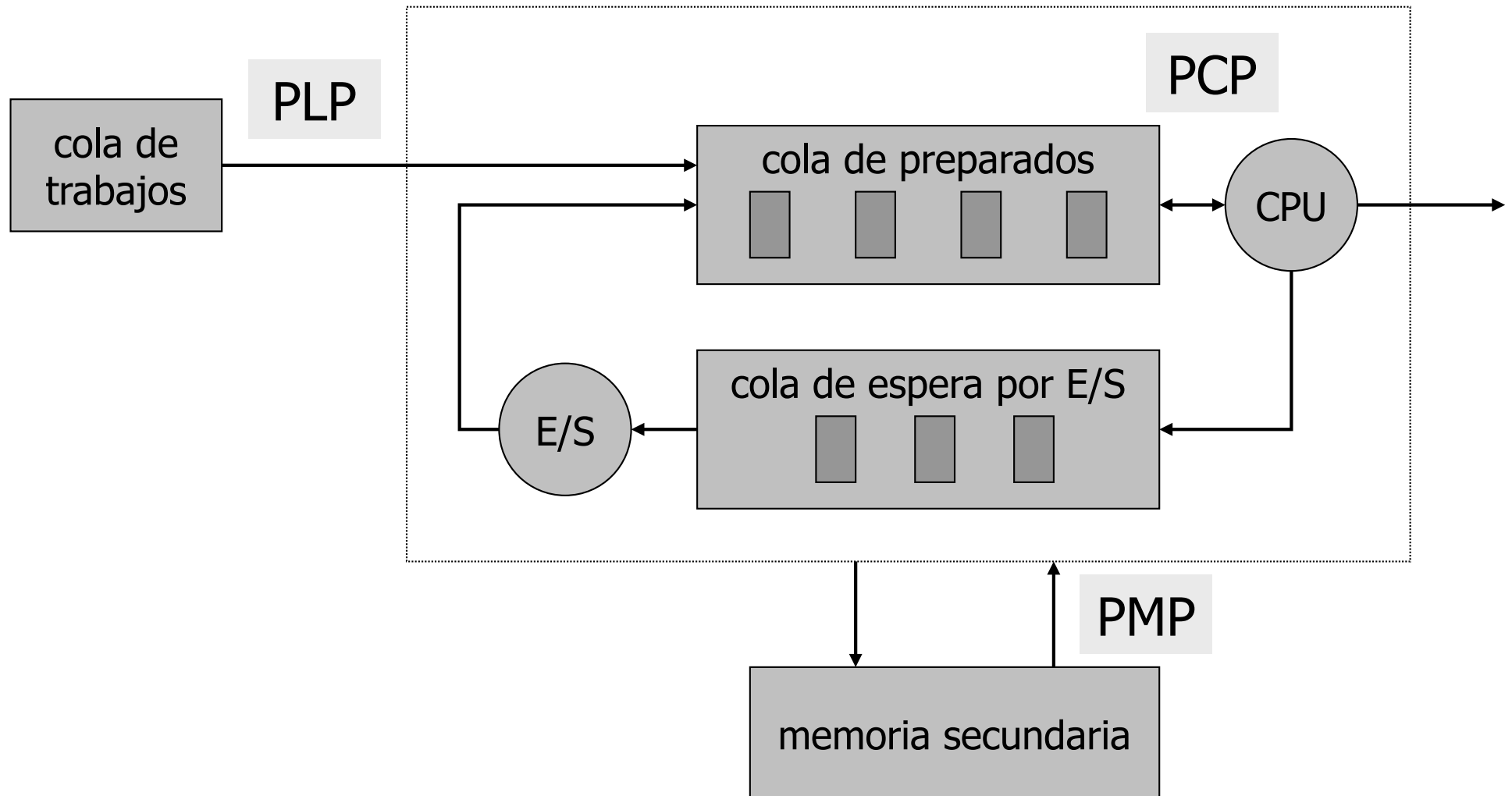
- El SO organiza los BCP en colas de espera por el procesador o por los dispositivos de E/S. (colas de planificación: cola de procesos, colas de dispositivos)



Niveles de planificación

- El **planificador de corto plazo** o de bajo nivel es el que asigna y desasigna la CPU.
- En los sistemas por lotes, existe un **planificador de largo plazo (PLP)** o de alto nivel, que suministra procesos a la cola de preparados.
- El PLP trata de conseguir una mezcla adecuada de trabajos intensivos en CPU y en E/S. Se ejecuta con poca frecuencia.
- **Planificador de medio plazo (*swapper*)**. Envía al disco procesos bloqueados, para liberar memoria principal a los otros procesos → **intercambio (*swapping*)**.

