Sistemas Operativos II

Procesos concurrentes y *Threads*

Universidad Arturo Jauretche
Ingeniería Informática

Docentes:

Ing. Jorge Osio

Ing. Eduardo Kunysz

Procesos (repaso)

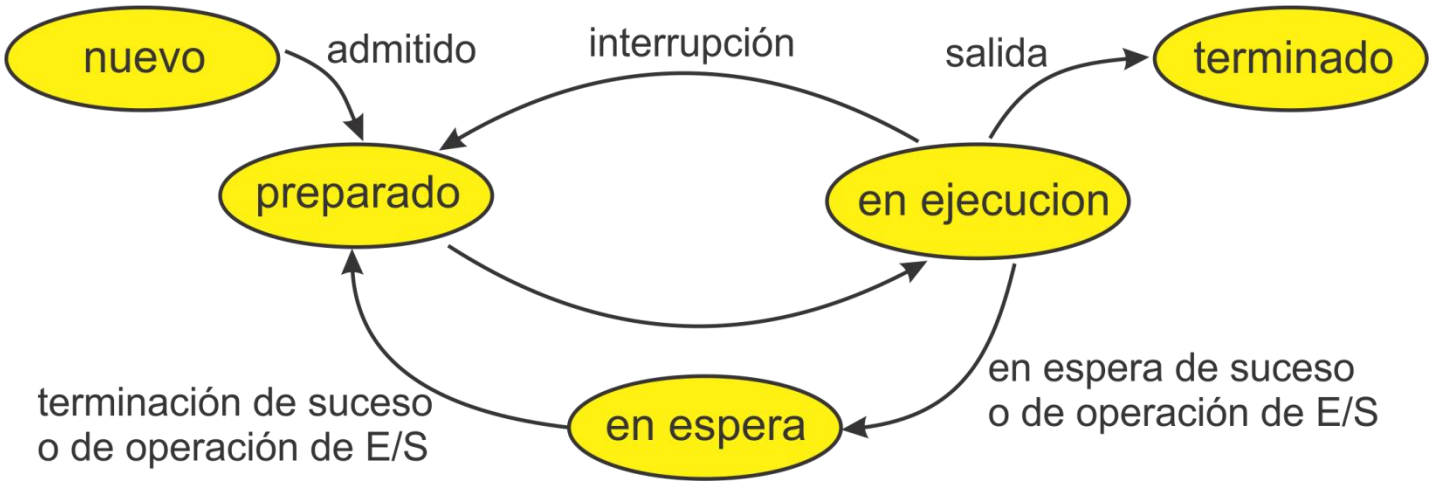
- Abstracción de un programa en **ejecución**

(un programa por si solo **no es** un proceso)

  - Un **programa** es una entidad pasiva.
  - Un **proceso** es una entidad activa.

- Capacidad de operar (pseudo) concurrentemente, incluso cuando hay sólo un CPU disponible.

Estados de un proceso



- **Nuevo:** el proceso está siendo creado
- **En ejecución**: se están ejecutando las instrucciones.
- **En espera:** el proceso está esperando a que se produzca un suceso (como la terminación de una operación de E/S o la recepción de una señal)
- **Preparado:** el proceso está a la espera de que le asignen el procesador.
- **Terminado:** ha finalizado la ejecución del proceso.

Bloque de control de procesos (PCB)

- Cada proceso se representa mediante un "Procces control Block" (PCB). Elementos:

  - **Estado del proceso:** nuevo, preparado, etc…
  - **Contador de programa:** apunta a la siguiente instrucción del proceso.
  - **Registro de la CPU:** acumulador, índice, punteros, estado.
  - **Información de planificación de la CPU:** prioridad del proceso.
  - **Información de gestión de memoria:** valor de registros básicos, tablas de paginación o segmentos, etc.
  - **Información contable:** cantidad de CPU, tiempo asignado …
  - **Información del estado E/S:** lista de dispositivos asignados al proceso

Concurrencia: Algunos conceptos

Para entender la programación concurrente hay que tener claros tres conceptos:

- Concurrencia
- Paralelismo
- Tiempo Real

Concurrencia

- Dos procesos son concurrentes si **son independientes** entre si.
- Dos procesos son independientes si el resultado de la ejecución de ambos es siempre el mismo independientemente del orden en que se ejecutan.
- Ejemplo:

P1: `printf("pelo ");`

P2: `printf("corto ");`

No son concurrentes. Resultados posibles:

1. `"pelo corto"` (se ejecuta primero P1 y luego P2)

2. `"corto pelo"` (se ejecuta primero P2 y luego P1)

Paralelismo

- Ejecución simultánea de varios procesos.
- Se requieren hardware adecuado (varios procesadores)
- Paralelismo indica ejecución simultánea por lo tanto:

Paralelismo ⇒ Concurrencia
Concurrencia **no implica** Paralelismo

Tiempo real

- Relacionado con las restricciones temporales a las que se somete cierto proceso.
- Muchas veces por diseño se requiere concurrencia.
- Pero no necesariamente concurrencia equivale a tiempo real

Gestión de procesos

- **Multiprogramación:** Consiste en la gestión de varios procesos dentro de un sistema mono-procesador.
- **Multiprocesamiento:** consiste en la gestión de varios procesos dentro de un sistema multiprocesador.
- **Procesamiento distribuido:** consiste en la gestión de varios procesos, ejecutándose en sistemas de computadores múltiples y distribuidos.

