


# TEMA 1

# Introducción a la Programación Concurrente

# Introducción a la Programación Concurrente

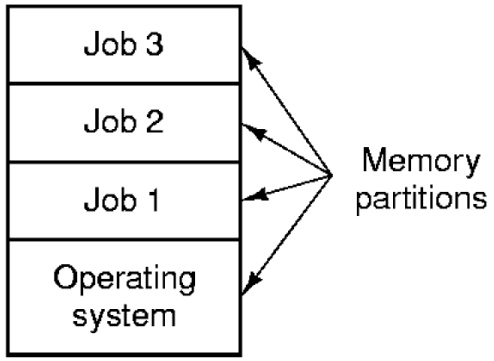
1. Introducción
2. Concepto de la programación concurrente
3. Ventajas y desventajas de la programación concurrente
4. Arquitecturas Hardware para la concurrencia
5. Procesos vs Hilos
6. Especificaciones de ejecución concurrente
7. Características de los sistemas concurrentes
8. Problemas inherentes a la programación concurrente
9. Corrección de programas concurrentes

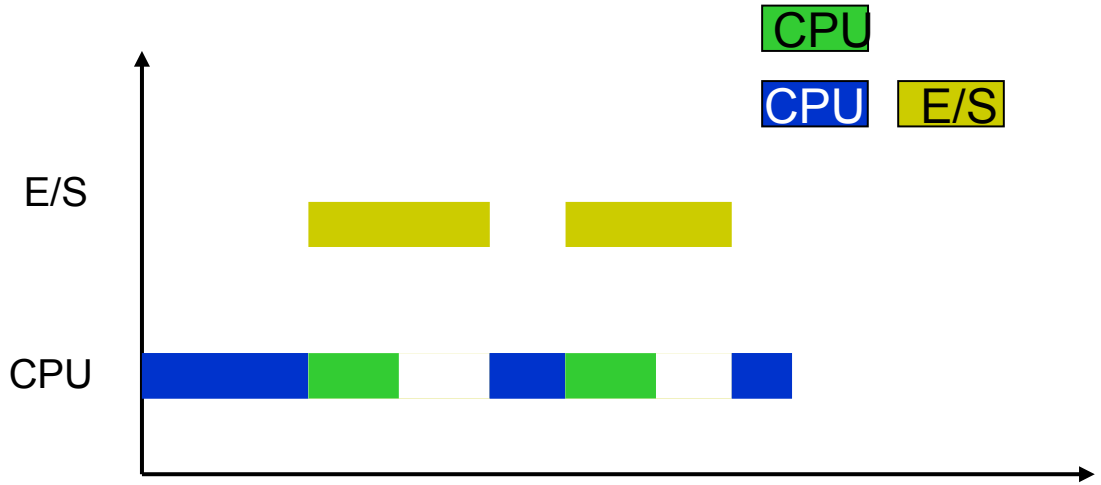
# 1. Introducción

- Concurrencia (R.A.E): ***Coincidencia, concurso simultáneo de varias circunstancias***.
- La programación concurrente surge con la ***Multiprogramación***.
- Incialmente solo se hacia en ensamblador, hasta 1972, cuando **Brinch Hansen** crea el lenguaje de alto nivel *Concurrent Pascal*, con capacidad de manejar la concurrencia.

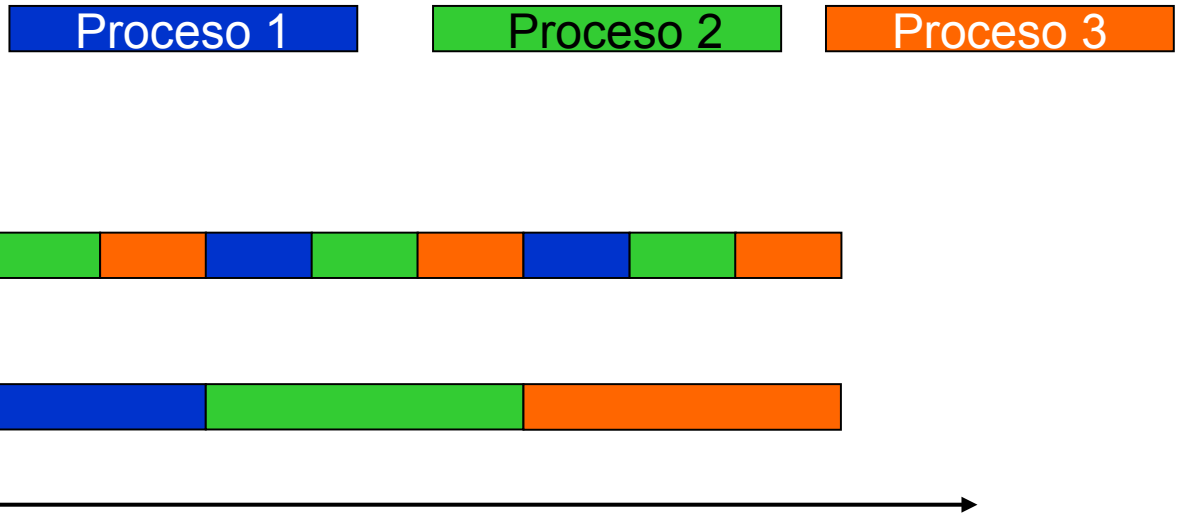
# 1. Introducción

## MULTIPROGRAMACIÓN




## TIEMPO COMPARTIDO



# 1. Introducción

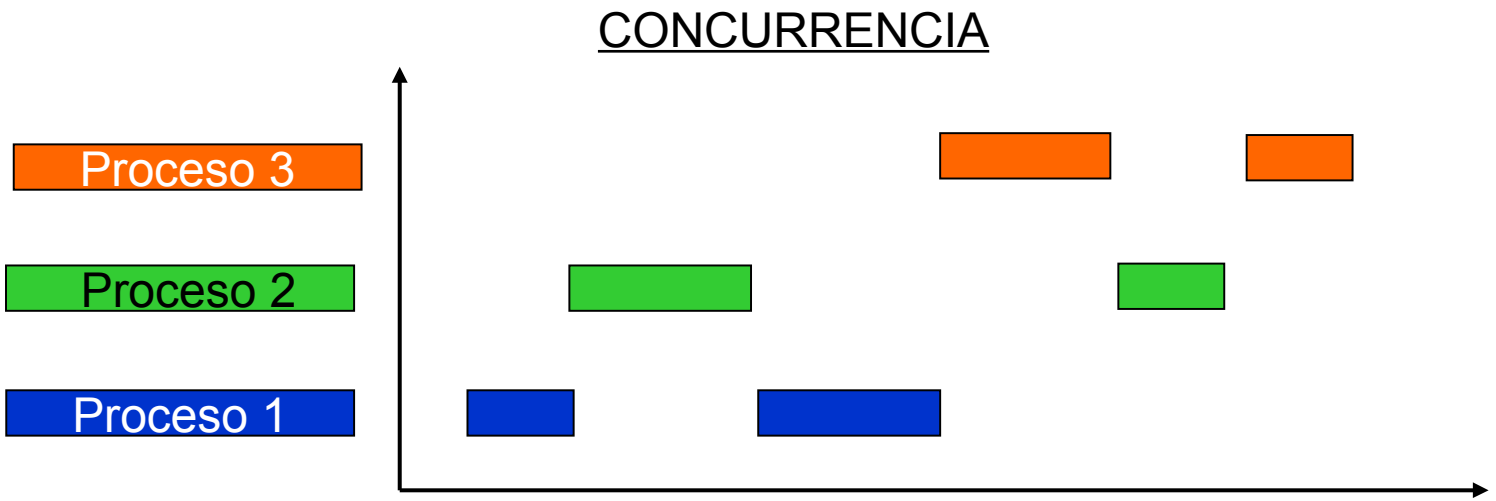
- A partir de entonces la programación concurrente toma cada vez más relevancia debido a tres **hitos**.
  - La aparición del concepto de **hilo** (hebra, thread, proceso ligero)
  - La aparición de **Java** que da soporte a la concurrencia
  - La aparición de **Internet**