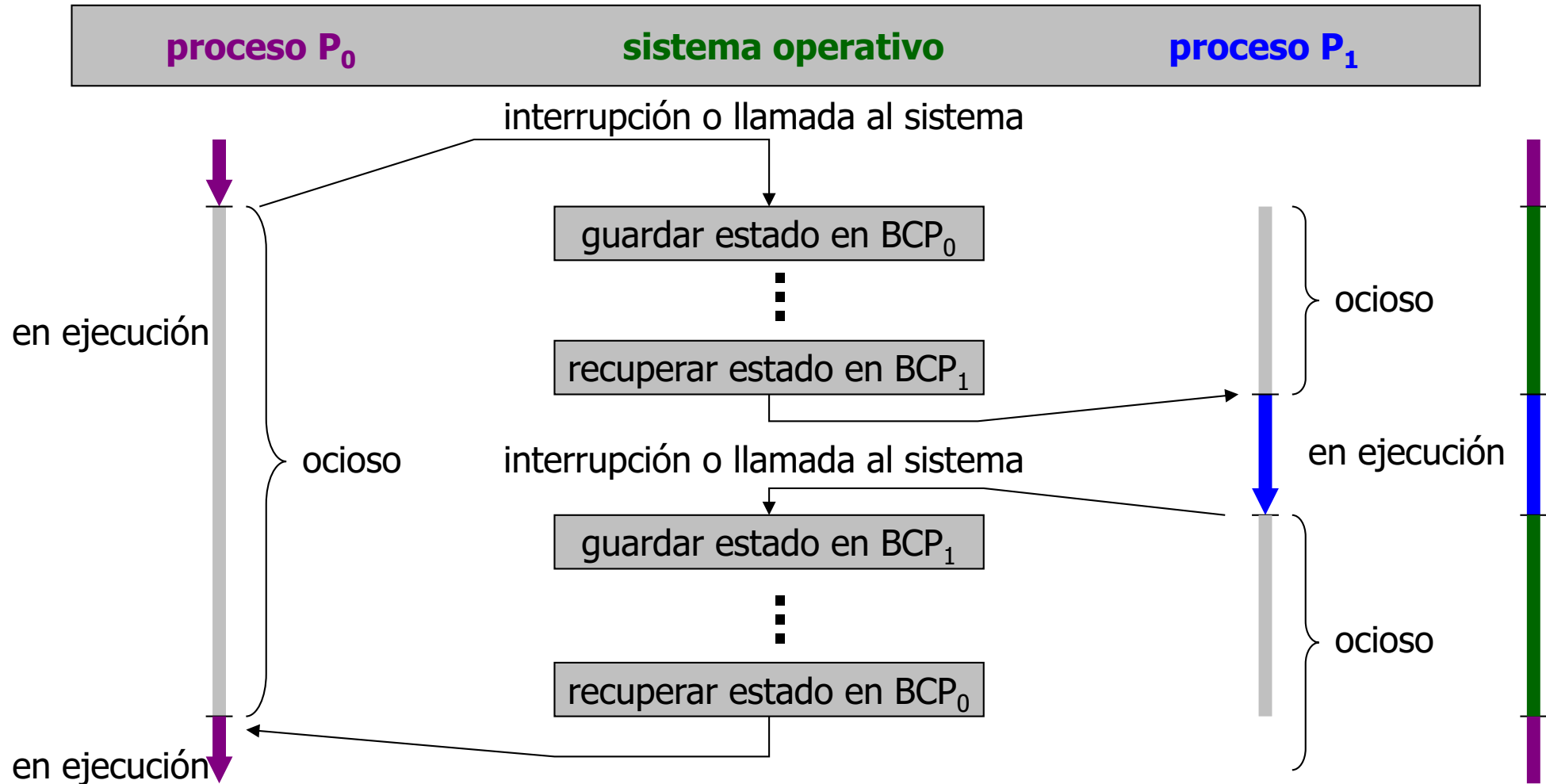
Niveles de planificación



Cambio de contexto (*context switch*)

- Es la operación que consiste en desalojar a un proceso de la CPU y reanudar otro.
- El módulo del SO encargado de esta acción se suele llamar **despachador** (*dispatcher*).
- Hay que guardar el estado del proceso que sale en su BCP, y recuperar los registros del proceso que entra. Cada contexto se encuentra en cada BCP.
- El cambio de contexto es *tiempo perdido*, así que debe ser lo más rápido posible.
- El hardware a veces ofrece mecanismos para reducir el tiempo del cambio de contexto (ej. en x86, instrucción PUSHA = guarda todos los registros; bancos de registros alternativos; etc.).