Diseño concurrente

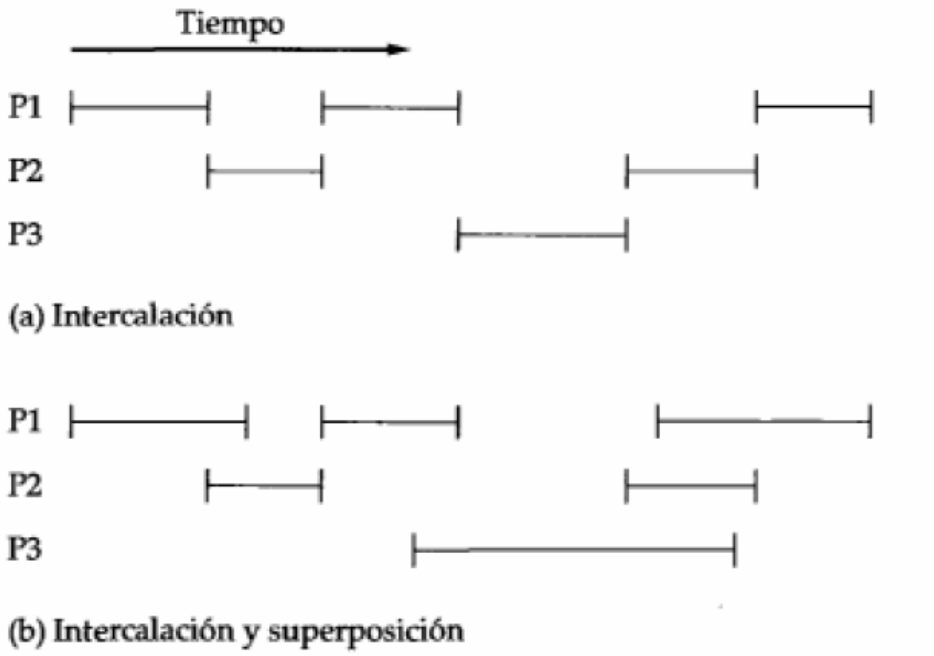
- La concurrencia es el punto clave de los puntos anteriores.
- Cuestiones de diseño concurrentes:
  - Comunicación entre procesos
  - Compartición y competencia por los recursos
  - Sincronización de la ejecución de varios procesos
  - Asignación del tiempo del procesador a los procesos.

Contextos de concurrencia

- **Varias aplicaciones:** la multiprogramación se creó para permitir que el tiempo del procesador de la máquina fuese compartido dinámicamente entre varios procesos.
- **Aplicaciones estructuradas:** programación modular y estructurada (ej tiempo real)
- **Estructura del sistema operativo:** los sistemas operativos se implementan como conjunto de procesos.

Ejecución de procesos concurrentes

- En un único procesador, los procesos se intercalan (a).
- Varios procesadores se permite superposición (b).

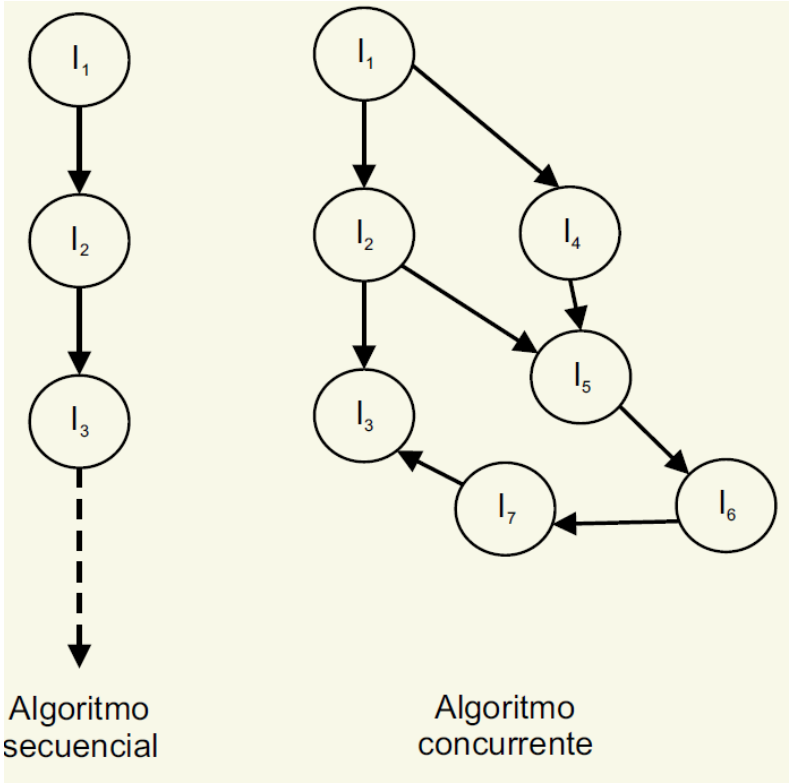


Ambas técnicas pueden contemplarse como procesos concurrentes y las dos plantean los mismos problemas

Grafos de precedencia

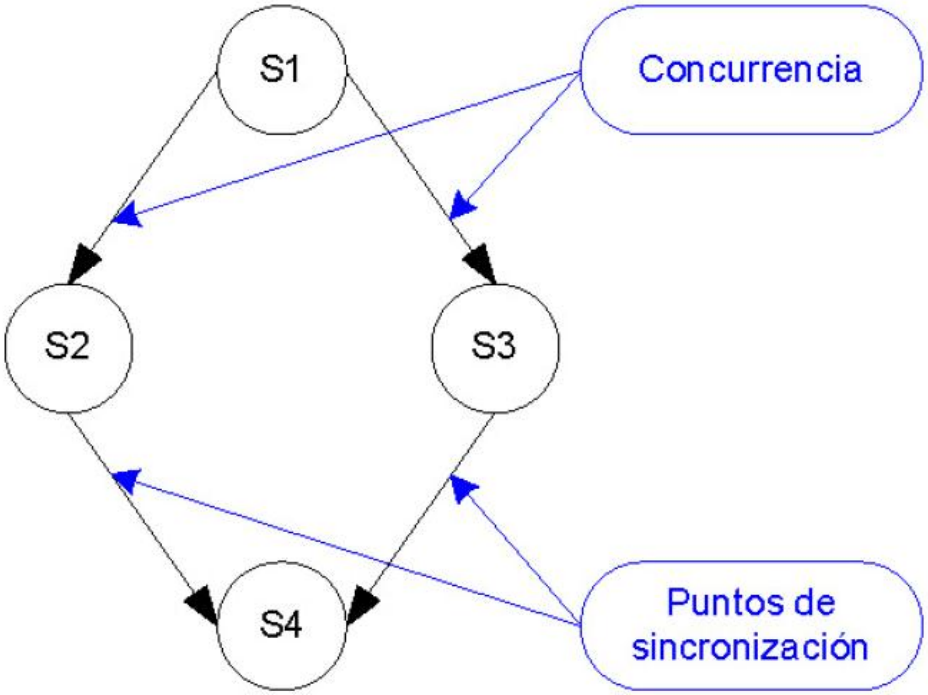
- Representación gráfica de un algoritmo en el que se establece el orden de ejecución de un conjunto de instrucciones y/o datos.