# 2. Concepto de Programación Concurrente

- Concurrencia: ***Acontecimiento o concurso de varios sucesos en un mismo tiempo***.
- Cambiando suceso por **proceso** tenemos una primera aproximación.
- **¿Qué es un proceso?**
  - Puede entenderse como una instancia de un programa en ejecución, ….

# 2. Concepto de Programación Concurrente

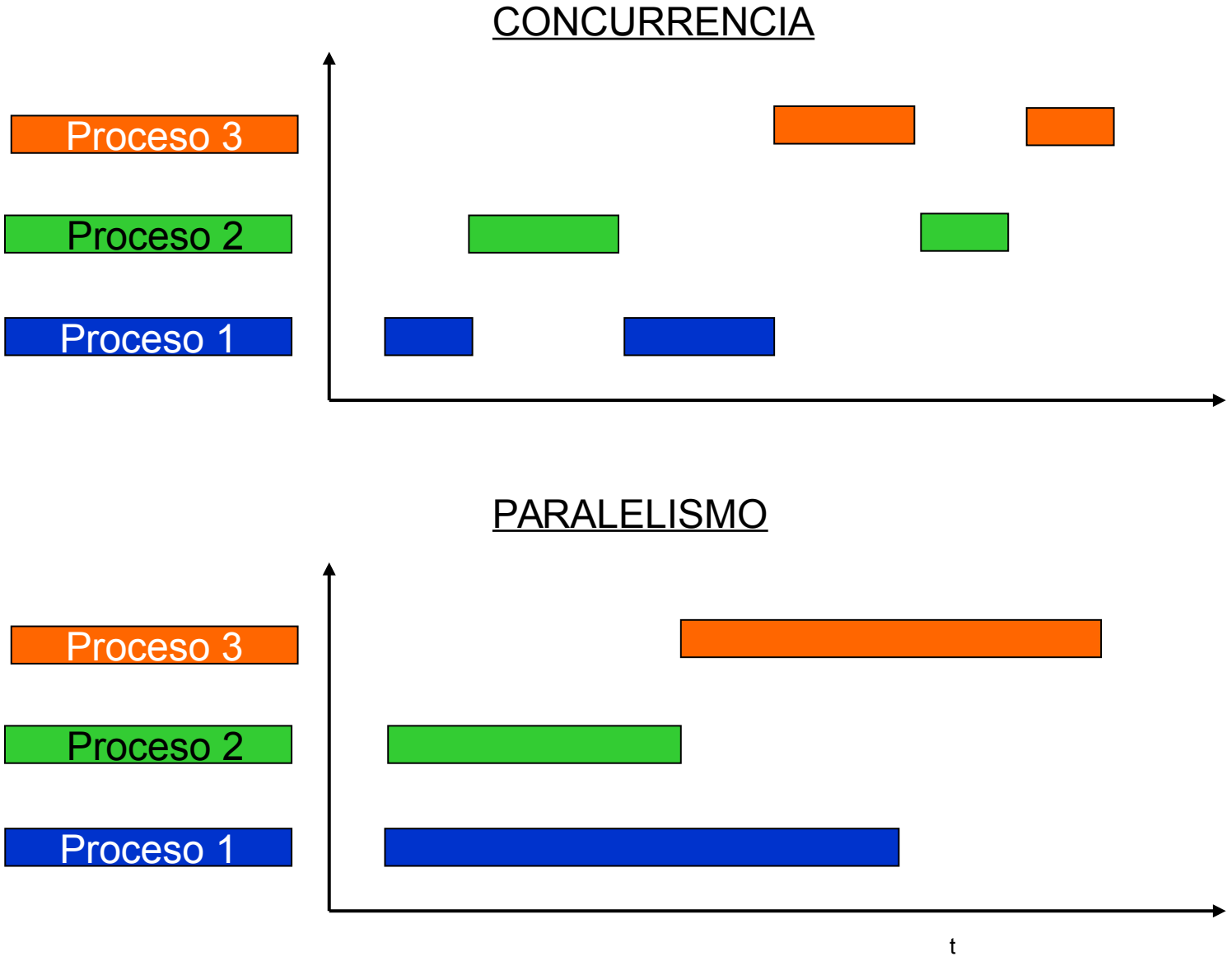
- Dos **procesos** serán **concurrentes** cuando la primera instrucción de uno de ellos se ejecuta después de la primera del otro y antes de la última.



# 2. Concepto de Programación Concurrente

- En la **Programación Paralela**, las instrucciones de dos o más procesos se ejecutan **a la vez**. Es necesario Hardware adecuado.
- Los procesos concurrentes pueden **colaborar** y/o **competir** por los recursos. Para llevar a cabo esa colaboración hacen falta **mecanismos de comunicación y sincronización entre procesos**, y de ellos se encarga la Programación Concurrente.

# 2. Concepto de Programación Concurrente



# 3. Ventajas y desventajas de la Programación Concurrente

## VENTAJAS

- **Mejor utilización de los recursos**
  - Imagine una aplicación que lee y procesa archivos. Supongamos que la lectura de un archivo tarda 5 segundos y el procesamiento que se necesita 2 segundos.

```
  5 seconds reading file A
  2 seconds processing file A
  5 seconds reading file B
  2 seconds processing file B
----------------------
 14 seconds total
```

```
  5 seconds reading file A
  5 seconds reading file B + 2 seconds processing file A
  2 seconds processing file B
----------------------
 12 seconds total
```

  - Recordemos que la CPU está prácticamente inactiva mientras espera la lectura del disco

# 3. Ventajas y desventajas de la Programación Concurrente

## VENTAJAS

- **Diseño más simple de las aplicaciones**
  - Para el ejemplo anterior, en lugar de controlar la lectura y procesamiento de los dos ficheros, se pueden crear dos hilos, cada uno de los cuales se encargue de leer y procesar un fichero
- **Velocidad de Ejecución**
  - Sobre todo cuando hay paralelismo
- **Solución a problemas concurrentes**:
  - Sistemas de Control
  - Tecnologías WEB
  - Aplicaciones con interfaces de usuario
  - Simulaciones
  - SGBD

# 3. Ventajas y desventajas de la Programación Concurrente

## VENTAJAS

- **Mejor tiempo de respuesta de las aplicaciones**
  - Imagine una aplicación de servidor que escucha en un puerto para recibir solicitudes. Cuando se recibe una petición, se controla la solicitud y luego vuelve a escuchar.

```
while(server is active){
  listen for request
  process request
}
```

```
while(server is active){
  listen for request
  hand request to worker thread
}
```

  - Un diseño alternativo sería que el subproceso de escucha para aprobar la solicitud, se la pasa a un hilo, y volver a escuchar. El hilo procesará la solicitud y enviará una respuesta al cliente.
  - Lo mismo ocurre con las aplicaciones de consola, por ejemplo para tareas largas tras la pulsación de un botón.