Cambio de contexto



API PARA TRABAJAR CON PROCESOS

¿Qué ofrece una API de procesos?

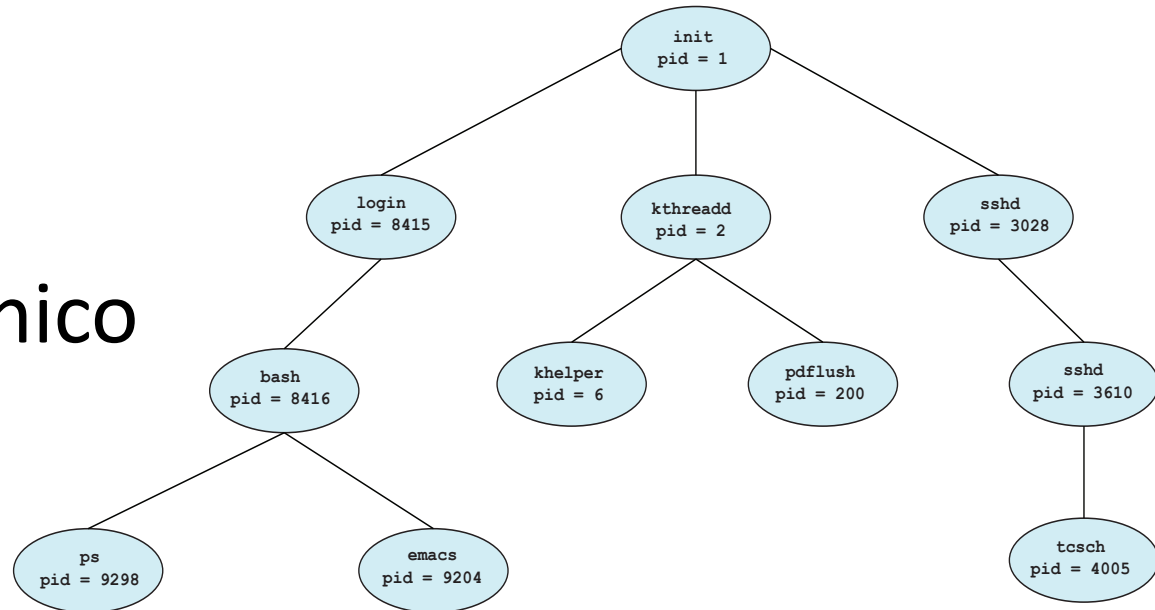
- Creación de nuevos procesos
- Terminación de procesos
- Comunicación entre procesos (IPC)

Creación de procesos

- Todo proceso se crea mediante una llamada al sistema ejecutada por otro proceso.
- Al creador se le llama **proceso padre** y al nuevo, **proceso hijo**.
- Relación entre el padre y el hijo:
 - ¿ejecutan el mismo código?
 - ¿utilizan la misma área de memoria?
 - ¿comparten archivos abiertos y otros recursos?
 - cuando el padre termina, ¿el hijo muere con él?

Identificadores y árboles de procesos

- Cada proceso recibe un identificador único (**PID** en Unix y Windows)
- Los padres e hijos van formando un **árbol de procesos**



Ejemplos de llamadas al sistema

- Windows:
 - **CreateProcess()**: se indica el fichero ejecutable donde está el código del hijo
- UNIX:
 - **fork()**: crea un proceso hijo que es un duplicado del padre
 - **exec()**: sustituye el código por un nuevo fichero ejecutable (no crea un nuevo proceso)

Ejemplo de fork() y exec()

```
pid_t pid = fork();
if ( pid!=0 ) {
    puts ("Soy el proceso padre");
    // otras acciones del proceso padre
}
else { // el hijo entrará por aquí
    puts ("Soy el proceso hijo");
    // el hijo ejecuta la orden "ls -l"
    // que reemplaza el código actual del hijo
    execlp ("ls", "ls", "-l", NULL);
}
```

Terminación de procesos

- Un proceso termina cuando invoca a una llamada al sistema específica, ej. **exit()**
- También si se genera una excepción y el S.O. decide abortarlo.
- En UNIX, cuando un proceso termina, con él mueren sus descendientes.
- Podría existir una llamada al sistema para abortar otro proceso, ej. **kill()**
- Esperar por la terminación de un proceso hijo: **wait()**