- Son grafos cíclicos, no presentan bucles, donde cada Ii representa un conjunto de sentencias que se unen mediante arcos que indican precedencia de orden de ejecución.
- Las flechas indican secuencialidad



Procesos concurrentes

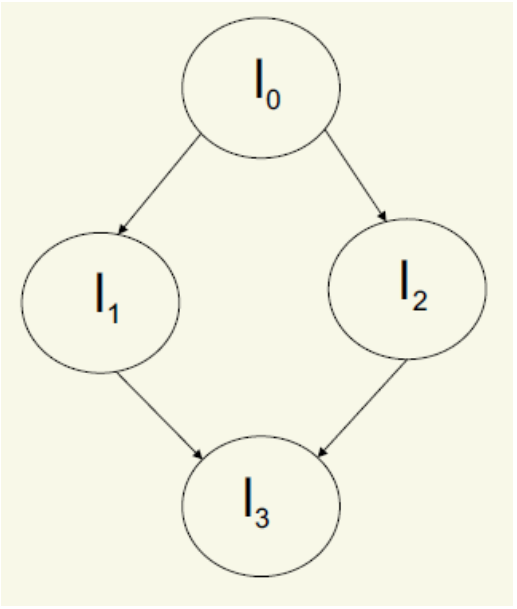
- Los procesos concurrentes de un algoritmo pueden ser independientes o compartir datos
- Nuestro objetivo: estudiar ambos tipos de relaciones entre procesos concurrentes, mediante soluciones independientes del procesador y su algoritmo de planificación



Sentencias FORK / JOIN

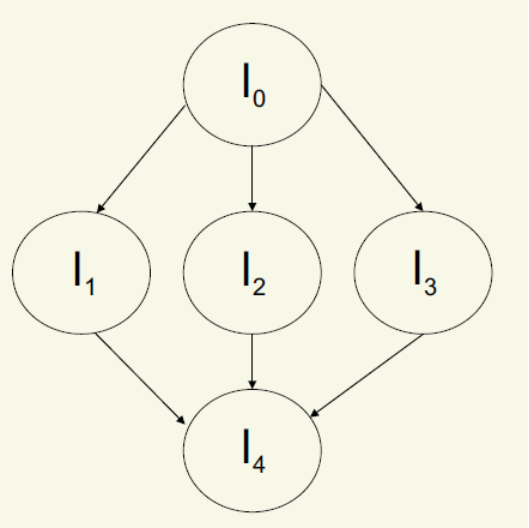
- Fork <etiqueta> => Divide el flujo en 2 procesos:
  - El que viene después de Fork
  - El que viene después de la etiqueta
- Join <contador>: Espera hasta que se acaben los "contador" procesos que debe unir antes de seguir la ejecución.
- **Ejemplo 1:**

```
contador = 2;
      I0;
      Fork L1
      I1;
      GOTO L2;
L1:  I2;
L2:  JOIN contador;
      I3;
```



Contrucciones FORK y JOIN (II)

- Ejemplo 2:



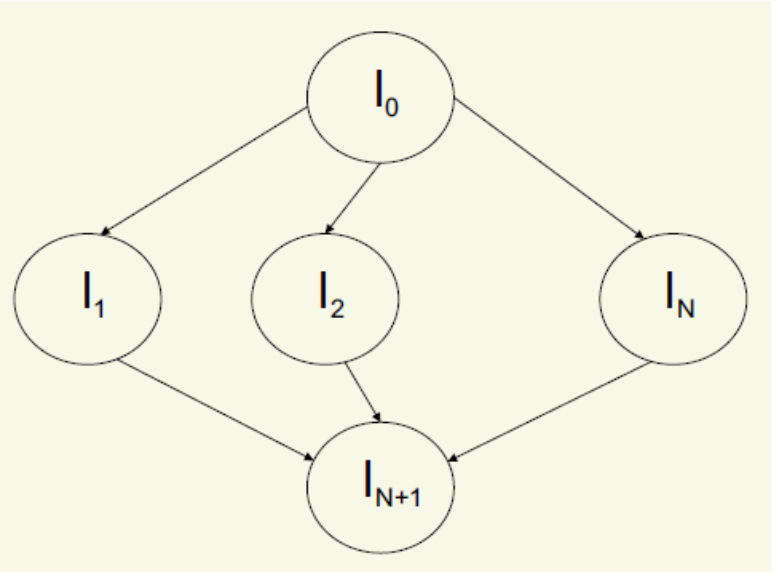

```
        contador = 3;
        I0;
        FORK L1;
        I1;
        GOTO L2;
L1:     FORK L3;
        I2;
        GOTO L2;
L3:     I3;
L2:     JOIN contador;
        I4;
```

- Problemas:
  - El uso de GOTO perjudica la legibilidad del programa
  - No empleado en lenguajes de alto nivel.
  - Difícil depuración => etiquetas
- Alternativa: sentencia COBEGUIN y COEND

Construcciones COBEGIN y COEND (Dijkstra)

- También denominada PARBEGIN y PAREND
- COEND cierra tantas ramas como haya abierto COBEGIN.
- Ejemplo:

```
I0;
COBEGIN
        I1, I2,…., In;
 COEND;
 In+1;
```



- Mas difícil de expresar con grafos de precedencia.
- Se implementan mediante grafos jerarquizados (muchos procesos)