# 3. Ventajas y desventajas de la Programación Concurrente

## DESVENTAJAS

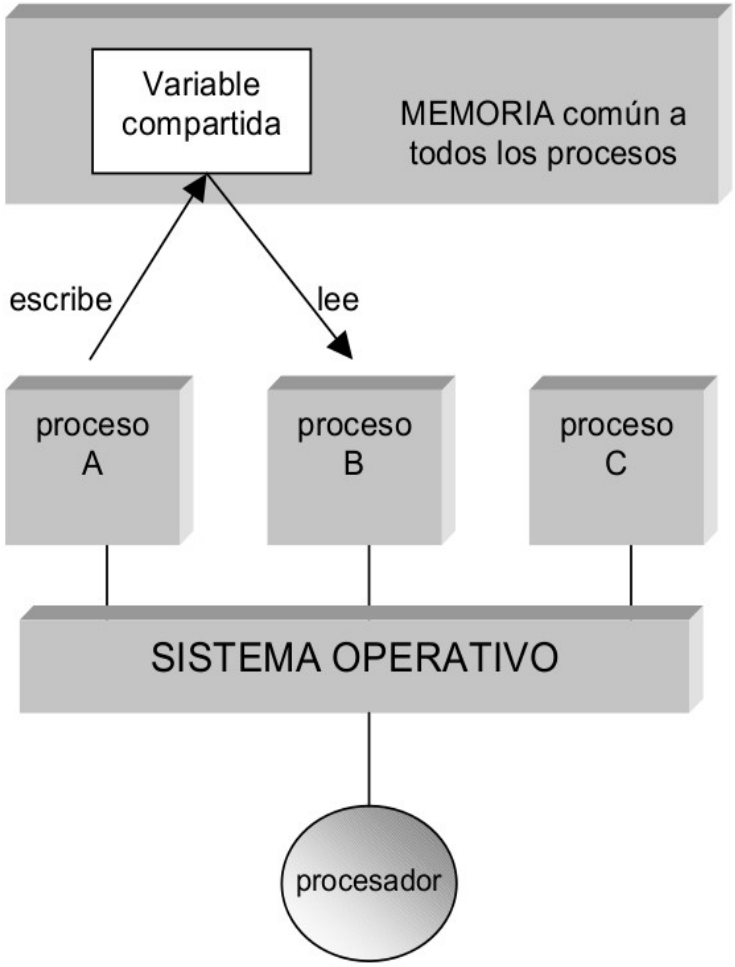
- **Diseño más complejo de las aplicaciones**
  - Aunque algunas partes del diseño puedan ser más simples, otras son más complejas, sobre todo cuando las tareas deben sincronizarse para acceder a recursos compartidos.
- **Sobrecarga en los cambios de contexto**
  - Aunque los cambios de contexto de hilos dentro de un mismo proceso son más rápidos que los cambios de contexto de un proceso, al haber más hilos hay más cambios de contexto, y si son de distintos procesos el tiempo se incrementa.
- **Incremento en el consumo de recursos**:
  - Los hilos necesitan, además de CPU, memoria para mantener su propia pila local. Además, el sistema operativo necesita recursos para manejar dichos hilos

# 4. Arquitecturas Hardware para concurrencia

- Cuando hablamos de hardware nos estamos refiriendo fundamentalmente al **número de procesadores** en el sistema.
- Así, se puede hacer una primera distinción entre aquellos sistemas donde sólo hay un procesador, **sistemas monoprocesador**, y aquellos en los que hay más de un procesador, **sistemas multiprocesador**. En ambos sistemas es posible tener concurrencia.

# 4. Arquitecturas Hardware para concurrencia

- **Sistemas monoprocesador**. Hacen uso de la *Multiprogramación* o el *Tiempo Compartido* para la ejecución concurrente de procesos.
  - En los sistemas monoprocesador **todos los procesos comparten la misma memoria**. Así pues la comunicación puede hacerse mediante memoria compartida.



# 4. Arquitecturas Hardware para concurrencia

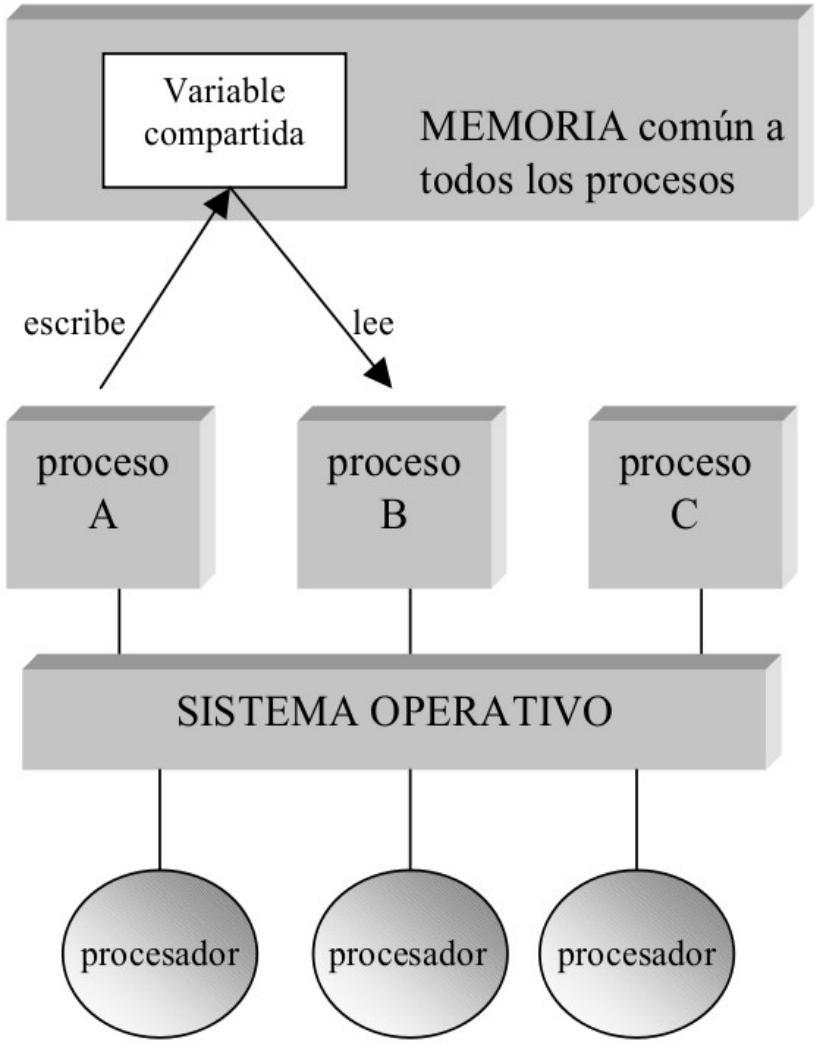
- **Sistemas monoprocesador**
- Aparte de la situación en la que un proceso puede aprovechar ciclos de CPU mientras otro proceso hace operaciones de entrada/salida, existen otros posibles **beneficios** del uso de concurrencia en sistemas monoprocesador:
  - La posibilidad de proporcionar un **servicio interactivo a múltiples usuarios**. Este sería el caso por ejemplo de un Sistema Operativo multiusuario ejecutándose sobre una máquina monoprocesador.
  - La posibilidad de dar una solución adecuada a **problemas que son de naturaleza eminentemente concurrente**

# 4. Arquitecturas Hardware para concurrencia

- **Sistemas Multiprocesador**. Puede haber paralelismo, aunque es común que haya más procesos que procesadores. Se clasifican en:
  - **Sistemas fuertemente acoplados**: tanto los procesadores como otros dispositivos (incluida la memoria) están conectados a un bus. Esto permite que **todos los procesadores puedan compartir la misma memoria**. Puede ocurrir que cada procesador tenga su propia memoria local, pero la sincronización y comunicación entre procesos se hará mediante variables situadas en la memoria compartida, es decir, mediante variables.

