Sistemas Operativos

Tema 3 – Procesos

http://www.ditec.um.es/so

Departamento de Ingenieria y Tecnologia de Computadores

Universidad de Murcia

Índice

Índice

1.1 Concepto de Proceso

- La CPU es un recurso importantísimo
  - Más procesos que CPUs $\Rightarrow$ ¿qué proceso la usa?
  - Ocurre igual con otros recursos limitados (p.ej. la memoria)
- Distintos usuarios realizan tareas concurrentemente
- Alternancia rápida entre procesos: **cambio de contexto** $\Rightarrow$ **Pseudoparalelismo**
- **Modelo de procesos**: Aparentemente secuenciales, ocultan la dificultad de las interrupciones
- Es la unidad para describir las tareas de cada usuario y para asignar recursos
- Recordemos: **proceso**, dinámico (PC, pila, registros, variables...); **programa**, estático

1.1 Concepto de Proceso (ii)

- ¿Qué proceso usa la CPU?
  - Algoritmo de planificación
- Ventajas de tener varios procesos en ejecución:
  - compartir recursos soft y hard
  - acelerar cálculos con varias CPUs
  - modularidad
  - comodidad
  - etc.

1.2 Creación de Procesos

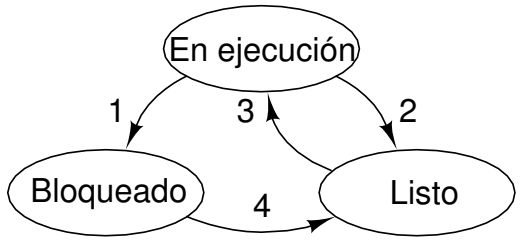
- Necesario un mecanismo de creación y terminación de procesos
- Creación de procesos:
  - Inicio del sistema
    - Procesos interactivos *vs*. demonios (*daemons*)
  - Llamada al sistema para crear procesos
    - UNIX `fork+exec`, Win32 `CreateProcess`
  - El usuario inicia un proceso
    - Ya sea en modo gráfico o en algún *shell*
  - Inicio de un trabajo por lotes
    - Normalmente en sistemas de *mainframe*

1.2 Terminación de Procesos

- Un proceso puede terminar por varias razones:
  - Terminación normal (voluntaria)
    - 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)
  - Terminado por otro proceso (involuntaria)
    - UNIX `kill`, Win32 `TerminateProcess`
- **Jerarquía de procesos**
  - Unos procesos crean a otros procesos
  - Se puede ver como una jerarquía de procesos

1.3 Estados de un proceso

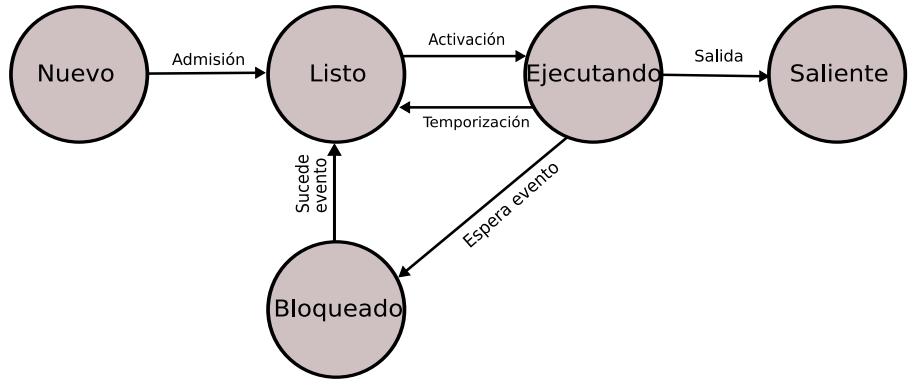
- Estados de un proceso
  - **En ejecución**: tiene la CPU
  - **Listo**: detenido, otro tiene la CPU
  - **Bloqueado**: no se puede ejecutar, espera algún evento



1. El proceso se bloquea en espera de datos
2. El planificador elige otro proceso
3. El planificador elige a este proceso
4. Datos disponibles

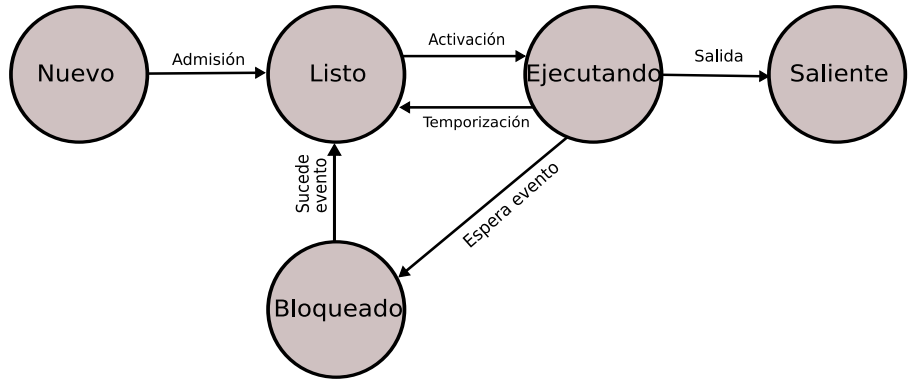
- Se puede incluir la posibilidad de **suspender** y **reanudar** procesos
  - Aliviar la carga temporalmente
  - Depuración

1.3 Estados de un proceso (i)



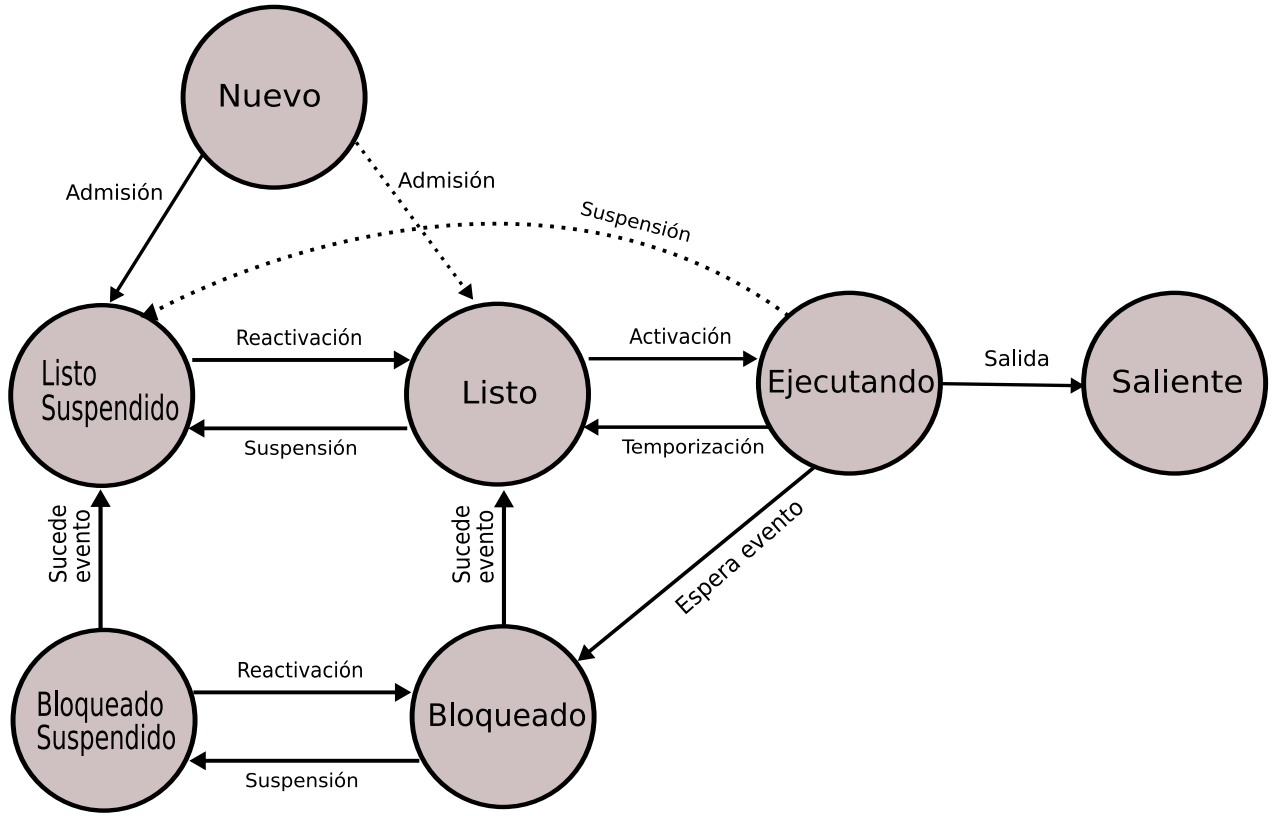
- **Nuevo**. Se acaba de crear y aunque tiene el BCP, no ha sido cargado en memoria
- **Ejecutando**.
- **Listo**. Está preparado para ejecutarse cuando haya oportunidad
- **Bloqueado**. No se puede ejecutar hasta que se cumpla un evento determinado o se complete una operación de E/S
- **Saliente**. Se ha quitado del grupo de procesos ejecutables porque ha sido abortado por alguna razón

1.3 Estados de un proceso (ii)



- **Nuevo** $\Rightarrow$ **Listo**. Se intenta que no haya demasiados procesos activos
- **Listo** $\Rightarrow$ **Ejecutando**. Se selecciona un proceso Listo y se le asigna la CPU
- **Ejecutando** $\Rightarrow$ **Saliente**. El proceso finaliza por cualquier motivo
- **Ejecutando** $\Rightarrow$ **Listo**. Acaba el quantum o en algunos SSOO se desbloquea un proceso prioritario
- **Ejecutando** $\Rightarrow$ **Bloqueado**. Solicita algo por lo que debe esperar
- **Bloqueado** $\Rightarrow$ **Listo**. Sucede lo que estaba esperando
- **Listo/Bloqueado** $\Rightarrow$ **Saliente**. No se muestra. A veces un padre puede terminar la ejecución de un hijo alguna razón

1.3 Estados de un proceso (iii)



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

1.3 Estados de un proceso (iv)

- **Bloqueado** $\Rightarrow$ **Bloqueado suspendido**. Si no hay procesos listos, un proceso bloqueado se pasa a disco para hacer hueco en memoria o para liberar memoria.
- **Bloqueado suspendido** $\Rightarrow$ **Listo suspendido**. Sucede lo que estaba esperando.
- **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.
- **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 cuando hay muchos bloqueados, o inmediatamente.
- **Bloqueado suspendido** $\Rightarrow$ **Bloqueado**. Un proceso termina liberando memoria, uno de los bloqueados susp. tiene la mayor prioridad y el SO sospecha que el evento que espera sucederá en breve.
- **Ejecutando** $\Rightarrow$ **Listo/Suspendido**. Un proceso de mayor prioridad despierta de bloqueado suspendido.
- **Cualquier estado** $\Rightarrow$ **Saliente**.