Relaciones jerárquicas de procesos

- En UNIX todo proceso tiene proceso padre que lo ha creado y este crea hijos

- Eventos que crean procesos:
  - El arranque del sistema
  - La ejecución, desde un proceso, de una llamada al sistema para creación de procesos.
  - Una pteición de usuario para crear un proceso.
  - El inicio de un trabajo por lotes

Relaciones jerárquicas de procesos

- Los procesos creados pueden ser:
  - En primer plano (interacción con el humano)
  - Segundo plano (**demonios** o "daemons").
- Para ver por ejemplo una lista de los procesos en ejecución en UNIX se puede utilizar el comando **PS:**

```
ahornero@Eee:~$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.1   2528  1488 ?        Ss   11:01   0:01 /sbin/init
root         2  0.0  0.0      0     0 ?        S<   11:01   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S<   11:01   0:00 [migration/0]
root         4  0.0  0.0      0     0 ?        S<   11:01   0:01 [ksoftirqd/0]
root         5  0.0  0.0      0     0 ?        S<   11:01   0:00 [watchdog/0]
```

Creación de procesos en C

```
int n, pid, status;
n := 2;
if ((pid = fork()) == -1) {
        printf("error en la creación del proceso hijo\n");
        exit(1);
}
if (pid == 0) {
        /* codigo hijo */
        exit(0);
}
else {
        /* proceso padre */
}
wait(&status); /*variable que indica la causa de la
finalización */
exit(0);
```

Creación de procesos en C

- Cuando se ejecuta fork(), se crea un proceso hijo con igual código que el padre, con los mismo datos.

- "fork()" puede devolver:
  - -1 -> error, no se puede crear el proceso hijo
  - 0 -> se lo devuelve al proceso hijo
  - >0 -> se le devuelve al padre el pid del hijo

- Cuando finaliza el hijo queda en estado zombi hasta que el padre ejecute wait. (permanece en la lista PCB del SO)

Creación de procesos en C

- En la creación de procesos hijos pueden ocurrir dos casos:

  1) El proceso hijo termina antes de que termine el proceso padre

  ```
  exit(0);
  ```

  El proceso padre finaliza su código y ejecuta:

  ```
  wait(&status);
  ```

  2) El proceso padre termina antes que el hijo

  ```
  wait(&status);
  ```

  El proceso hijo finaliza su ejecución

  ```
  exit(0);
  ```

Condición de disyunción

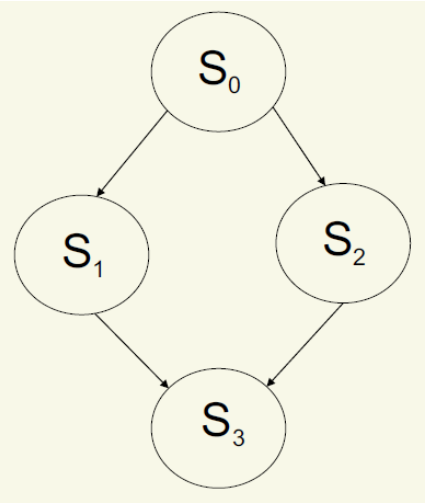
La condición de disyunción es una condición suficiente para que dos conjuntos de sentencias **S1 y S2 sean independientes**, por tanto, **los podemos ejecutar de forma concurrente**

- Si existe alguna variable u otro recurso compartido que es modificado por S1 no puede aparecer en S2.

- En el siguiente caso se presenta dicha problemática:

Procesos concurrentes que comparten datos

- Al compartir datos entre procesos se pueden producir problemas de indeterminismo (resultados diferentes según escenario de prueba)
- Ejemplo: S1 y S2 no son independientes, sino que comparten la variable x.




```
S0: x=100;
S1: x:= x + 10;
S2: if x > 100 then write(x)
       else write(x - 50);
S3: nop;
```

Escenario #1: S1 y S2 => Se escribe x = 110.
Escenario #2: S2 y S1 => Se escribe x = 50.
Escenario #3: S2 pierde el control (ej fin de quantum) antes de escribir y sigue S1 => Se escribe x = 60

Solución de Bernstein

- Cada proceso Pi está asociado a dos conjuntos:
  - R(Pi): conjunto de variables accedidas durante la ejecución de Pi.
  - W(Pi): conjunto de variables modificadas durante la ejecución de Pi.

Para dos procesos Pi y Pj, **concurrentes,** puedan ejecutarse de forma determinista tienen que satisfacerse las siguientes condiciones:
$R(Pi) \cap W(Pj) = 0$
$W(Pi) \cap R(Pj) = 0$
$W(Pi) \cap W(Pj) = 0$
Condiciones suficientes pero no necesarias => sólo se pueden compartir variables de lectura

Hilos – *Threads - Hebras*

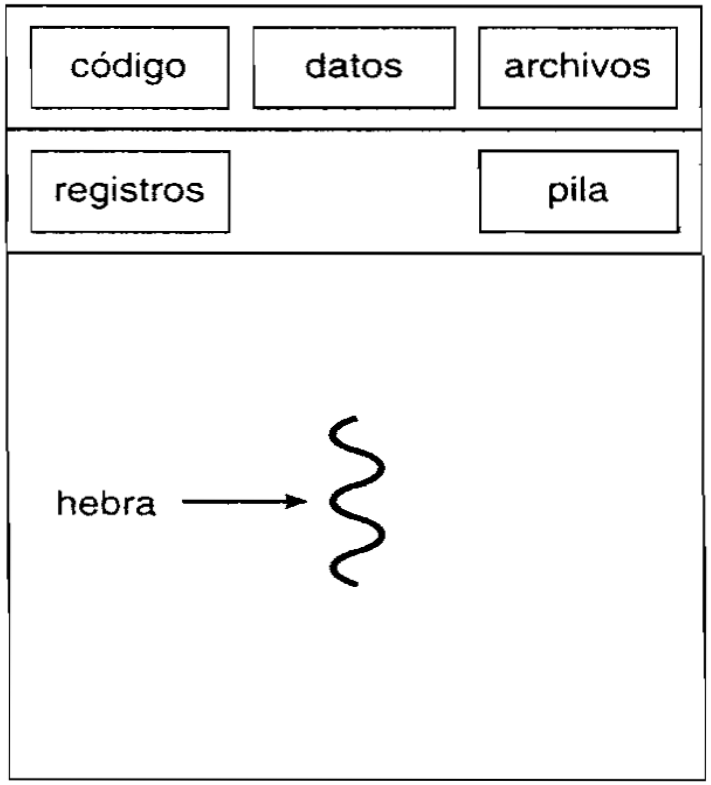
- Un **hilo o hebra** es un miniprocesos dentro de procesos que comparten el mismo espacio de direcciones.