Ejemplo de sincronización con wait()

```
pid_t pid = fork();
if ( pid!=0 ) {
    puts ("Soy el padre, pid=%d\n", getpid());
    // ... otras acciones del padre
    int status;
    pid_t hijo = wait(&status); // espera por el hijo
    // ... el padre puede continuar
}
else {
    printf("Soy el hijo, pid=%d\n", getpid());
    // ... otras acciones del hijo
    exit(1234);
}
```

Comunicación entre procesos (IPC)

- ¿Cómo pueden intercambiar datos varios procesos? Dos modelos:
 - Memoria compartida
 - Paso de mensajes
- Mecanismos clásicos (Unix, Windows):
 - Ficheros normales (ineficiente)
 - Zonas de memoria compartida
 - Tuberías (*pipes*)
 - Sockets
 - Llamadas a procedimiento remoto (RPC)

HILOS (THREADS)

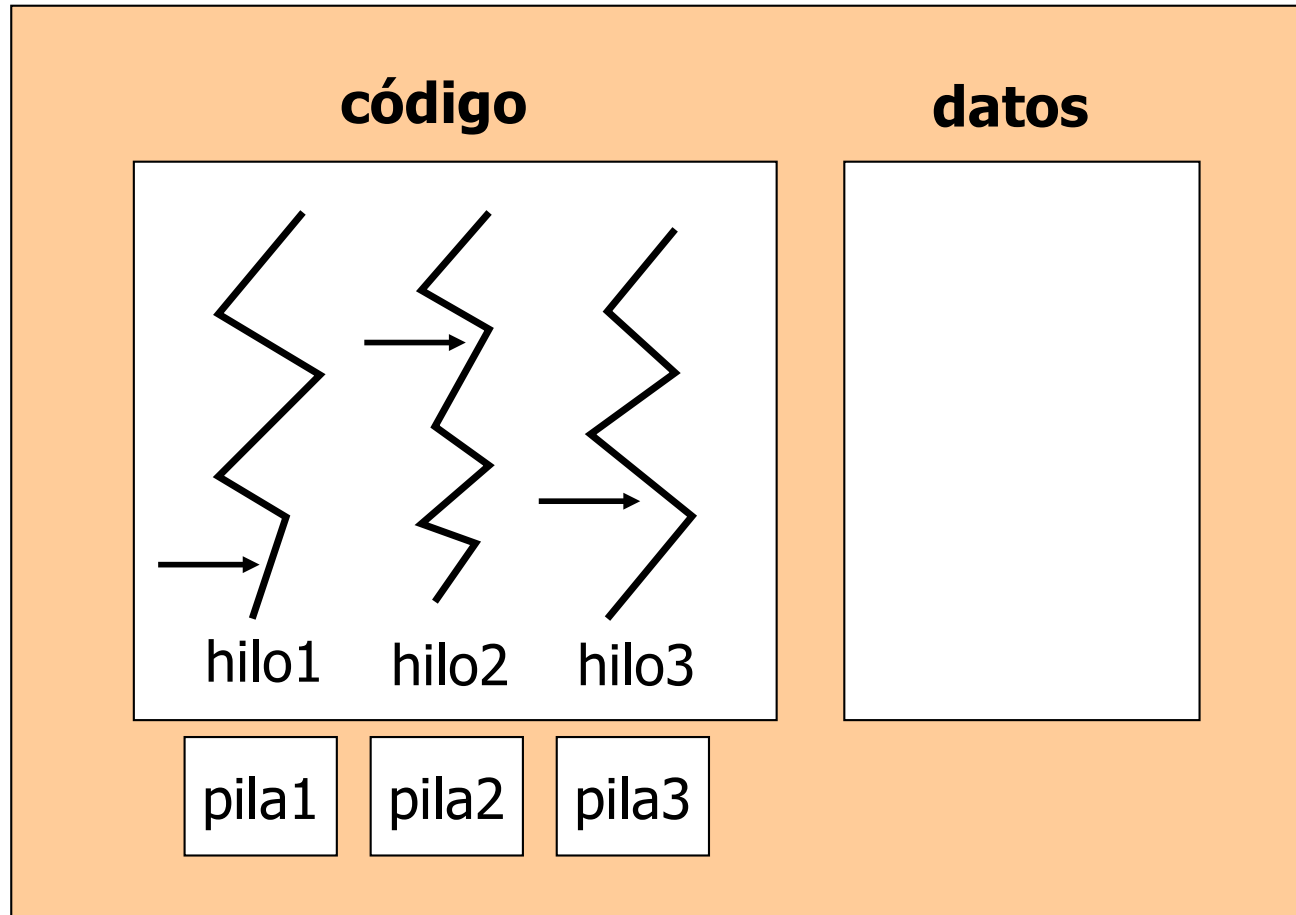
Hilos (*threads*)