# 4. Arquitecturas Hardware para concurrencia

- **Sistemas Multiprocesador**. Puede haber paralelismos, aunque es común que haya más procesos que procesadores. Se clasifican en:
  - **Sistemas fuertemente acoplados**:

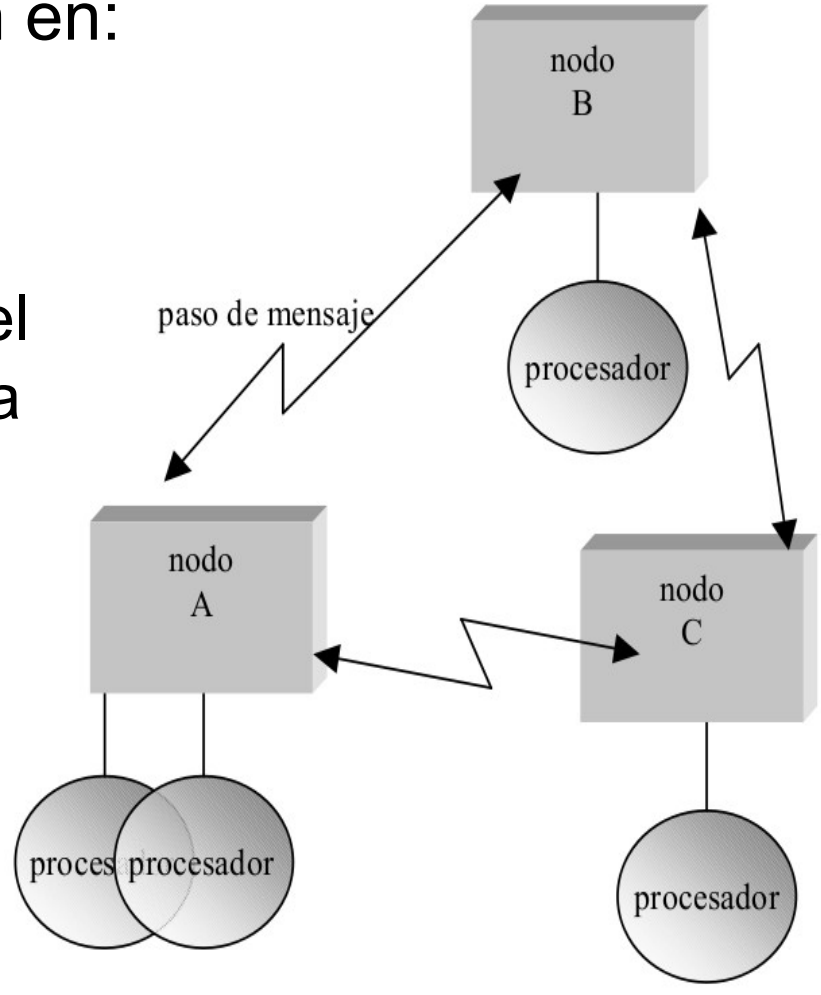


# 4. Arquitecturas Hardware para concurrencia

- **Sistemas Multiprocesador**. Puede haber paralelismos, aunque es común que haya más procesos que procesadores. Se clasifican en:
  - **Sistemas débilmente acoplados**: aquí no existe una memoria compartida por los procesadores. **Cada procesador tiene su propia memoria local y está conectado con otros procesadores mediante algún tipo de enlace de comunicación**. Un tipo especial de estos sistemas lo constituyen los **sistemas distribuidos**, que están formados por un conjunto de nodos distribuidos geográficamente y conectados de alguna forma. Estos nodos pueden ser a su vez mono o multiprocesador.

# 4. Arquitecturas Hardware para concurrencia

- **Sistemas Multiprocesador**. Puede haber paralelismos, aunque es común que haya más procesos que procesadores. Se clasifican en:
  - **Sistemas débilmente acoplados**:
  - El **paso de mensajes** es el mecanismo empleado para comunicar y sincronizar procesos en sistemas distribuidos. Se puede usar también en multiproceso y multiprogramación.



# 4. Arquitecturas Hardware para concurrencia

- **Multiproceso**: Gestión de varios procesos dentro de un sistema **multiprocesador**, donde cada procesador puede acceder a una **memoria común**
- **Procesamiento Distribuido**: Gestión de varios procesos en **procesadores separados**, cada uno con su **memoria local**.
- **Programa concurrente**: Conjunto de acciones que pueden ejecutarse simultaneamente.
- **Programa paralelo**: Programa concurrente ejecutado en un sistema multiprocesador.
- **Programa distribuido**: Programa paralelo que se ejecuta en un sistema distribuido