- Unidad básica que utiliza:
  - CPU
  - ID de hilo
  - Un contador de programa
  - Un conjunto de registros y una pila

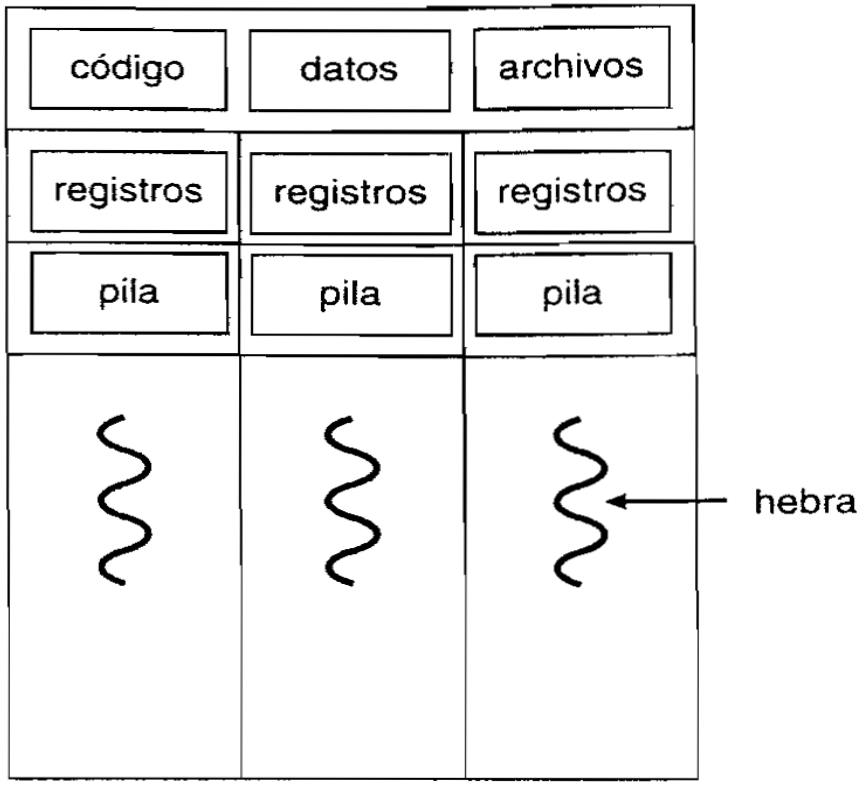
- Comparte con otras hebras:
  - El mismo proceso
  - La sección de datos
  - Otros recursos del SO: archivos abiertos, señales, etc.

Procesos multi-hilos

- Un proceso pesado tiene una hebra de control
- Un proceso con múltiples hebras puede realizar mas de una tarea a la vez.



proceso de una sola hebra



proceso multihebra

Ventajas de la programacion multihebra

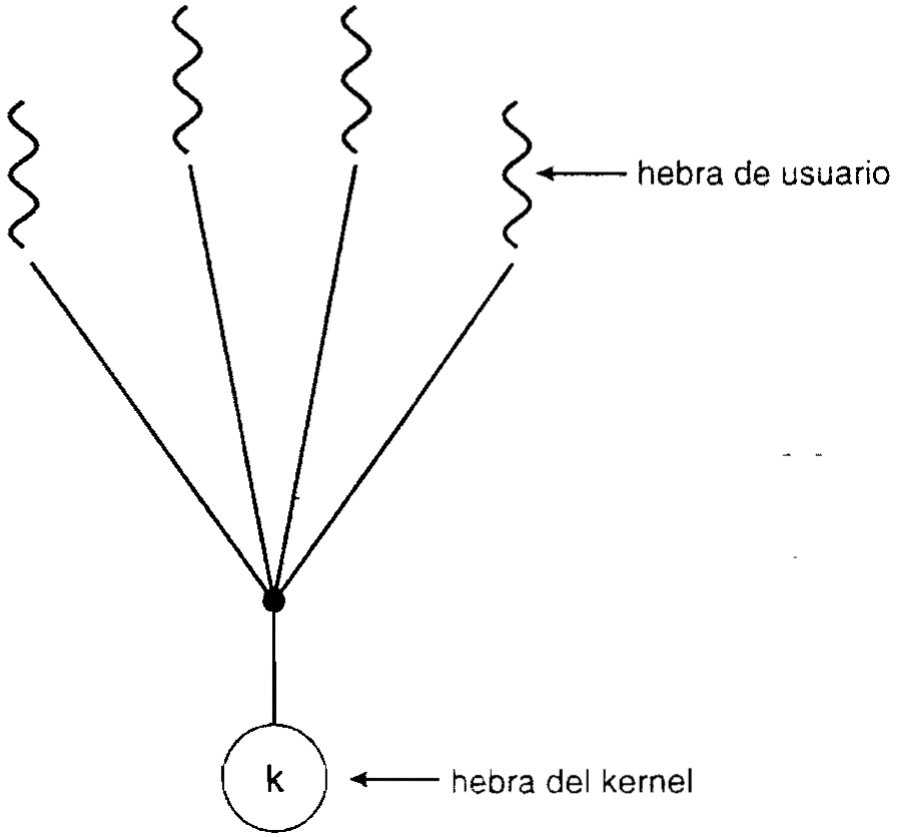
- **Capacidad de respuesta:** permite que un programa continúe ejecutándose incluso aunque parte de él esté bloqueado.
- **Compartición de recursos:** por omisión, las hebras comparten la memoria y los recursos del proceso al que pertenecen.
- **Economía:** asignar memoria y recursos para crear procesos es costosa, es más económico realizar cambios de contexto entre hebras
- **Multiprocesador:** se pueden ejecutar en paralelo en diferentes procesadores (un proceso monohebra solo se puede ejecutar en un solo uP).