1.4 Descripción de Procesos (Stallings, 3.2)

- Para administrar los procesos se usa una **tabla de procesos.**
- Cada entrada de la tabla es un **PCB** (Process Control Block)

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

1.5 Control de Procesos (i)

- Modos de ejecución
  - Modo usuario
  - Modo núcleo (root, protegido, supervisor...)
  - El proceso está compuesto por la parte usuario (lo que el usuario implementa) y la parte de núcleo (las rutinas del núcleo que utiliza)
- ¿Cómo se crea un proceso?
  - Asignar un ID único al proceso (y entrada en la tabla de procesos)
  - Asignar espacio para el proceso
  - Iniciar el PCB
  - Incluir al proceso en los algoritmos de planificación
  - Actualizar otras tablas (p.ej. contabilidad)

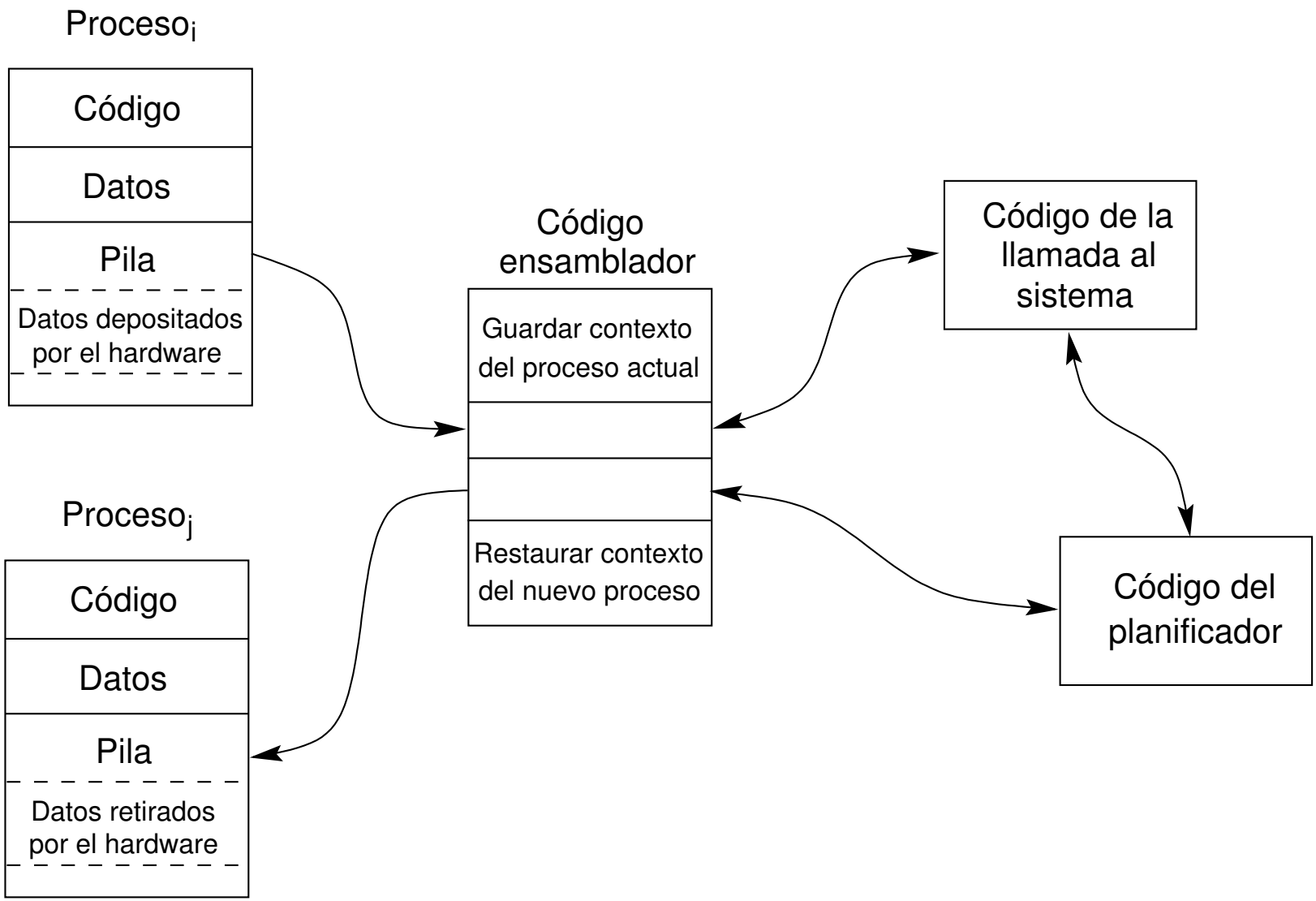
1.5 Control de Procesos (ii)

- ¿Cuándo cambiar de proceso?
  - Interrupción de reloj
  - Interrupción de E/S
  - Llamada al sistema
  - Fallo de memoria (fallo de memoria virtual)
- Todo a través de interrupciones (hw. o sw.)
- No es lo mismo un cambio de proceso que un cambio de modo
  - Cambiar de modo: guarda registos, cambia pila, memoria virtual...
  - Cambiar de proceso: actualizar BCP, mover entre colas, seleccionar nuevo proceso, activarlo...

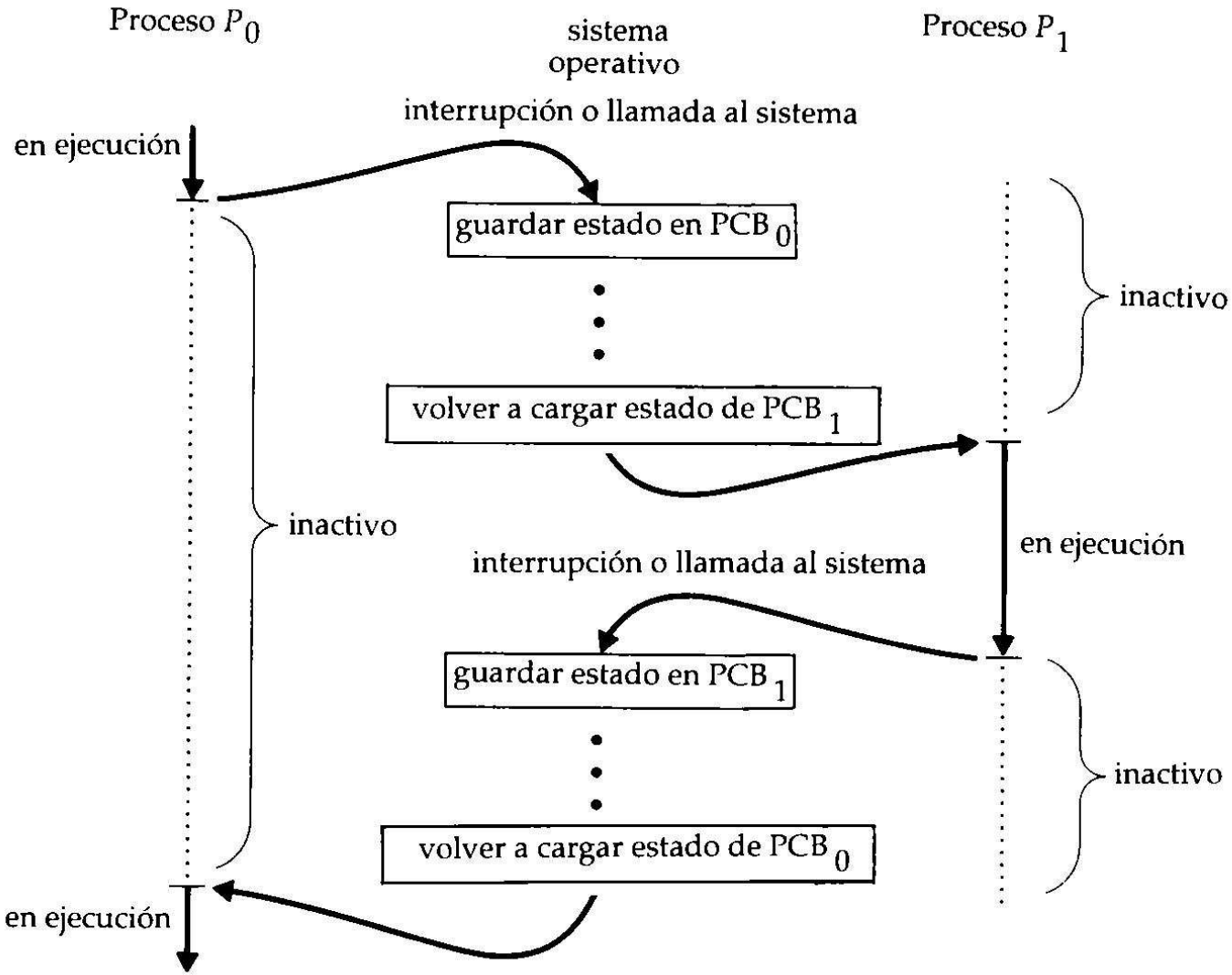
1.5 Control de Procesos (iii)

- Mecanismo de cambio de modo
  1. El PC pasa a apuntar a la rutina de tratamiento de la excepción
  2. Cambia de modo usuario a modo núcleo
  3. El SO guarda el contexto del proceso interrumpido en su BCP
- Mecanismo de cambio de proceso
  1. Salvamos el estado del procesador (PC, registros...)
  2. Actualizar campos BCP (estado, contabilidad, auditoría...)
  3. Mover el BCP a la cola apropiada (Listo, bloqueado..)
  4. Selección del nuevo proceso a ejecutar
  5. Actualizar el BCP del proceso elegido (estado Ejecutando...)
  6. Actualizar estructuras de datos de gestión de memoria
  7. Restaurar el estado del procesador cuando se interrumpió el nuevo proceso

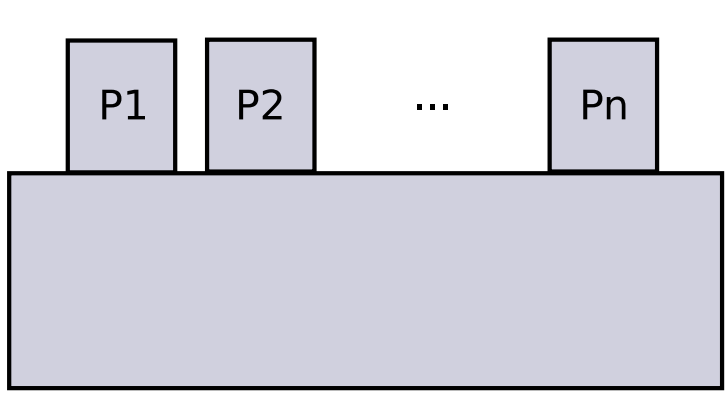
Cambio de contexto (i)