- Un **hilo**, **hebra** o **proceso ligero** es una unidad básica de ejecución, con su propio:
 - contador de programa
 - registros de CPU
 - pila (stack)
- Los hilos dentro de una misma aplicación comparten:
 - código y datos
 - recursos del S.O. (ficheros, E/S, etc.)
- NOTA: *thread* se pronuncia «**zred**», NO «zrid»

Procesos pesados, procesos ligeros

- Un **proceso pesado** contiene uno o varios hilos que comparten la misma memoria y recursos.
 - Proceso pesado = aplicación
 - Hilos = actividades concurrentes dentro de la aplicación
- Hilos = **procesos ligeros** (*lightweight processes*)

La memoria en una aplicación multihilo



Proceso pesado

Beneficios de los hilos

- Ventajas de N hilos en un único proceso pesado, respecto a N procesos pesados:
 - Comunicación más sencilla entre los procesos (memoria compartida)
 - Más concurrencia con menos recursos (usan el mismo espacio de memoria para código y datos)
 - La conmutación entre hilos puede ser más eficiente (ej. al cambiar de hilo no hay que limpiar cachés y tablas de páginas)

Hilos a nivel de usuario / de núcleo (*kernel threads / user threads*)

- Algunos sistemas manejan los hilos fuera del núcleo, dentro de la propia aplicación → **hilos de usuario**
 - Java JVM clásica, POSIX threads, etc.
- **Hilos de núcleo** → el propio núcleo es el que soporta los hilos
 - Actualmente, es el caso más frecuente
- En un sistema con hilos de usuario, el núcleo no «sabe» que un proceso tiene concurrencia interna
 - si un hilo queda bloqueado, se bloquea a todo el proceso pesado

API de hilos: pthreads (Unix)

```
#include <pthread.h>

void* rutina1 ( void* arg ) { /* acciones que realizará un hilo */ }
void* rutina2 ( void* arg ) { /* acciones que realizará otro hilo */ }

int main()
{
    // Declara dos objetos de tipo hilo
    pthread_t un_hilo, otro_hilo;

    // Lanza un hilo concurrente que ejecuta: rutina1(NULL)
    pthread_create ( &un_hilo, NULL, rutina1, NULL);

    // Lanza otro hilo que ejecuta: rutina2(NULL)
    pthread_create ( &otro_hilo, NULL, rutina2, NULL);

    // El hilo principal espera a que finalicen los otros dos hilos
    void* dummy; // no se usa en este ejemplo
    pthread_join ( un_hilo, &dummy );
    pthread_join ( otro_hilo, &dummy );
    exit(0);
}
```

API de hilos: Java

```
// Método uno: implementar la interfaz Runnable
public class UnTipoDeHilo implements Runnable {
    public void run() { /* acciones del hilo */ }
}

// Método dos: heredar de la clase Thread
class OtroTipoDeHilo extends Thread {
    public void run() { /* acciones del hilo */ }
}

public static void main(String args[]) {
    Thread t1 = new Thread(new UnTipoDeHilo);
    t1.start();

    Thread t2 = new OtroTipoDeHilo();
    t2.start();

    // espera a que acaben los hilos
    t1.join();
    t2.join();
}
```


Hilos implícitos / automáticos

- Objetivo: aliviar al programador del manejo de hilos
- Compiladores, bibliotecas, lenguajes de programación... que tratan de crear y gestionar hilos automáticamente a partir de indicaciones abstractas del programador
- OpenMP, GCD (Apple), MTTD (Microsoft)...

Hilos implícitos / automáticos

- Lenguajes con marcas para identificar zonas concurrentes
 - OpenMP, go (Google), GCD (Apple)...
 - El entorno de tiempo de ejecución decide si se crea o no un hilo, dependiendo de los recursos disponibles;
 - O bien mantiene un «pool de hilos» de tamaño fijo y a medida que se quedan libres hilos del *pool* se van ejecutando los bloques concurrentes

Fundamentos de los Sistemas Operativos

Tema 2. Procesos FIN DEL PRIMER BLOQUE

© 1998-2015 José Miguel Santos - Alexis Quesada - Francisco Santana