Modelos multihebra

- El soporte para hebras puede proporcionarse en el nivel de **usuario** o por parte del **kernel**
- Soporte para hebras de usuario se proporcionan por encima del kernel y se gestionan sin soporte del mismo
- Las hebras de kernel son soportadas y gestionadas por el SO.
- Existe una relación entre las hebras de usuario y las de kernel:
  - Modelo muchos a uno
  - Modelo uno a uno
  - Modelo muchos a muchos

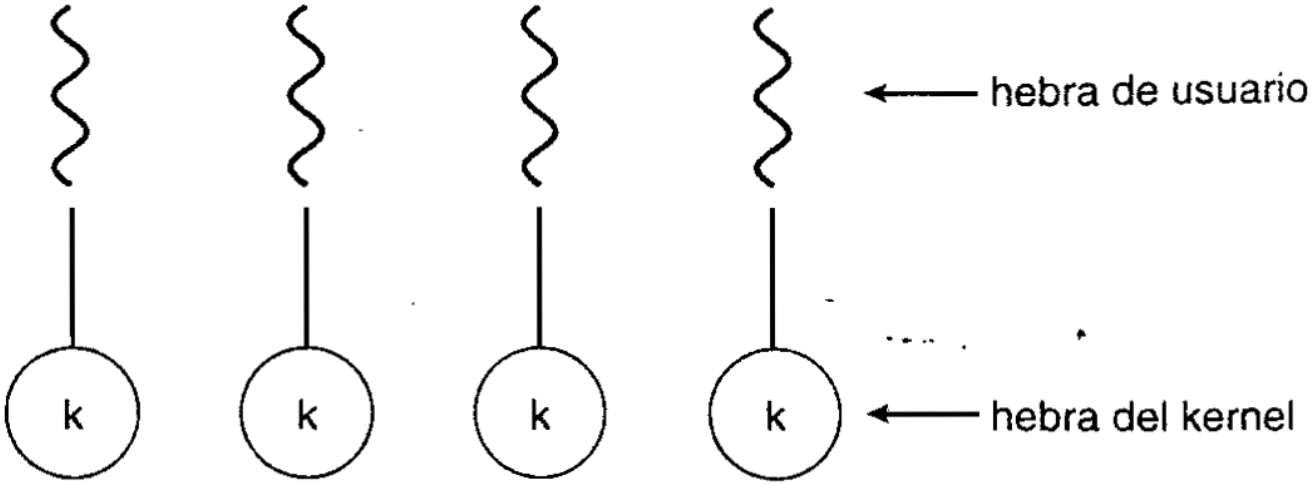
Modelo muchos a uno

- Múltiples hebras nivel usuario a una hebra del kernel.
- Gestión mediante biblioteca de hebras en espacio de usuario
- Eficiente, pero el proc. Completo se bloquea si una hebra realiza una llamada bloqueante al sistema



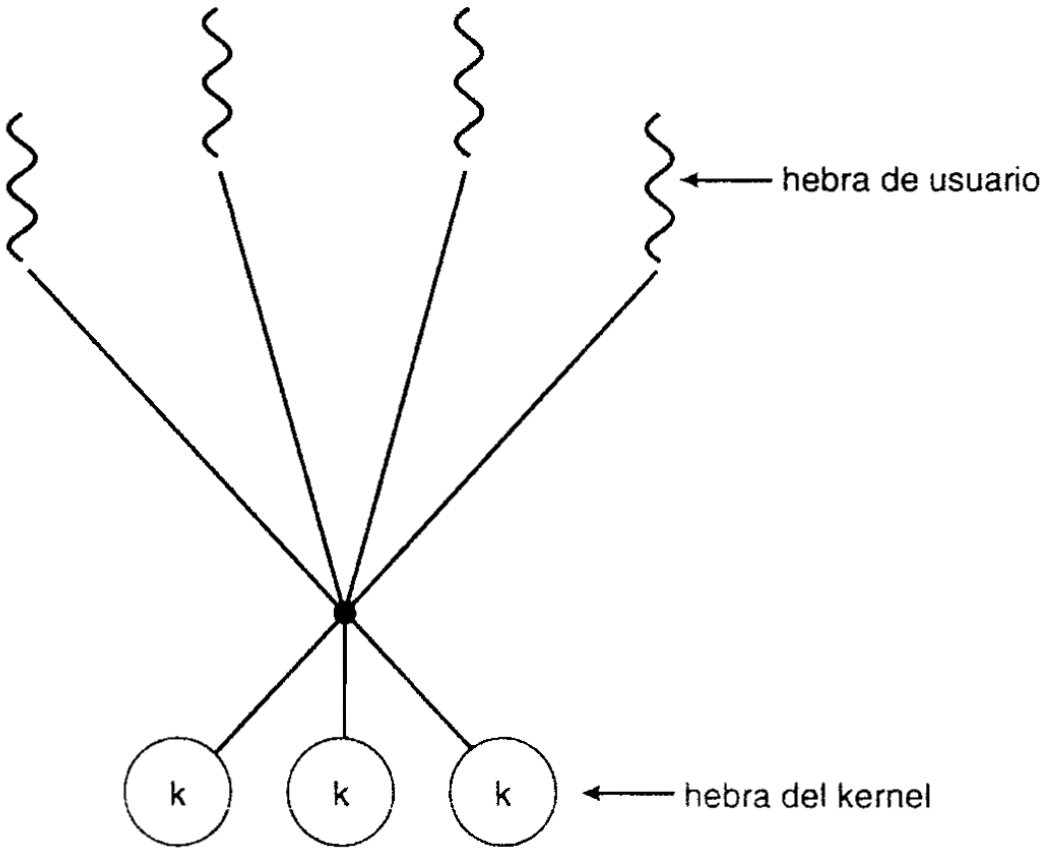
Modelo uno a uno

- Asigna cada hebra de usuario a una de kernel
- Proporciona mayor concurrencia que el anterior, se permiten hebras bloqueantes mientras se ejecutan otras
- Inconveniente: crear una hebra de usuario requiere crear la correspondiente en kernel, repercute en el rendimiento