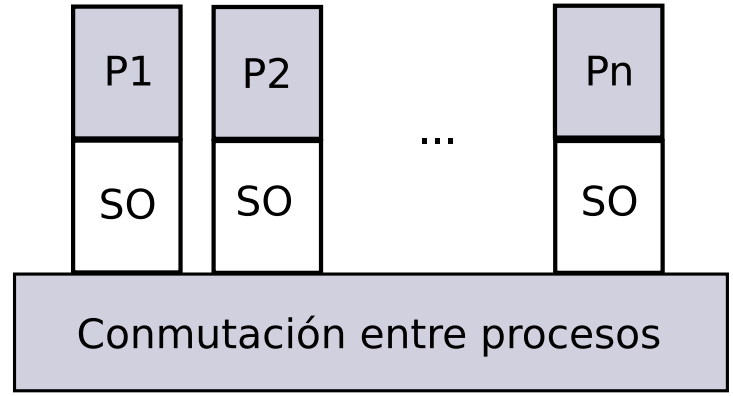


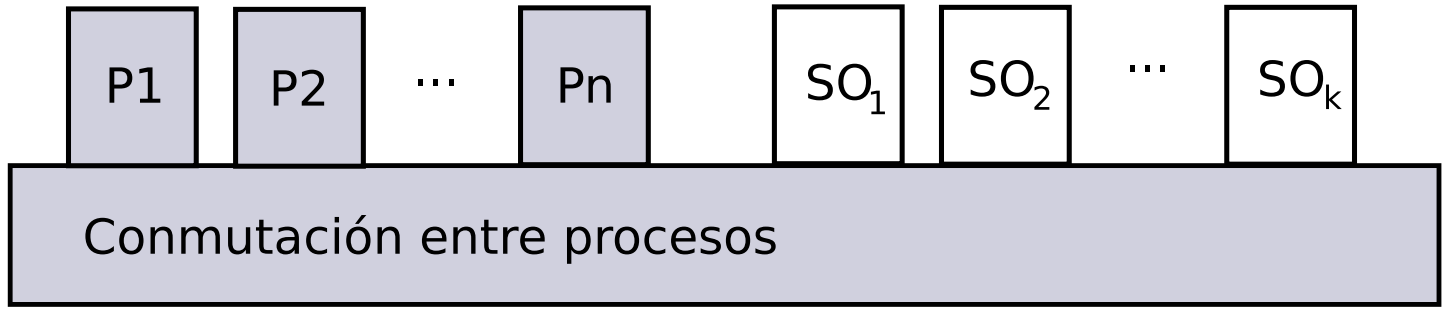
Cambio de contexto (ii)



¿Dónde está el SO?

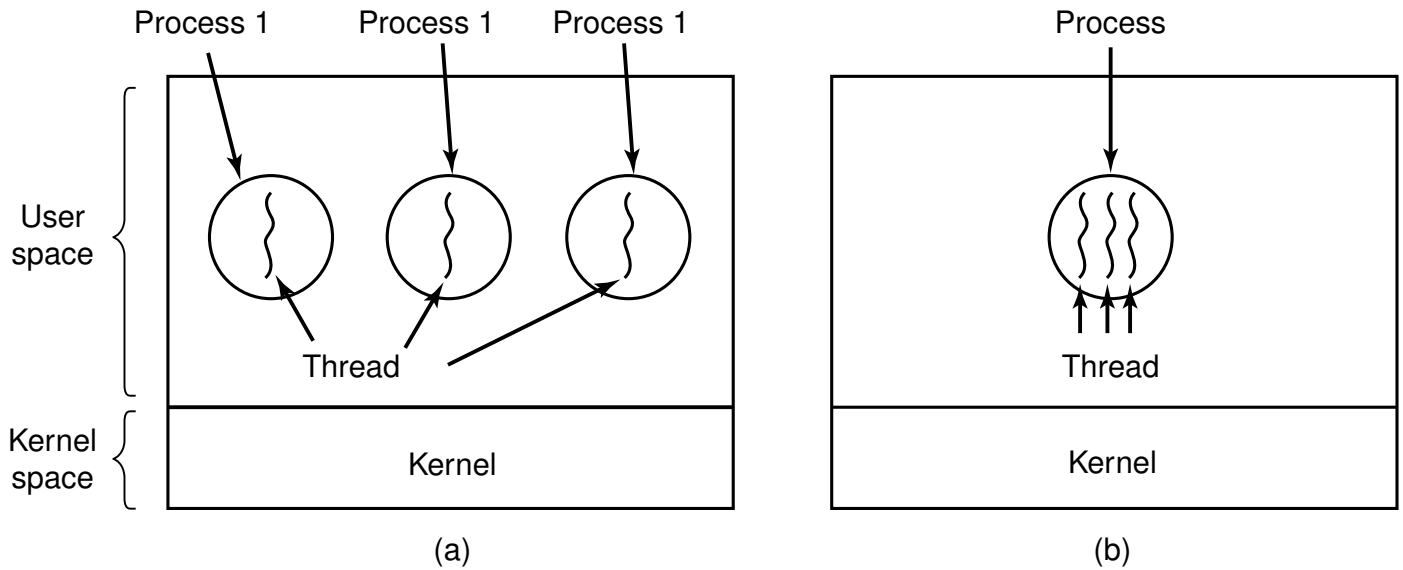






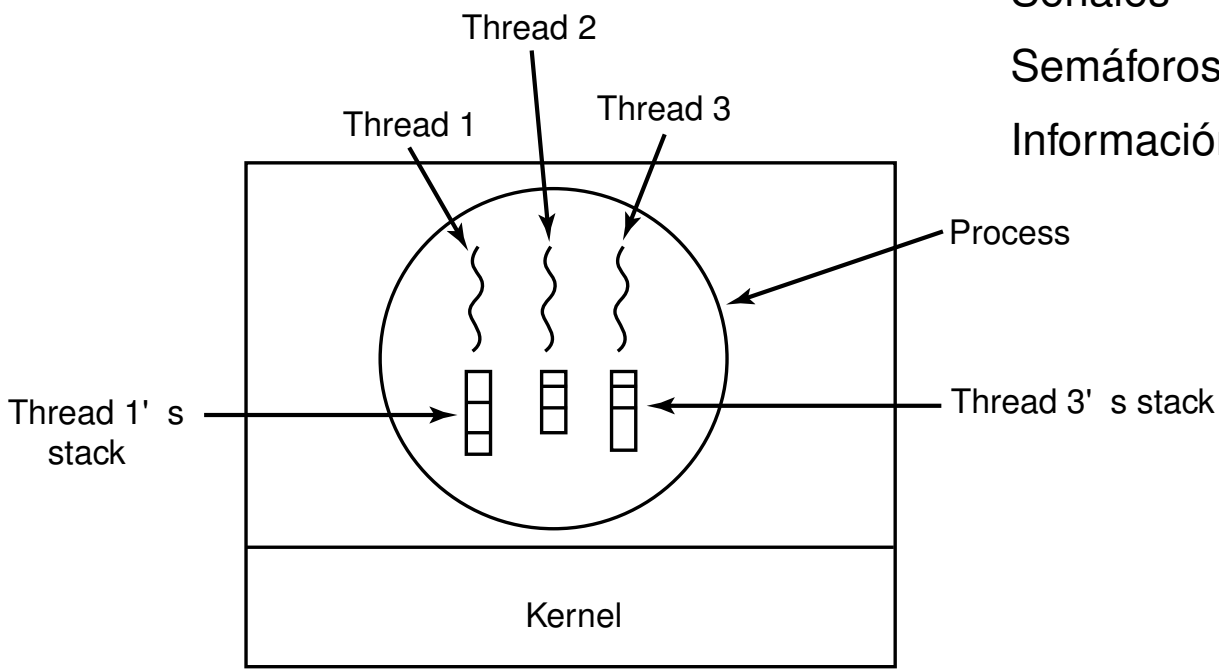
2. Hilos, Hebras o *Threads*

- Hasta ahora, un proceso es, a la vez:
  - **Unidad de asignación de recursos**
  - **Unidad de planificación y ejecución**
- Aunque no tiene por qué: **hilo** $\Rightarrow$ unidad de planificación, **proceso** $\Rightarrow$ unidad de asign. de recursos
- Un proceso puede tener **varios hilos**



2. Hilos, Hebras o *Threads* (ii)

Elementos por hilo	Elementos por proceso
Contador de programa	Espacio de direcciones
Pila	Variables globales
Estado + contexto	Ficheros abiertos
Memoria privada (var. locales)	Procesos hijos
	Cronómetros
	Señales
	Semáforos
	Información contable



2. ¿Para qué quiero hilos?

- Descomponer la aplicación en múltiples subprocesos secuenciales paralelos
- Permiten solapar E/S y cómputo dentro de un mismo proceso
- Con múltiples CPUs se puede conseguir verdadero paralelismo
- Son más rápidos de crear y destruir
- ¿Cómo harías sin hilos...?
  - Procesador de textos: guardar automáticamente, operaciones periódicas...
  - Hoja de cálculo: Recálculo de las celdas.
  - Servidor web: Hilo despachador que asigna consultas a hilos trabajadores.
  - Aplicación que procesa gran cantidad de datos: solapamos E/S y cálculo.
  - ¿Se te ocurre alguno más?

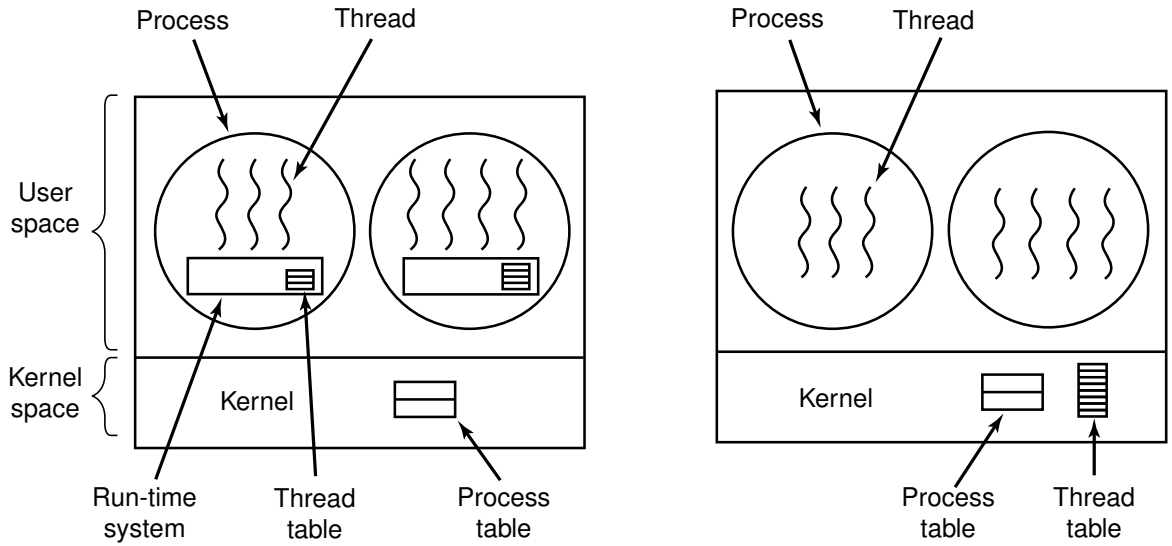
2. Hilos. Características.

- No existe protección (ni se necesita) entre los hilos de un mismo proceso
- Comparten variables globales $\Rightarrow$ **Sincronización**
- Mejoran el rendimiento (menos sobrecarga del *kernel*)
  - Creación más rápida
  - Comunicación más eficiente
- Soportados por el S.O (modo núcleo) o por bibliotecas (modo usuario)

2. Hilos en modo usuario.

- Ventajas
  - El nucleo no sabe que existen
  - Tabla de subprocesos privada para cambios de contexto
  - Cambio de contexto mucho más rápido entre hilos (no se pasa al kernel)
  - Cada proceso puede tener su algoritmo de planificación
- Inconvenientes
  - Llamadas bloqueantes al sistema $\Rightarrow$ funciones no bloqueantes
  - Fallos de página
  - Tienen que ceder la CPU entre ellos $\Rightarrow$ Conmutación en el mismo proceso
  - Precisamente queremos hilos en procesos que tienen mucha E/S para obtener paralelismo, es decir, que se están bloqueando muy frecuentemente.