# 5. Procesos VS Hilos

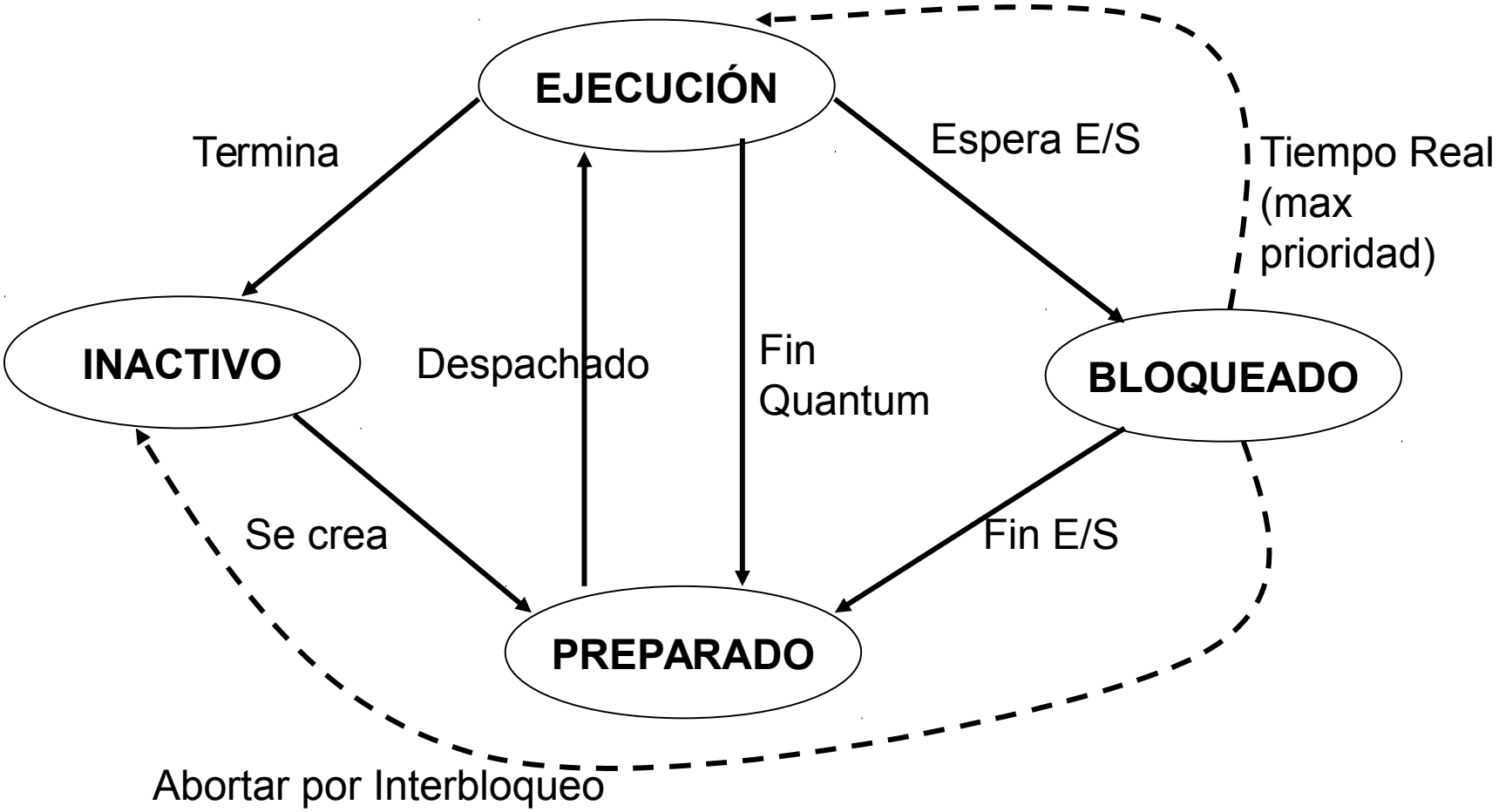
- Un proceso es un ***programa en ejecución*** que comprende el valor actual del contador de programa, de los registros y de las variables, siendo además la unidad de propiedad de los recursos y la unidad de expedición.
- También podemos decir que un **proceso** es una instancia de un programa en ejecución, por tanto es una **entidad dinámica**, al contrario del concepto de **programa** que es una **entidad estática**.
- Normalmente el procesador está conmutando de un proceso a otro, pero el sistema es mucho más fácil de entender si lo consideramos como un conjunto de procesos que se ejecutan quasi en paralelo.

# 5. Procesos VS Hilos

- Un proceso puede pasar por los siguientes **estados**:
  - **Inactivo**. Consideraremos así a los programas, es decir, al proceso que aún no ha sido ejecutado, y por tanto no ha sido creado
  - **Preparado**. Proceso activo que posee todos los recursos necesarios para ejecutarse, pero al que le falta, precisamente, el procesador.
  - **Ejecución**. Proceso que está siendo ejecutado por el procesador. Con un solo procesador sólo puede haber un proceso en ejecución.
  - **Bloqueado**. Proceso que además de no contar con el procesador requiere algún otro recurso del sistema.

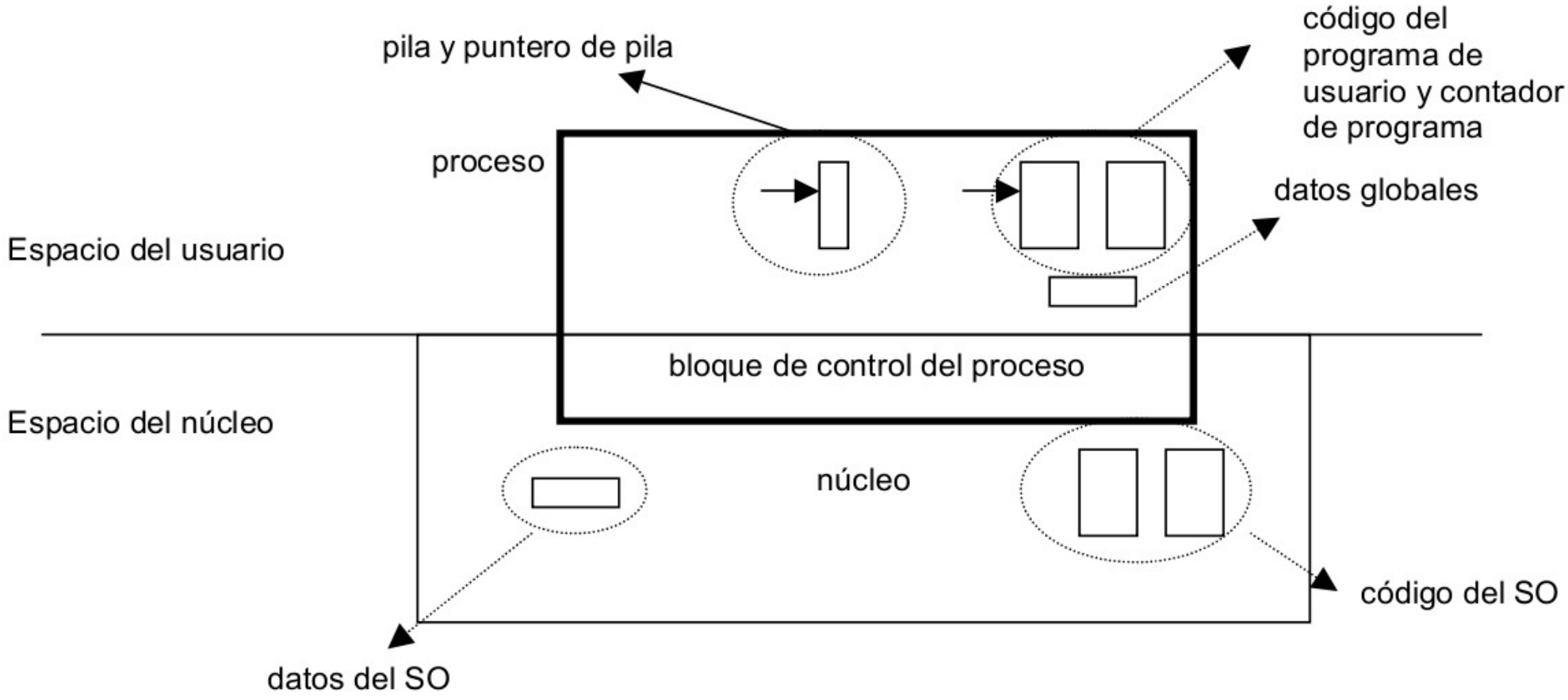
# 5. Procesos VS Hilos

- Un proceso puede pasar por los siguientes **estados**:



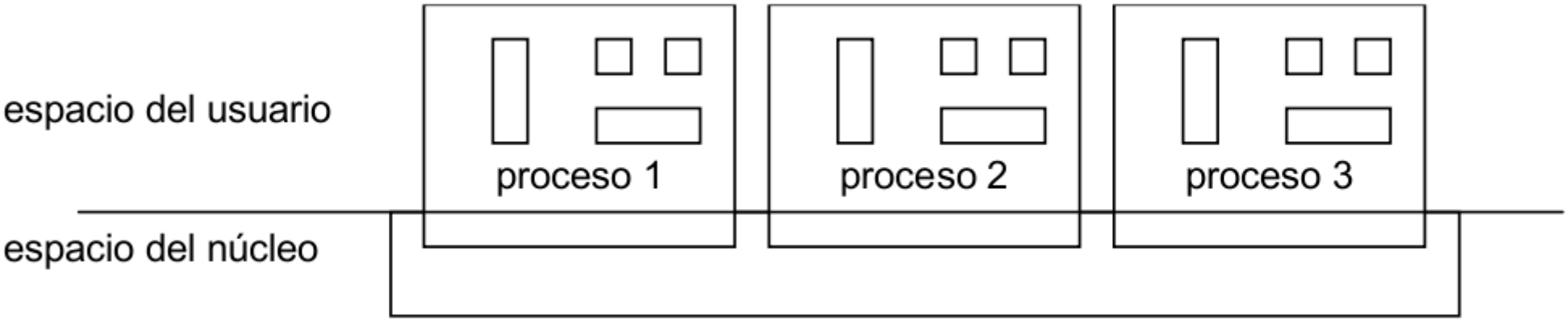
# 5. Procesos VS Hilos

- **Disposición en memoria de un proceso**



# 5. Procesos VS Hilos

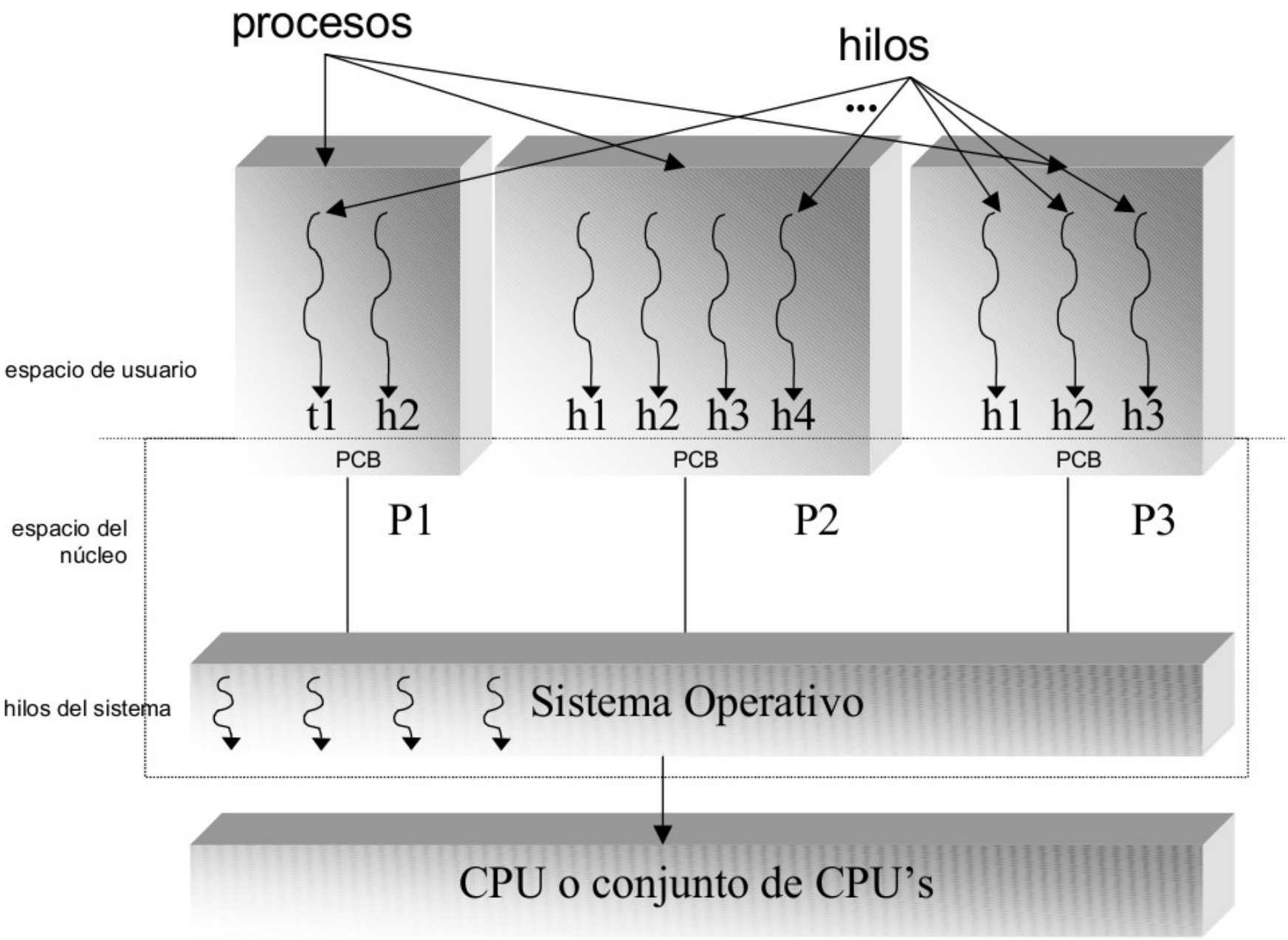
- **Disposición en memoria varios procesos en un Sistema Operativo Multitarea**



# 5. Procesos VS Hilos

- Pero, **¿qué es un hilo?**.
- De la misma manera que un Sistema Operativo puede ejecutar varios procesos al mismo tiempo bien sea por concurrencia o paralelismo, dentro de un proceso puede haber varios hilos de ejecución.
- Por tanto, un hilo puede definirse como *cada secuencia de control dentro de un proceso que ejecuta sus instrucciones de forma independiente*.