Modelo muchos a muchos

- Multiplexa muchas hebras de usuario sobre un número menor o igual de hebras de kernel
- Se pueden crear tantas hebras de usuario como sea necesario y las correspondientes hebras del kernel pueden ejecutarse en paralelo en un multiprocesador.
- Cuando una hebra realiza una llamada bloqueante, el kernel planifica otra hebra



Hilos en modo usuario

- **Ventajas**
  - El núcleo no sabe que existen
  - Tabla de subprocesos probada para cambios de contexto
  - Cambio de contexto mucho mas rápido entre hilos (no se pasa al kernel)
  - Cada proceso puede tener su algoritmo de planificación
- **Inconvenientes**
  - Llamadas bloqueantes al sistema
  - Fallos de página
  - Tienen que ceder la CPU entre ellos => conmutación en el mismo proceso
  - Precisamente queremos hilos en procesos con muchas E/S para obtener paralelismo, es decir que se están bloqueando muy frecuentemente

Hilos en modo Kernel

- **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.
  - La creación y destrucción de procesos es mas costoso => reutilización de hilos

Bibliotecas de hebras

- Proporciona al programador una API para crear y gestionar hebras
- Pueden ser:
  - Espacio de usuario
  - Espacio de kernel
- Principales bibliotecas
  - POSIX (usuario o kernel)
  - Win32 (kernel)
  - Java (usuario)

Hilos en POSIX

- Estandar IEEE 1003.1C: **Pthreads**
- Cada llamada tiene ciertas propiedades:
  - Identificador
  - Conjunto de registros
  - Conjunto de atributos

Llamada de hilo	Descripción
Pthread_create	Crea un nuevo hilo
Pthread_exit	Termina el hilo llamador
Pthread_join	Espera a que un hilo específico termine
Pthread_yield	Libera la CPU para dejar que otro hilo se ejecute
Pthread_attr_init	Crea e inicializa la estructura de atributos de un hilo
Pthread_attr_destroy	Elimina la estructura de atributos de un hilo

Ejemplo de utilización de hilos

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

#define NUMERO_DE_HILOS 10

void *imprimir_hola_mundo(void *tid)
{
   /* Esta funcion imprime el identificador del hilo
   y después termina. */
   printf("Hola mundo. Saludos del hilo %d0, tid);
   pthread_exit(NULL);
}
```

Ejemplo de utilización de hilos (2)