2. Hilos en modo núcleo.



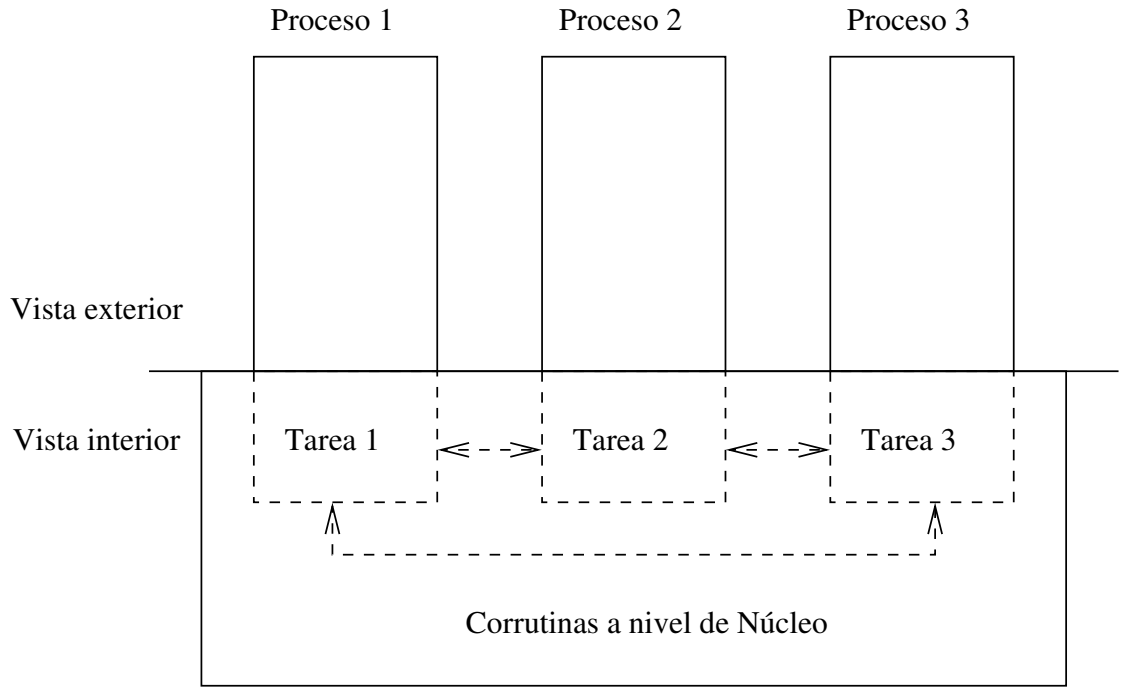
- Ventajas
  - El núcleo mantiene la tabla de hilos, que es un subconjunto de la de procesos
  - Las llamadas bloqueantes no necesitan funciones especiales
  - Los fallos de página no suponen un problema
  - Al bloquearse un hilo, el núcleo puede conmutar a otro hilo de otro proceso
- Inconvenientes
  - Las llamadas bloqueantes son llamadas al sistema, e.d. más costosas
  - La creación y destrucción de procesos es más costoso $\Rightarrow$ Reutilización de hilos

Índice

3. Comunicación entre procesos: sincronización

- Motivación
  - Si el Sistema Operativo protege y aisla los procesos, ¿cómo se sincronizan un conjunto de procesos?
  - El S.O. también provee mecanismos de comunicación y sincronización entre procesos (IPC, *Inter-Process Communication*)
  - Imprescindible en *threads*: Todos los hilos comparten un espacio de memoria $\Rightarrow$ Sincronización
  - Además, todos los hilos (y todos los procesos) hacen llamadas al sistema
    - Cada hilo de ejecución de cada proceso que llama al núcleo, se convierte en un hilo dentro del núcleo
    - Puede haber varias llamadas pendientes en un instante
    - Por ello, necesidad de que el propio núcleo esté diseñado en base a hilos

Hilos dentro del núcleo



- Varios hilos en modo usuario pueden terminar solicitando la misma llamada al sistema, o llamadas al sistema que comparten código en el núcleo
- Resultado: aunque en modo usuario no existan los hilos, en modo núcleo es importante que se diseñe como si los tuviera

3.1 Condiciones de carrera

- Diferentes hilos que acceden a zonas comunes de memoria
- El planificador cambia entre los distintos hilos de forma impredecible
- A veces los hilos tienen que hacer una tarea de forma atómica. Por ejemplo:
  Hilo$_i$:

```
{
   <Acción>
   A = A + 1;
}
```

- Se tienen que ejecutar en exclusión mutua

3.2 Secciones Críticas

- Las Secciones o regiones críticas son trozos de programa que se tienen que ejecutar en **exclusión mutua**
- Para implementarlos:
  - Semáforos (mutex)
  - Monitores
  - Variables Condición

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Hilo$_i$:

```
{
  <Acción>
  pthread_mutex_lock (&mutex);
  /* Sección crítica */
  A = A + 1;
  pthread_mutex_unlock (&mutex);
}
```

3.3 Bloqueos (Stallings, 6.1 y 6.2)

Un bloqueo se produce cuando varios procesos no pueden continuar su ejecución (se bloquean entre ellos mismos) al luchar por algún recurso. Por ejemplo, dos procesos:

```
Proceso P
...
lock(A)
...
lock(B)
...
...
...
unlock(A)
...
unlock(B)
...
```

```
Proceso Q
...
lock(B)
...
lock(A)
...
...
...
unlock(B)
...
unlock(A)
...
```

3.3 Bloqueos (ii)

- Posible esquema de ejecución:
  - P obtiene el candado A
  - Q obtiene el candado B
  - P espera a obtener el mutex B, que lo tiene Q
  - Q espera a obtener el mutex A, que lo tiene P
  - Ninguno puede continuar
- Los sistemas con múltiples procesos/hilos son difíciles de diseñar y tienen que tener en cuenta estas circunstancias

Índice

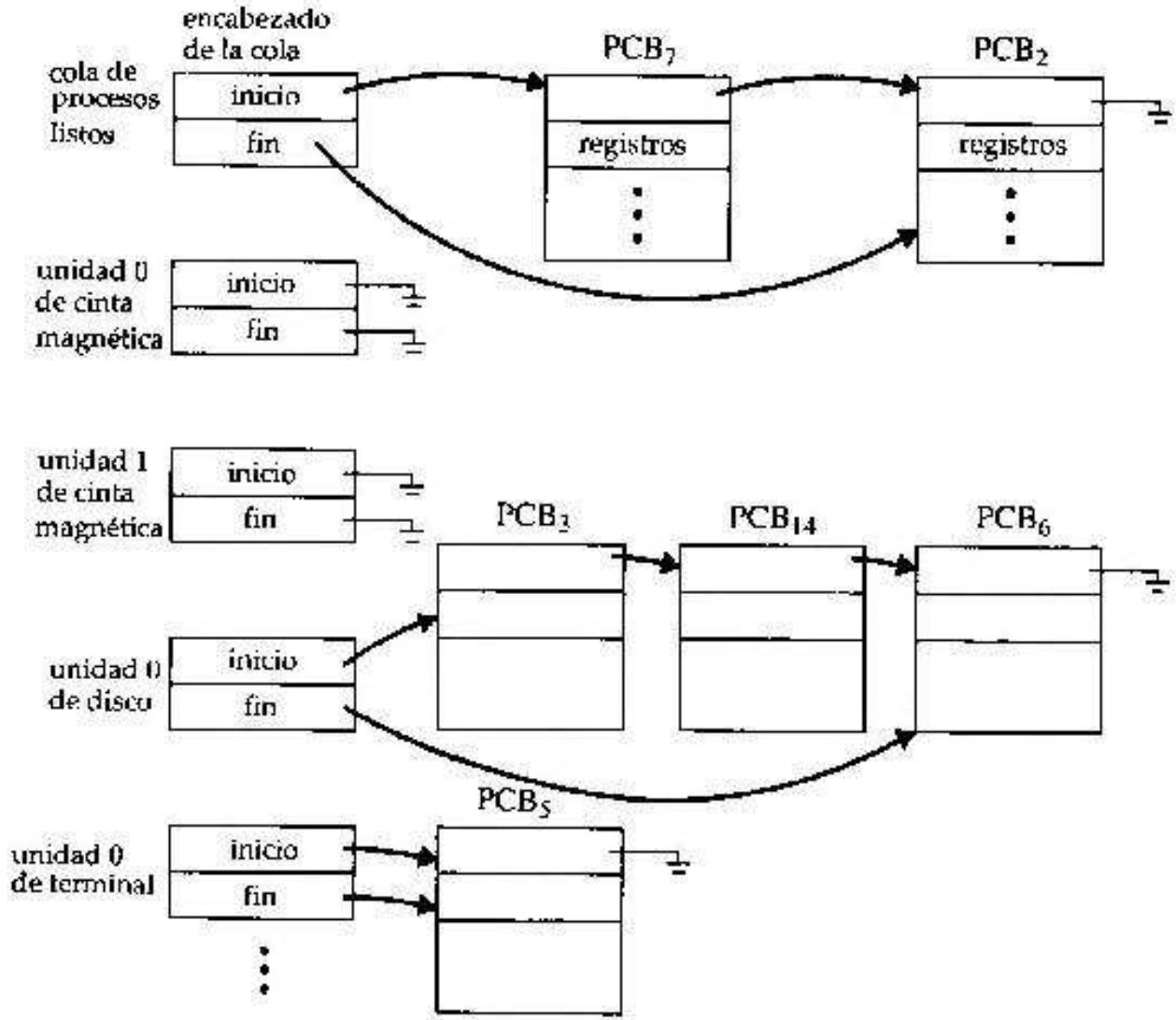
4.1 Planificadores

- En un momento dado puede haber varios procesos listos:
  - La parte del S.O. que decide es el **planificador** siguiendo un **algoritmo de planificación**
  - El **despachador** (*dispatcher*) debe:
    - Cambiar de contexto, cambiar a modo usuario, reiniciar el proceso adecuado en la posición en que quedó
  - Metas:
    - Equidad
    - Eficacia
    - Tiempo de respuesta, tiempo de espera, tiempo de regreso
    - Rendimiento o Productividad

4.1 Planificadores (ii)

- Planificación **apropiativa**
  - El proceso puede ser interrumpido por el S.O. para entregar la CPU a otro proceso
- Planificación **no apropiativa**
  - El proceso se ejecuta hasta que cede el control al S.O.