# 5. Procesos VS Hilos



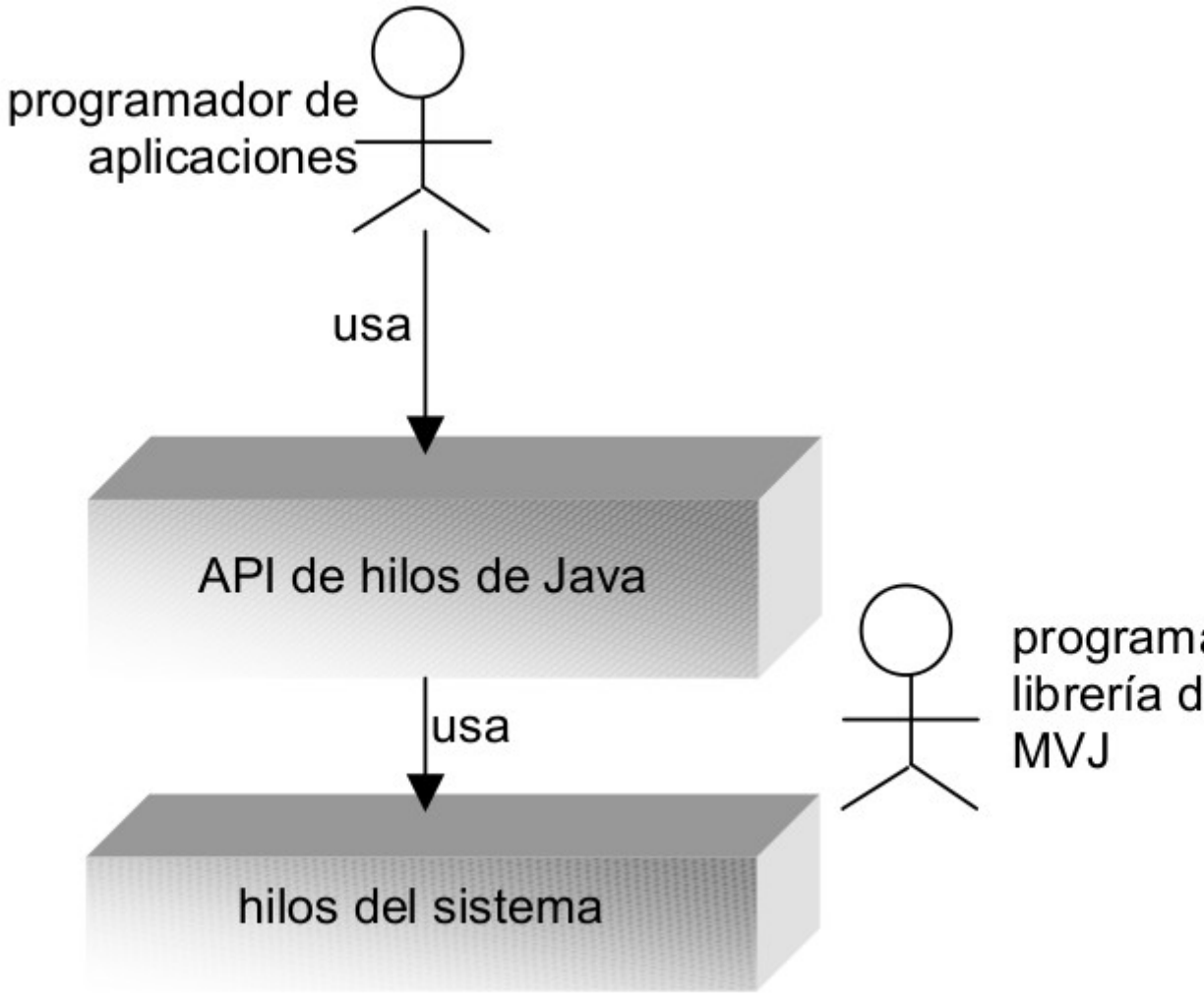
# 5. Procesos VS Hilos

- Los procesos son **entidades pesadas**.
  - La estructura del proceso está en la parte del núcleo y, cada vez que el proceso quiere acceder a ella, tiene que hacer algún tipo de llamada al sistema, consumiendo tiempo extra de procesador. Los **cambios de contexto** entre procesos son **costosos**.
- La estructura de los hilos reside en el espacio de usuario, con lo que un hilo es una **entidad ligera**.
  - Los hilos comparten la información del proceso (código, datos, etc). Si un hilo modifica una variable del proceso, el resto de hilos verán esa modificación cuando accedan a esa variable. Los **cambios de contexto entre hilos consumen poco tiempo de procesador**, de ahí su éxito.

# 5. Procesos VS Hilos

- Podemos hablar de **dos niveles de hilos**:
  - aquellos que nosotros usaremos para programar y que pueden crearse desde lenguajes de programación como Java y
  - aquellos otros hilos del propio SO que sirven para dar soporte a nuestros hilos de usuario y que son los hilos del sistema.
- Cuando nosotros programamos con hilos, hacemos uso de un **API** (ApplicationProgram Interface) proporcionado por el lenguaje, el SO o un tercero mediante una librería externa. El implementador de este API será el que haya usado los hilos del sistema para dar soporte de ejecución a los hilos que nosotros creamos en nuestros programas.

# 5. Procesos VS Hilos



# 5. Procesos VS Hilos

- Cada SO implementa los hilos del sistema como quiere, aunque se puede hablar de la existencia de tres **estándares**: .NET, OS/2 y POSIX.
- Las dos primeras son propietarias y sólo corren bajo sus respectivas plataformas (Windows, OS/2).
- La especificación **POSIX** (IEEE 1003.1c) conocida como **pthreads** [Butenhof, 1997] está pensada para todas las plataformas y está disponible para la mayoría de las implementaciones UNIX y Linux.

# 5. Procesos VS Hilos

- Los **hilos de Java** están implementados en la **MVJ** (Máquina Virtual Java) que a su vez está construida sobre la librería de hilos nativas de la correspondiente plataforma.
- Java sólo ofrece su **API** para manejar hilos, independientemente de la librería subyacente. De esta forma es mucho más fácil utilizar las hilos de Java. Sin embargo, y como se verá posteriormente, hay algunos asuntos importantes a tener en cuenta para que un programa multihilo en Java sea independiente de la plataforma.