```
int main(int argc, char *argv[])
{
   /* El programa principal crea 10 hilos y termina. */
   pthread_t hilos[NUMERO_DE_HILOS];
   int estado, i;
   for(i=0; i < NUMERO_DE_HILOS; i++) {
     printf("Aqui main. Creando hilo %d0, i);
     estado = pthread_create(%hilos[i], NULL,
              imprimir_hola_mundo, (void *)i);
     if (estado != 0) {
         printf("Ups. pthread_create devolvió el codigo
                 de error %d0", estado);
         exit(-1);
     }
   }
   exit(NULL);
}
```

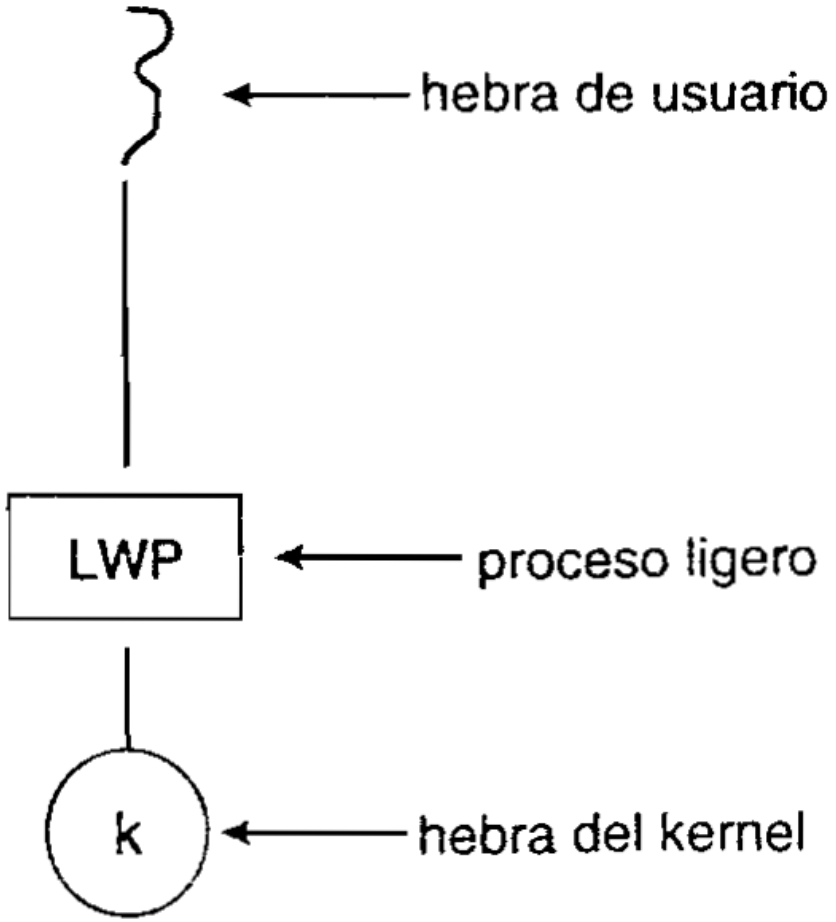
Activaciones del planificador

Edler (1988) y Scott (1990)

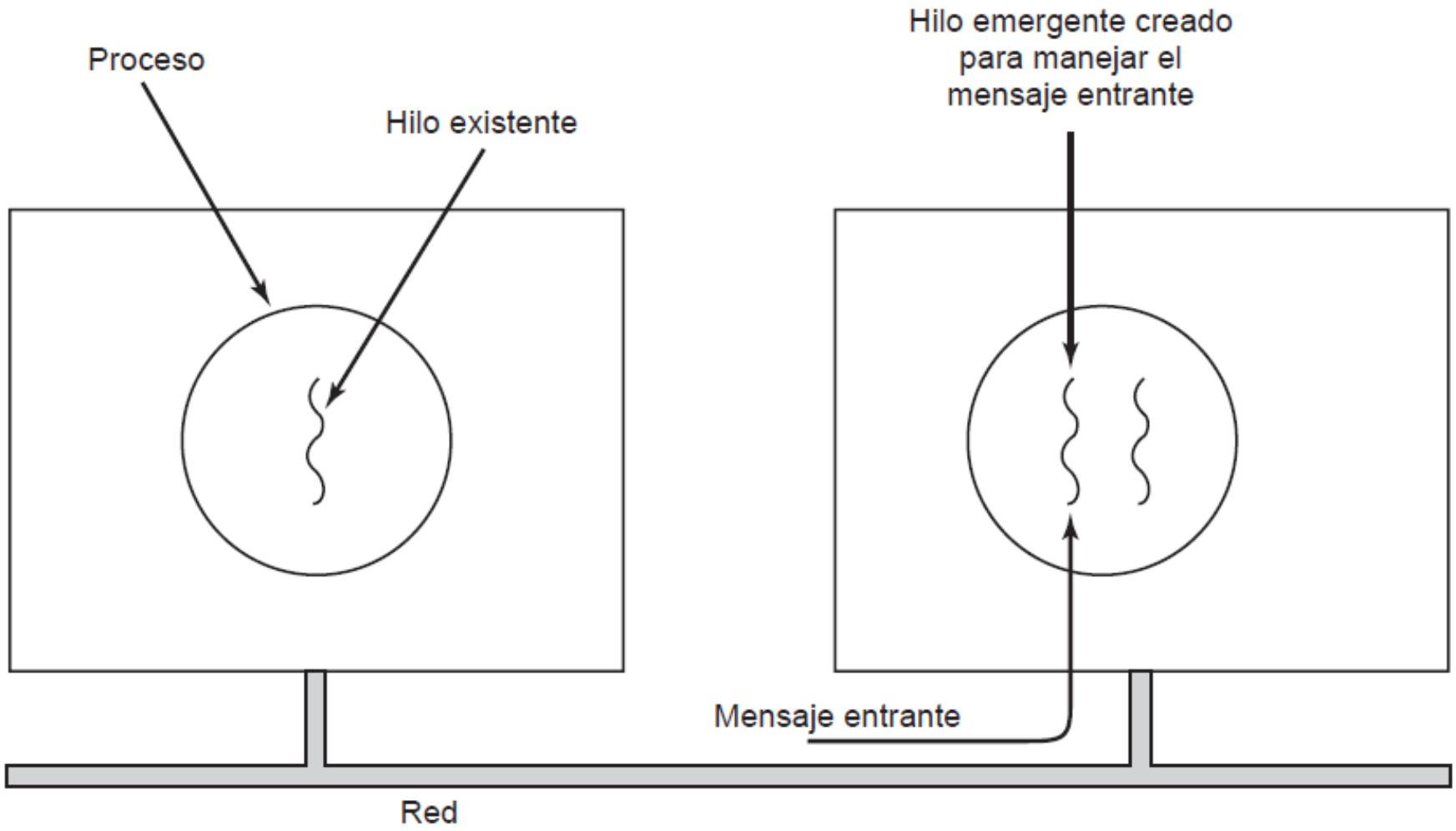
- Los hilos de kernel son mejores que los de usuario, pero también son mas lentos
- El **objetivo** de una activación del planificador es **imitar la funcionalidad de los hilos de kernel**, pero con el **mejor rendimiento** y la **mayor flexibilidad**
- La eficiencia se obtiene evitando transacciones innecesarias entre los espacios de usuario y de kernel.

Activaciones del planificador (2)

- Los sistemas que implementan el modelo muchos-a-muchos colocan una estructura de datos intermedia entre las hebras de usuario y del kernel llamada **LWP (lightweight process)**
- Cada procesos ligero es visto como un **procesador virtual**.
- Cada LWP se asocia a una hebra de kernel (que se ejecuta en el procesador físico)



Hilos emergentes



Antes de que llegue el mensaje

Después de que llega el mensaje

Hilos emergentes (2)

- Ventajas
  - Como son nuevos no tienen historial que sea necesario restaurar.
  - Se crean con rapidez.
  - La latencia entre la llegada del mensaje y el inicio del procesamiento puede ser baja.
- Desventaja
  - Es necesaria cierta planeación anticipada
- Hacer que el hilo emergente se ejecute en el espacio del kernel es mas rápido y sencillo. Puede acceder con facilidad a todas las tablas del kernel y a los dispositivos de E/S

Diferencias entre hilos y procesos

- Procesos conlleva gran cantidad de información de estados.
- Los hijos se comunican mas fácilmente.
- Los hilos comparten recursos, datos y espacio de direcciones.
- Tiempo de cambio de proceso es elevado (cambio de contexto). Los hilos usan el mismo contexto del proceso.
- Complejidad utilizando hilos es mas elevada (protecciones de variables)

Diferencias entre hilos y procesos

En resumen:

	Procesos	Hilos
Creación	Costosa	Ligera
Recursos y memoria	Independiente	Compartida
Comunicación	Compleja	Sencilla
Cambio por SO	Muy lento	Rápido
Complejidad en programación	Reducida	Alta

Bibliografía

- Cap 2, tanembaum "Sistemas Operativos modernos"
- Cap 3 y 4, Silberschatz "Fundamentos de sistemas operativos"
- Cap 3 y 4, William Stallings "Sistemas Operativos"