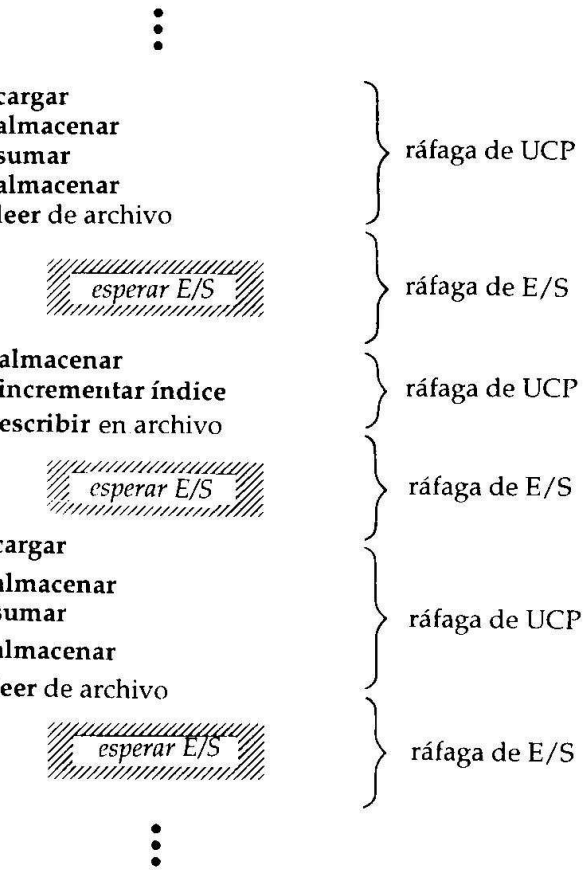
Diagrama de colas

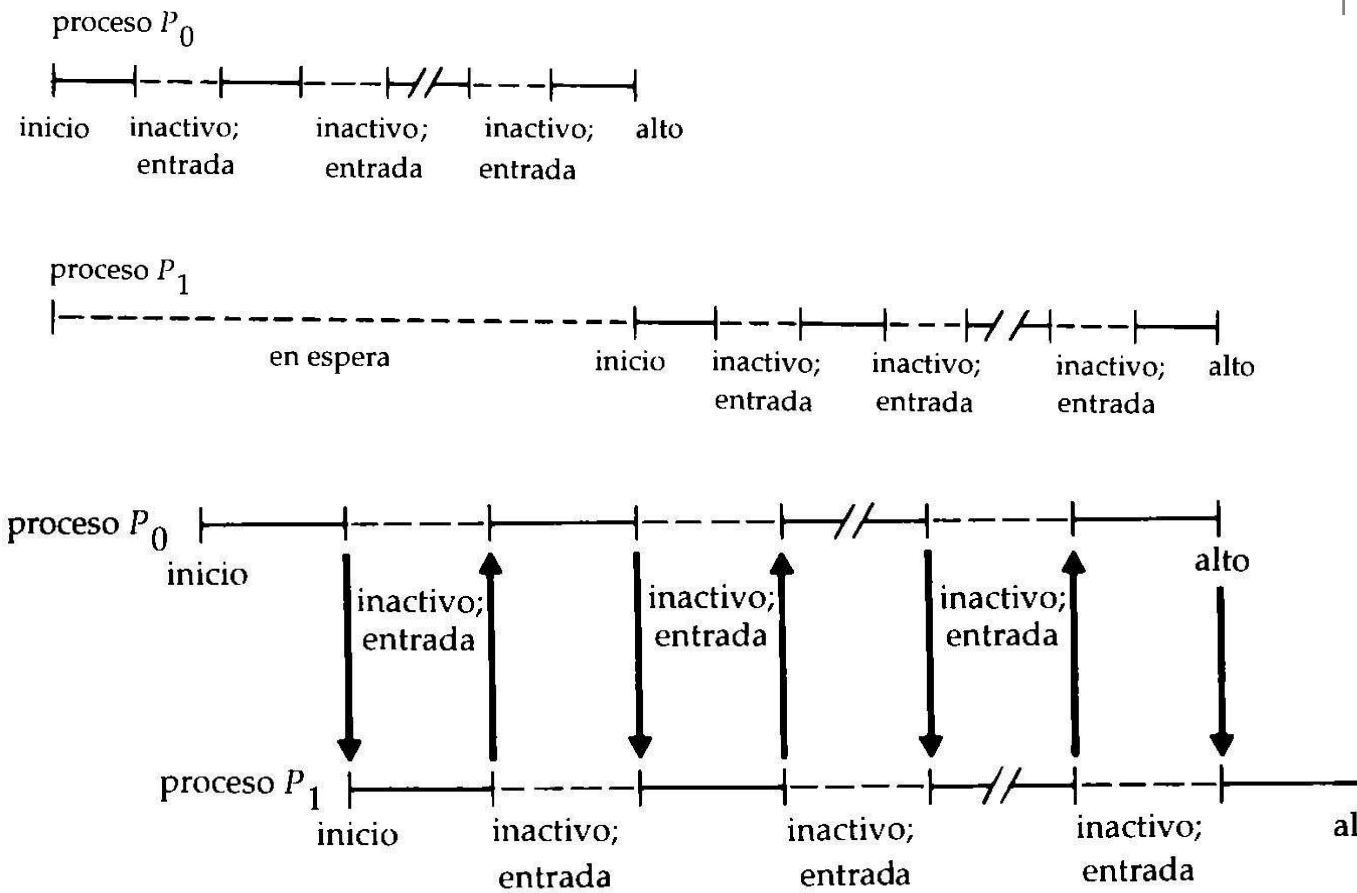


4.1 Planificadores (iii)

- Los procesos son impredecibles:
  - Aunque siempre siguen el patrón: Ráfagas de CPU seguidas de Ráfagas de E/S
  - La duración de las ráfagas no se sabe *a priori*, aunque se puede inferir de la historia de la ejecución del proceso
  - Procesos limitados por CPU
    - Hacen muy poca E/S
    - Su principal tarea es el uso de la CPU para realizar cálculos
  - Procesos limitados por E/S
    - Realizan procesos que requieren de poco cálculo y mucha entrada/salida
    - P.ej. Un shell, un comprobador de consistencia de un sistema de ficheros...

Ráfagas de CPU y de E/S



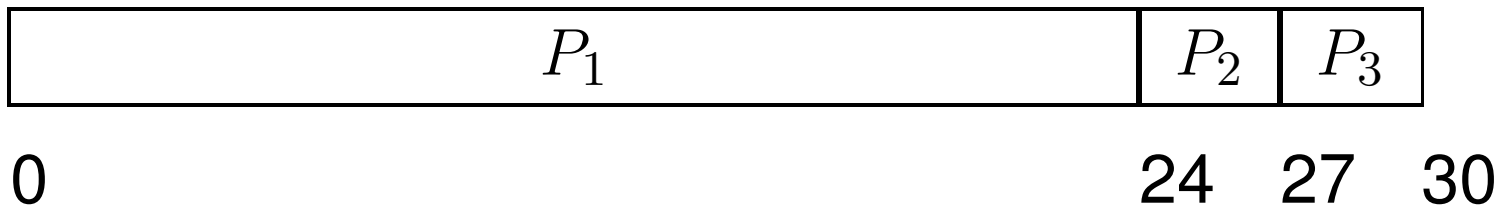


.2 Primero en llegar, primero en ser servid

FCFS (Primero en llegar, primero en ser servido)

- Se le asigna la CPU al primer proceso que la requiere $\Rightarrow$ FIFO

Proceso	Duración
P_1	24
P_2	3
P_3	3



- Tiempo medio: $\frac{24+27+30}{3} \equiv 27$
- Es **NO APROPIATIVA.** Efecto convoy.

4.3 SJF (Primero el trabajo más corto)

- Para procesos por lotes (o de duración conocida):

- También para procesos interactivos $\Rightarrow$ **predicción**

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

ESTIMACIÓN	TIEMPO REAL
T_0	T_1
$\frac{1}{2}T_0 + \frac{1}{2}T_1$	T_2
$\frac{1}{4}T_0 + \frac{1}{4}T_1 + \frac{1}{2}T_2$	T_3

4.3 SJF (ii)

- Para que SJF sea óptimo es necesario disponer de todos los procesos de forma simultánea:

	A	B	C	D	E
Tiempo de ejecución	2	4	1	1	1
Tiempo de llegada	0	0	3	3	3

Orden posible: A, B, C, D, E $\quad \frac{2+6+4+5+6}{5} = 4.6$

Orden mejor: B, C, D, E, A $\quad \frac{4+2+3+4+9}{5} = 4.4$

NO POSIBLE: C, D, E, A, B

  - ¡C, D y E llegan en el instante 3!
- El descrito es NO APROPIATIVO. Para hacerlo APROPIATIVO:
  - Primero el que tenga menor tiempo restante (SRTF)

4.4 Planificación Round Robin o circular

- A cada proceso se le asigna un **quantum** de tiempo:
  - Termina antes de consumir el *quantum*
  - Queda bloqueado (E/S)
  - Consume su tiempo
- Cada proceso espera a lo sumo $(n-1)q$ unidades de tiempo
- Importante: **Longitud del quantum**
  - Quantum pequeño: Ineficiente $\Rightarrow$
    Tiempo de administración $>$ Tiempo de ejecución
  - Quantum grande: Los procesos esperan mucho

4.5 Planificación por prioridad

- Normalmente los procesos tienen asociada una prioridad
- Se ejecuta el proceso con mayor prioridad
- Asignación de prioridades estática o dinámica
- Por ejemplo: para favorecer a los procesos limitados por E/S:
  - Prioridad $\frac{1}{F}$, donde F $\Rightarrow$ fracción de quantum que utilizó
  - $Q = 100\text{ms}$ (quantum)
  - $F = \frac{1}{2}$ (50ms) $\Rightarrow$ prio. 2; $F = \frac{1}{50}$ (2ms) $\Rightarrow$ prio. 50
- Puede ser **apropiativa o no**
- Se puede dar **bloqueo indefinido** o **inanición**:
  - Ir disminuyendo la prioridad de los que se ejecutan
  - Ir aumentando la prioridad de los que no se ejecutan

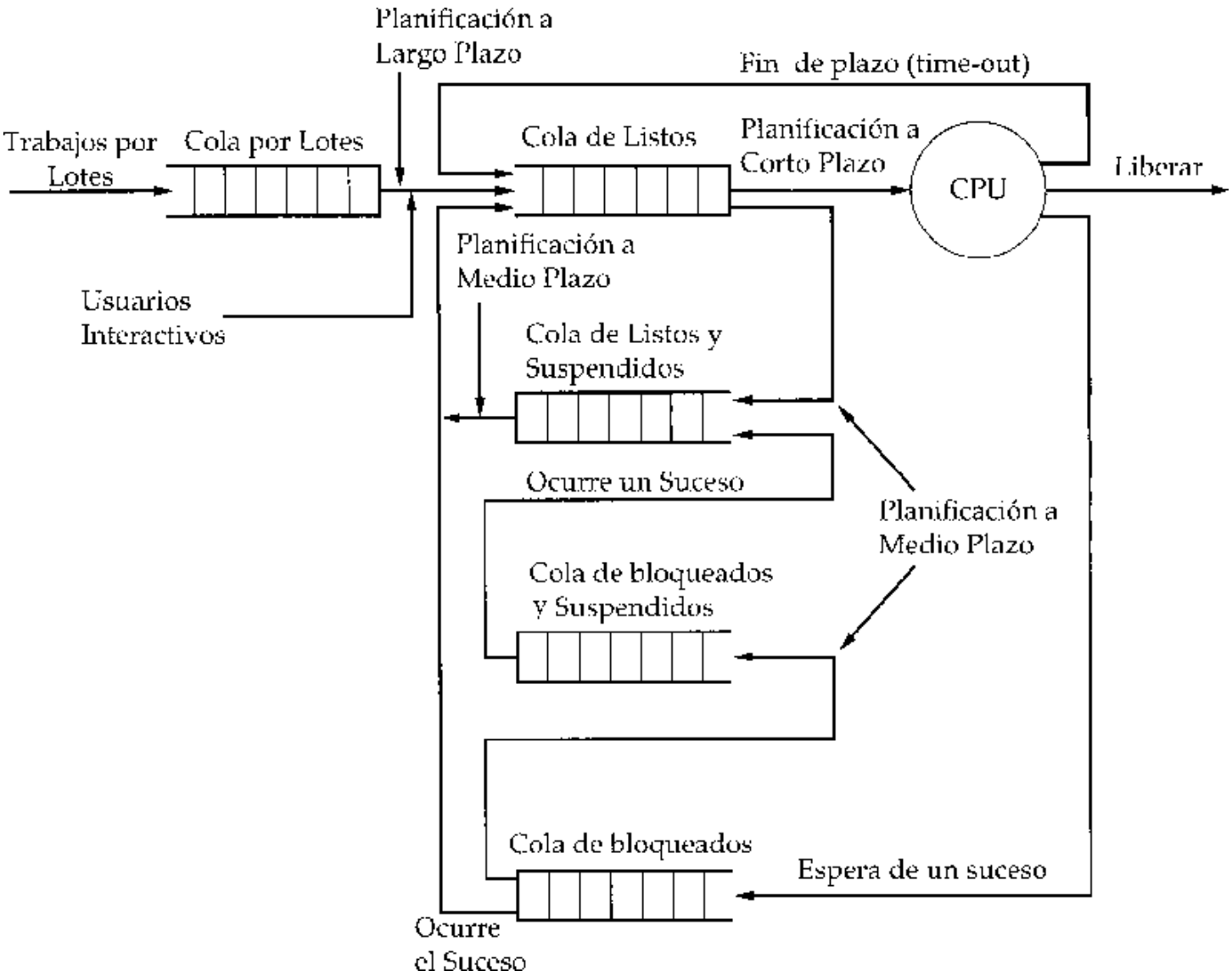
4.6 Planificación de múltiples niveles con realimentación

- Varias colas con distintos niveles de prioridad
- Cada cola $\Rightarrow$ propio algoritmo de planificación
- También planificación entre las distintas colas
- 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.

.7 Planificación a corto, medio y largo plaz

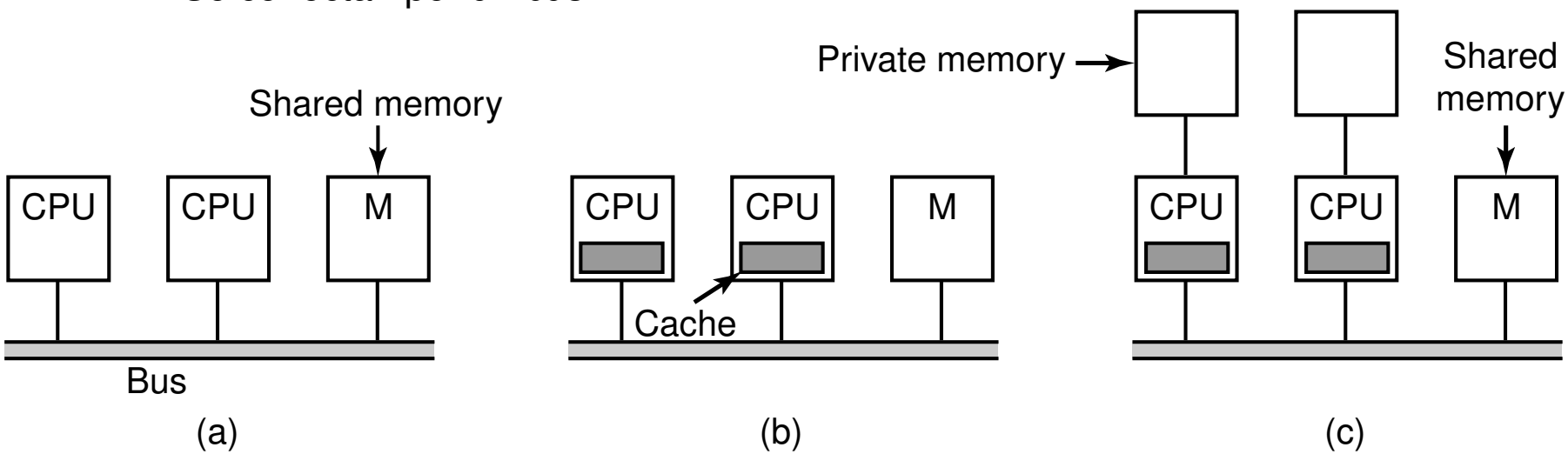
- Planificación de n niveles
  - No todos los procesos caben en memoria:
    - **Planificador a corto plazo (PCP)**
      - Selecciona de entre todos los procesos listos en memoria cuál pasar a la CPU
    - **Planificador a medio plazo (PMP)**
      - Selecciona qué procesos pasarán de memoria a disco (se suspenderán) y viceversa
    - **Planificador a largo plazo (PLP)**
      - Selecciona qué trabajos por lotes dejará pasar a la cola de listos en memoria

.7 Planificación a corto, medio y largo plaz



5 Multiprocesamiento (Stallings, 4.2), (Tanenbaum, 8.1)

- Normalmente, se consideran sistemas con una sola CPU
- Sin embargo, el desarrollo tecnológico permite cada vez más construir sistemas con varias CPUs
- Los más comunes son los multiprocesadores simétricos (SMP)
  - La memoria es compartida
  - Pueden tener una caché
  - Se conectan por un bus



5 Multiprocesamiento (ii)

- SMP
  - El núcleo puede ejecutarse en cualquiera de los procesadores
  - Además, se puede diseñar de manera que partes del núcleo se ejecuten en cada procesador
  - Un núcleo que soporte SMP es más complejo:
    - Dos procesadores no intenten ejecutar el mismo proceso
    - Intentar aprovechar el TLB y la caché
    - Mecanismos de sincronización y código del núcleo *reentrante*
    - Tolerancia a fallos
  - Planificación bidimensional: ¿qué proceso y en qué CPU?

6 Procesamiento en tiempo real

- La exactitud de un sistema no sólo depende del resultado sino también del instante en el que se produzca
- En un S.O de tiempo real, algunas tareas tienen restricciones estrictas de tiempo de ejecución, otras pueden ser periódicas
- Características necesarias de los S.O. de tiempo real:
  - Determinismo – Las operaciones se realizan en instantes fijos y predeterminados. Los servicios a interrupciones tardan un tiempo predecible
  - Control del usuario – El usuario puede modificar el algoritmo de planificación, se permiten prioridades, etc.
  - Fiabilidad – Normalmente controlan sistemas sensibles
  - Tolerancia a fallos

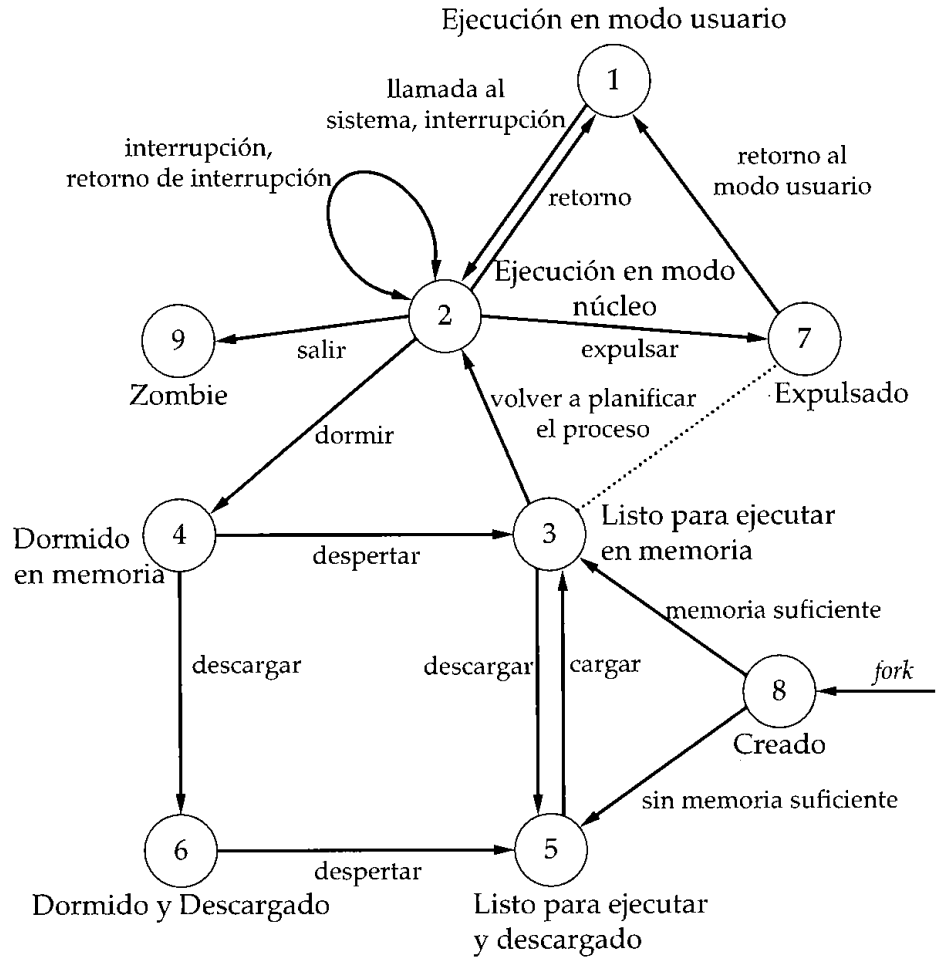
6 Procesamiento en tiempo real (ii)

- Características de los S.O. de tiempo real actuales:
  - Cambios rápidos entre procesos e hilos
  - Pequeño tamaño (funcionalidad mínima necesaria)
  - Respuesta rápida a interrupciones
  - Multitarea con herramientas de sincronización (semáforos, señales, etc.)
  - Archivos secuenciales especiales para adquisición de datos
  - Planificación basada en prioridades
  - Tiempo mínimo con interrupciones deshabilitadas
  - Primitivas para control estricto del tiempo, alarmas, temporizadores, etc.

Índice

7 Procesos en Unix

- Cada proceso con dos partes:
  - Parte de núcleo: pila y contador de programa propios
  - Parte de usuario
- 9 estados posibles:
  1. Ejecución en modo usuario
  2. Ejecución en modo núcleo
  3. Listo en memoria
  4. Bloqueado en memoria
  5. Listo suspendido
  6. Bloqueado suspendido
  7. Apropiado (*preempted*)
  8. Creado
  9. Zombie



7.1 Estructuras de datos

- El núcleo guarda dos estructuras:
  - Tabla de procesos
    - Reside en memoria
    - Parámetros de planificación (prioridad, etc.)
    - Imagen de la memoria (o posición en disco)
    - Máscara de señales (cuáles se ignoran, etc.)
    - Estado del proceso, tamaño, *pid*, *uid*, etc.
  - La estructura de usuario
    - Se intercambia junto al proceso. **Datos válidos sólo en ejecución**
    - Registros de la máquina
    - Manejadores de señales
    - Descriptores de ficheros
    - Contabilidad

7.2 Creación de procesos (`fork` y `exec`)

- `fork(2)`
  - El proceso padre llama a `fork()`
  - El núcleo asigna un *pid* al nuevo hijo
  - Tabla de procesos del padre $\Rightarrow$ hijo
  - Se copian segmentos de datos y pila.
    El código se comparte
  - También se copia la estructura de usuario
  - El proceso hijo $\Rightarrow$ "Listo"
  - Valor devuelto: *PID* al padre, 0 al hijo
- `exec(3)`
  - **No** se crea un proceso nuevo $\Rightarrow$ mismo *pid*
  - Se reemplazan código, datos, pila, etc.

7.3 Planificación de procesos en UNIX

- Algoritmo de dos niveles con PCP y PMP
- El PCP utiliza múltiples colas con realimentación
- Ejecuta el primer proceso de la cola de mayor prioridad
- Normalmente Q = 100ms, Reloj = 50 ó 60 Hz $\Rightarrow$ Cada *quantum* = 5 o 6 marcas $\Rightarrow$ se suman a la prioridad del proceso $\Rightarrow$ menos prioridad
- Además, se hace el siguiente cálculo cada segundo:
  Uso de CPU = Uso de CPU / 2. Nueva prioridad = Base + Uso de CPU
- Base $\equiv$ Número NICE. Usuario $\Rightarrow \geq 0$
- Un proceso en modo núcleo no puede ser expulsado

7.3 Planificación de procesos en UNIX (ii)

- Linux
  - Cada proceso tiene una prioridad (normalmente 20) y se utiliza el número *nice* para hacer: 20 - *nice*
  - Cada proceso tiene también un *cuanto*, contado en *jiffies* (10ms)
  - Para cada proceso se calcula un valor de bondad:
    - Máxima para los procesos en tiempo real
    - Más prioridad a los procesos que no consumen su cuanto completo
    - Mínima para los que han consumido su cuanto
  - Se elige el proceso de mayor bondad
  - Cuando todos los procesos listos se quedan sin cuanto: cuanto = (cuanto/2) + prioridad

7.3 Linux 2.4.22 kernel/sched.c

Goodness (no-realtime)

```
inline int goodness( ... )
{
                weight = p->counter;
                if (!weight)
                        goto out;
#ifdef CONFIG_SMP
                /* Give a largish advantage to the same processor...   */
                /* (this is equivalent to penalizing other processors) */
                if (p->processor == this_cpu)
                        weight += PROC_CHANGE_PENALTY;
#endif

                weight += 20 - p->nice;
                goto out;
out:
        return weight;
}
```

7.3 Linux 2.4.22 kernel/sched.c (ii)

```
void schedule(void)
{
repeat_schedule:
        /*
         * Default process to select..
         */
        next = idle_task(this_cpu);
        c = -1000;
        list_for_each(tmp, &runqueue_head) {
                p = list_entry(tmp, struct task_struct, run_list);
                if (can_schedule(p, this_cpu)) {
                        int weight = goodness(p, this_cpu, prev->active_mm
                        if (weight > c)
                                c = weight, next = p;
                }
        }
```

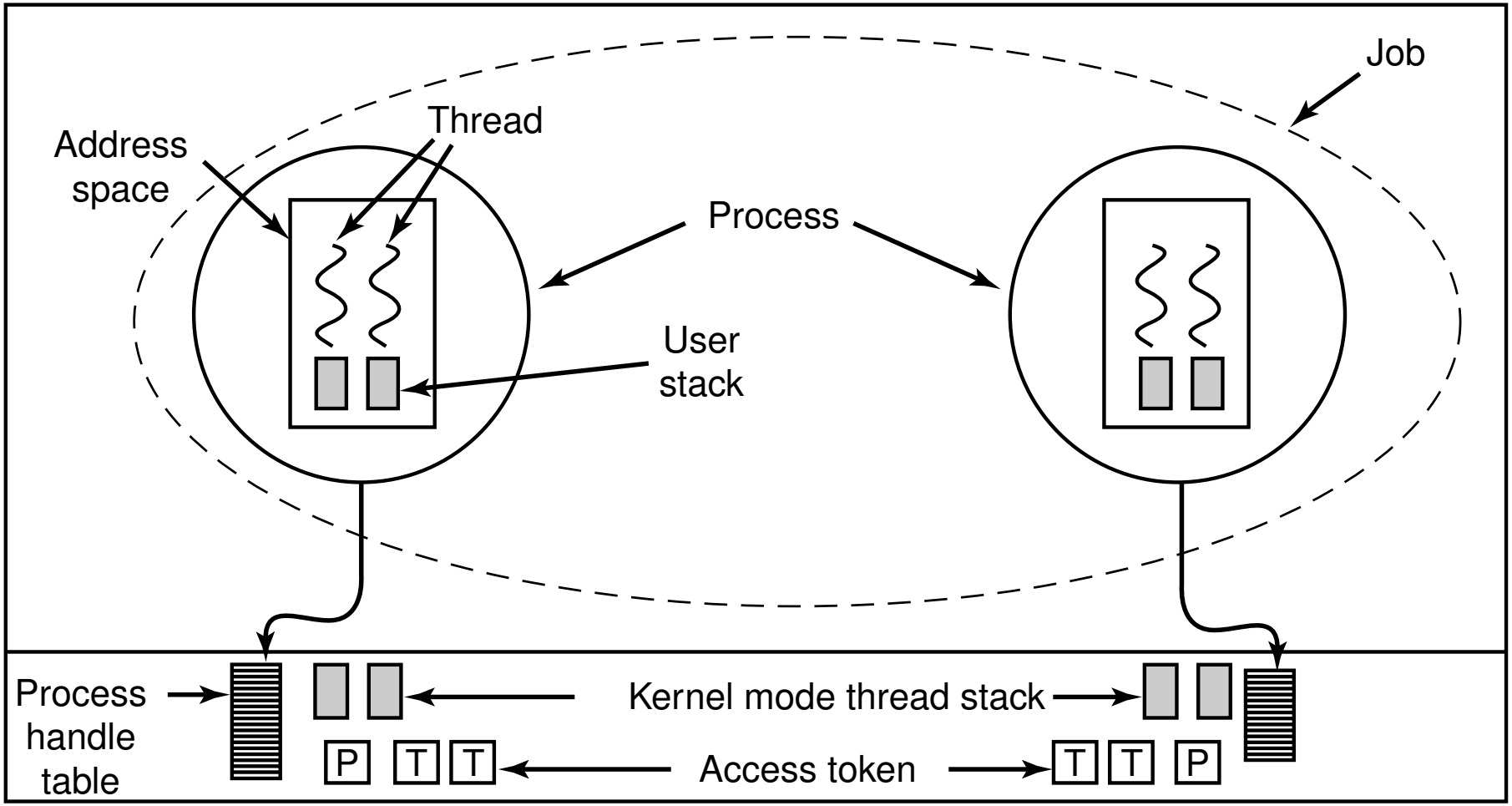
7.3 Linux 2.4.22 kernel/sched.c (iii)

```
/* Do we need to re-calculate counters? */
if (unlikely(!c)) {
        struct task_struct *p;

        for_each_task(p)
                p->counter = (p->counter >> 1) +
                                NICE_TO_TICKS(p->nice);
        goto repeat_schedule;
}

// next es el proceso a seleccionar...
```

Procesos en Windows 2000 (Tanenbaum, 11.4), (Stallings, 4.4, 10

- Windows 2000 es capaz de soportar varios modelos de procesos (por ejemplo, los distintos subsistemas de entorno)
- Los procesos en windows 2000:
  - Se implementan como objetos
  - Pueden tener varios hilos, pero al menos uno (planificados por el núcleo)
  - Pueden tener hilos a nivel de usuario (fibras)
  - Existe también el concepto de Trabajo: conjunto de procesos relacionados que comparten recursos
  - Los objetos proceso e hilo tienen capacidades de sincronización
  - El núcleo no conserva relaciones entre los procesos que crea

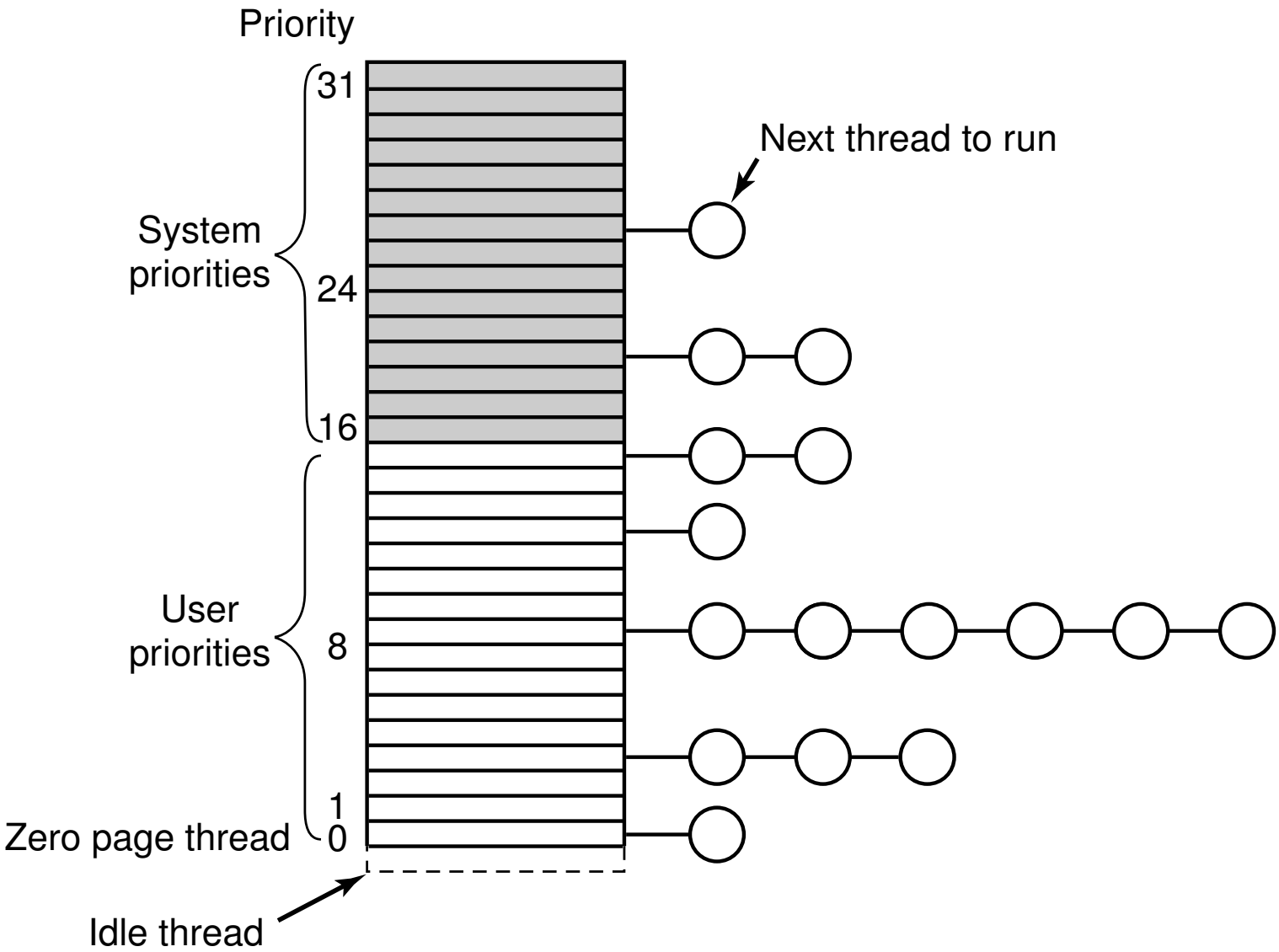
8 Procesos en Windows 2000 (ii)



8 Procesos en Win. 2000 – Planificación

- Windows 2000 es apropiativo
- El planificador se lanza por los siguientes eventos:
  - Un proceso se bloquea (semáforo, E/S)
  - Un proceso envía una señal a un objeto (equivale a una llamada al sistema)
  - Se agota el *quantum*
  - Termina una operación de E/S
  - Termina algún temporizador
- El algoritmo de planificación es de múltiples colas con prioridad y realimentación, con 32 niveles,
  - 31: tiempo real
  - 0–15: procesos del usuario
- Al terminar la E/S de un proceso, se aumenta su prioridad (+1 $\rightarrow$ disco, +2 $\rightarrow$ serie, +6 $\rightarrow$ teclado)

3 Procesos en Win. 2000 – Planificación (ii)



Procesos en OSO

- Acceso a memoria de vídeo
- Actividad desde la sai del timer
- Implementación de procesos

Procesos en OSO – Acceso a memoria de vídeo


```
int main (void)
{
        long i;
        unsigned char j;
        char far * pantalla = (char far *) 0xB8000000L;



        for (j = 0; j < 255; j++) {
                pantalla[200] = j;
                for (i = 0; i < 2000000; i++)
                        ;
        }
```

Cambia carácter

Espera un rato

Programa que hace un barrido ascii en la posición 200 de la memoria de vídeo

Procesos en OSO – Actividad desde la sai del Timer (i)


```
int main (void)
{
        long i;
        unsigned char j;
        char far * pantalla = (char far *) 0xB8000000L;

        asm cli;
        apuntaVectoraSAI (sai_timer, 0x08);
        asm sti;

        for (j = 0; j < 255; j++) {
                pantalla[200] = j;
                for (i = 0; i < 2000000; i++)
                        ;
        }
```

18.2 hz

```
void sai_timer(void)
{
        asm {
                push ax
                push bx
                push cx
                push dx
                push es
                push ds
        }

        ++(*(char far*)(0xb8000100L));

        asm  {
                pop ds
                pop es
                pop dx
                pop cx
                pop bx
                pop ax

                pop di   /*  El compilador apila  */
                pop si   /*  automáticamente      */
                pop bp   /*  estos registros      */
                iret
        }
}
```

18.2 veces por segundo se incrementa el código ascii de la posición 0x100 de la memoria de vídeo

Procesos en OSO – Actividad desde la sai del Timer (ii)


```
int main (void)
{
        long i;
        unsigned char j;
        char far * pantalla = (char far *) 0xB8000000L;

        asm cli;
        apuntaVectoraSAI (sai_timer, 0x08);
        asm sti;

        for (j = 0; j < 255; j++) {
                pantalla[200] = j;
                for (i = 0; i < 2000000; i++)
                        ;
        }
```

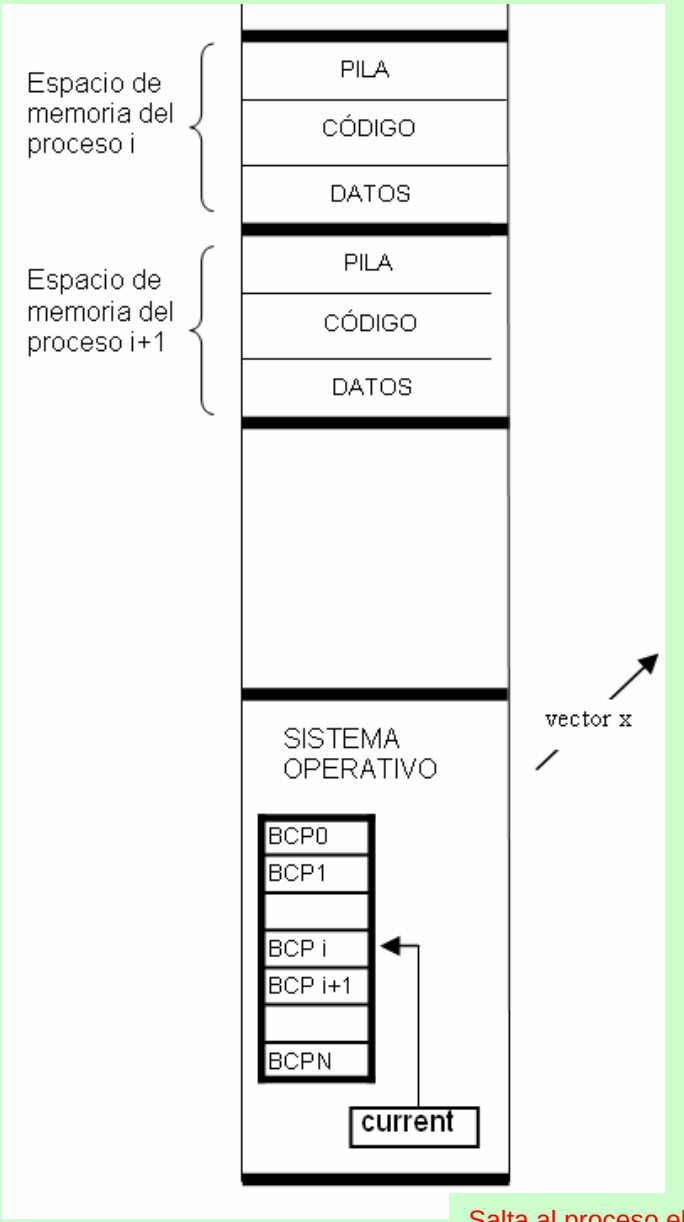
18.2 hz

```
void sai_timer(void)
{
        asm {
                push ax
                push bx
                push cx
                push dx
                push es
                push ds
        }

        ++(*(char far *)(0xb800100L));

        asm  {
                pop ds
                pop es
                pop dx
                pop cx
                pop bx
                pop ax

                pop di   /*  El compilador apila  */
                pop si   /*  automáticamente      */
                pop bp   /*  estos registros      */
                iret
        }
}
```

18.2 veces por segundo toma el control la sai del timer mediante el mecanismo de interrupción.

NORMALMENTE NO PINTA

Procesos en OSO – Implementación de procesos (i)

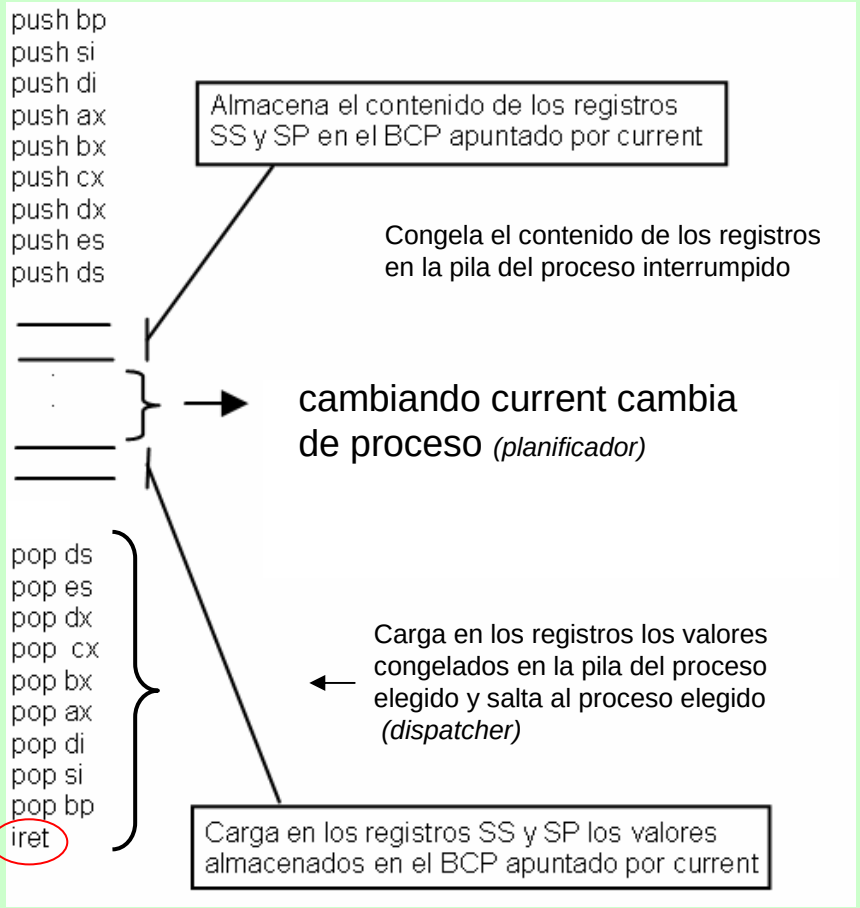


Entrada al kernel:

- *llamada al sistema*
  instrucción int 22h (vector 22)
- *activación eléctrica de cable*
  cable de timer (vector 8)
  cable de teclado (vector 9)

apilan flags y dirección de retorno

Stack Pointer: Registros SS y SP



Salta al proceso elegido (desapila flags y dirección de retorno)

Procesos en OSO – Implementación de procesos (ii)

```
void uno()
{
        unsigned far char *pantalla = (unsigned char far*) 0xb8000000L;
        long int i;
        unsigned char j;

        j=1;
        do {
                for (i = 0; i < 2000000; i++)
                {
                        /* Vacío */
                }
                pantalla[200] = j;
                ++j;
        }  while (j != 255);

        exitHilo ();
}
```

```
void dos()
{
        unsigned far char *pantalla = (unsigned char far*) 0xb8000000L;
        long int i;
        unsigned char j;

        j=1;
        do {

                for (i = 0; i < 2000000; i++)
                {
                        /* Vacío */
                }
                pantalla[300] = j;
                ++j;
        }  while (j != 255);

        exitHilo ();
}
```

OSO consigue construir la abstracción de proceso secuencial :

- Procesos (procesadores virtuales con las interrupciones ocultas)
- Implementación de la abstracción

```
struct BCP_
{
        short en_uso;  /* ¿Está siendo usado para describir un hilo? */
        short regSS;
        short regSP;
} BCP[HILOS];
```

```
int current;
```

```
void sai_timer(void)
{
        asm {
                push ax
                push bx
                push cx
                push dx
                push es
                push ds

                mov ax,cs
                mov ds,ax
                mov es,ax
        }

        BCP[current].regSS = _SS;
        BCP[current].regSP = _SP;

        current = roundrobin (current);

        _SS = BCP[current].regSS;
        _SP = BCP[current].regSP;

        asm  {
                pop ds
                pop es
                pop dx
                pop cx
                pop bx
                pop ax

                pop di
                pop si
                pop bp
                iret
        }
}
```

```
int roundrobin (int proc)
{
        do
        {
                ++proc;
                proc %= HILOS;
        } while (! BCP[proc].en_uso);

        return proc;
}
```

```
C:\PRACSO\CODIGO>
```