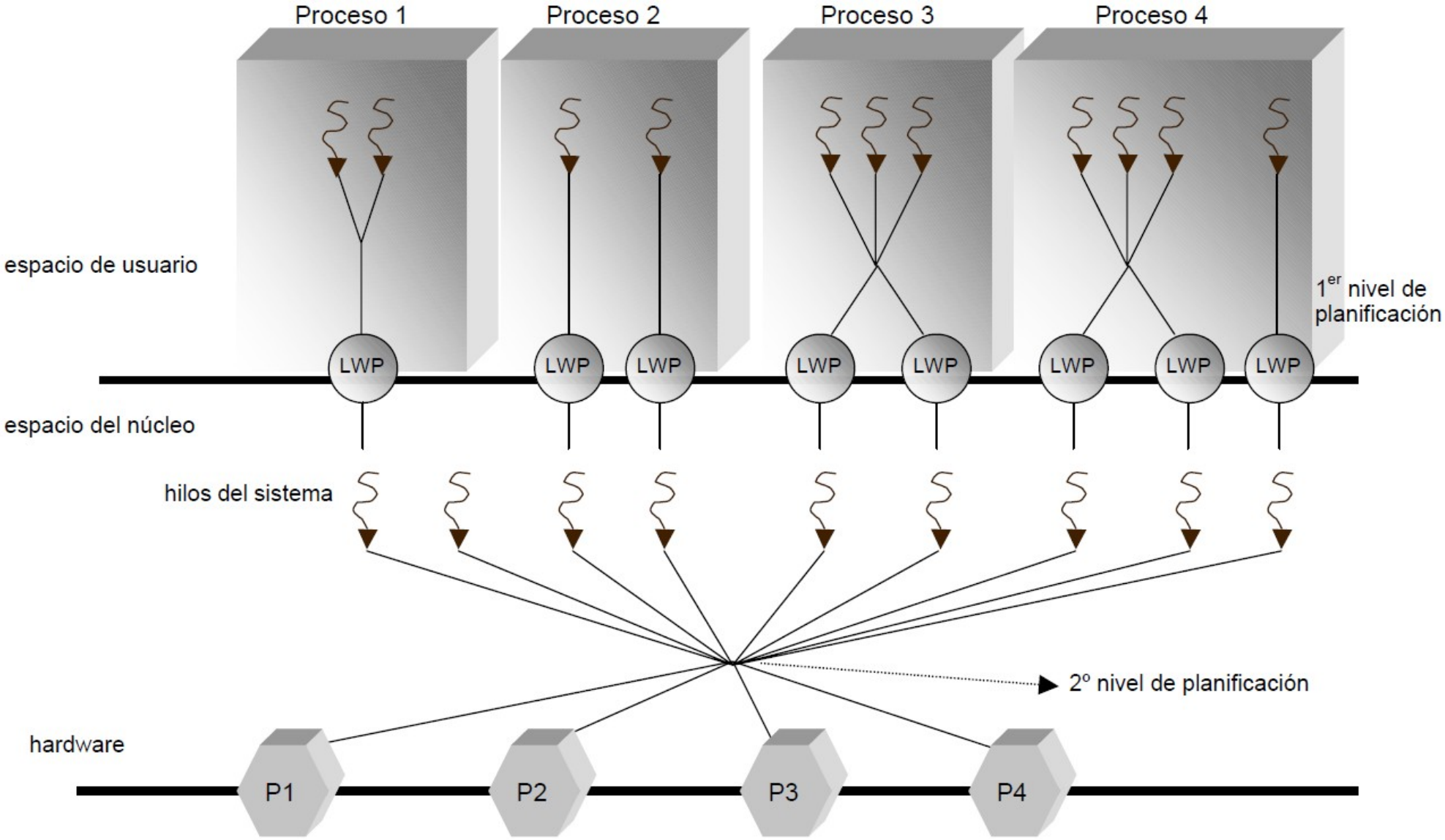
# 5. Procesos VS Hilos

- Hay fundamentalmente 2 formas de **implementar hilos**.
  - La primera es escribir una **librería al nivel de usuario**. Todas las estructuras y el código de la librería estarán en espacio de usuario. La mayoría de las llamadas que se hagan desde la librería se ejecutarán en el espacio de usuario y no hará más uso de las rutinas del sistema que cualquier otra librería o programa de usuario.
  - La segunda forma es una **implementación al nivel de núcleo**. La mayoría de las llamadas de la librería requerirán llamadas al sistema.
  - Algunas de las implementaciones del estándar POSIX son del primer tipo, mientras que tanto OS/2 como Win32 son del segundo tipo

# 5. Procesos VS Hilos

- **Planificación de Hilos**
- Los SSOO modernos como Solaris definen el concepto de **procesador lógico** de tal forma que donde se ejecutan los hilos de usuario es en un procesador lógico. En la terminología de Solaris a estos procesadores lógicos se les denomina **LWP** (Light-Weight Processes)
- De esta forma vamos a tener una **planificación a dos niveles**. Un primer nivel para asignar los hilos de usuario a los procesadores lógicos y otro para asignar los procesadores lógicos al procesador o procesadores físicos.

# 5. Procesos VS Hilos



# 5. Procesos VS Hilos

- Hay principalmente tres técnicas distintas para hacer la **planificación de hilos** :

1. **Muchos hilos en un procesador lógico** Conocido como el modelo **muchos-a-uno**.Todos los hilos creados en el espacio de usuario hacen turnos para ir ejecutándose en el único procesador lógico asignado a ese proceso.

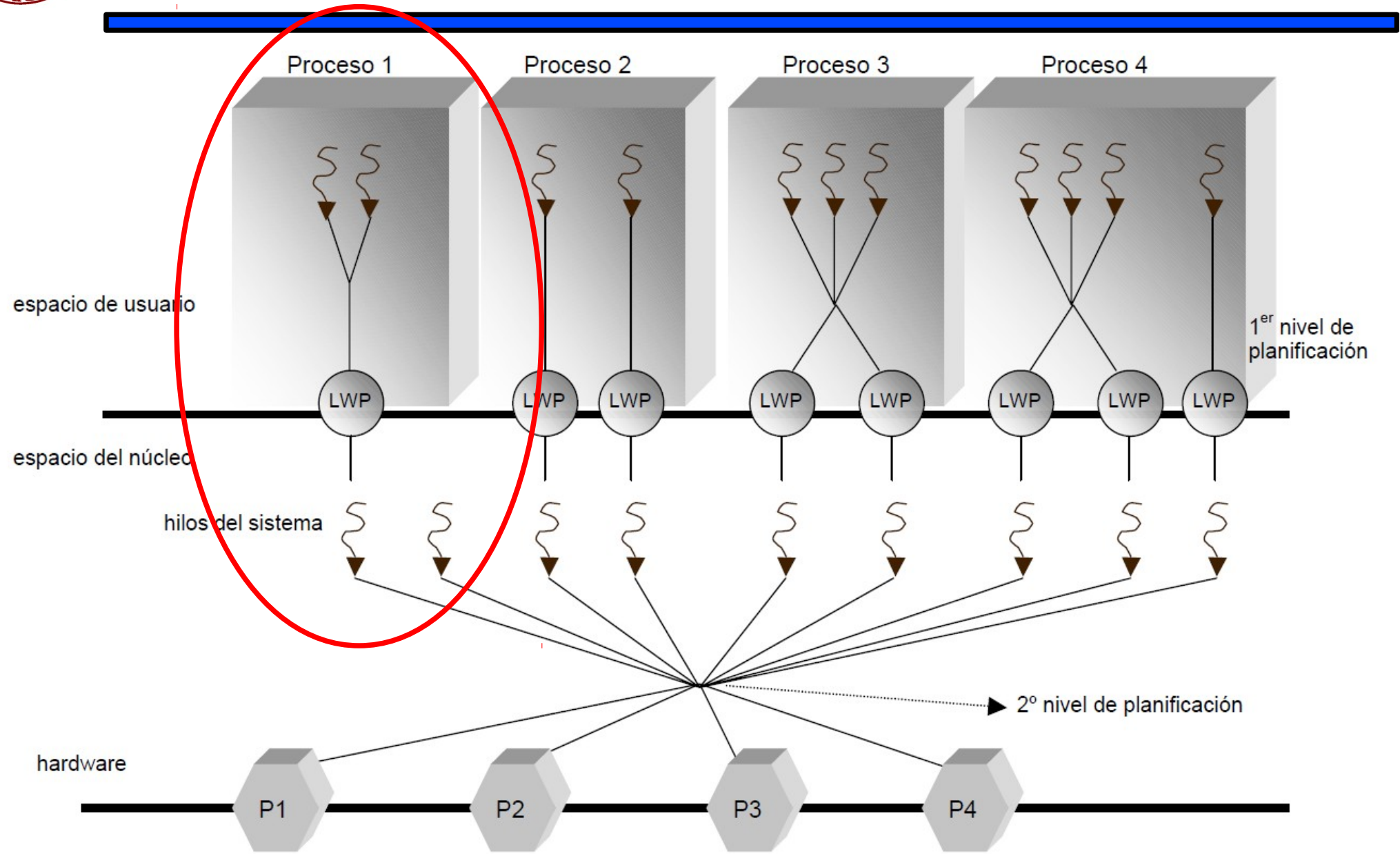
   **Desventajas**:

   - ✗ Un proceso no toma ventaja de una máquina con varios procesadores físicos.
   - ✗ Cuando se hace una llamada bloqueante, p. ej. de E/S, todo el proceso se bloquea.

   **Ventaja**:

   - ✓ La creación de hilos y la planificación se hacen en el espacio de usuario al 100%, sin usar los recursos del núcleo. Por tanto es más rápido.

# 5. Procesos VS Hilos



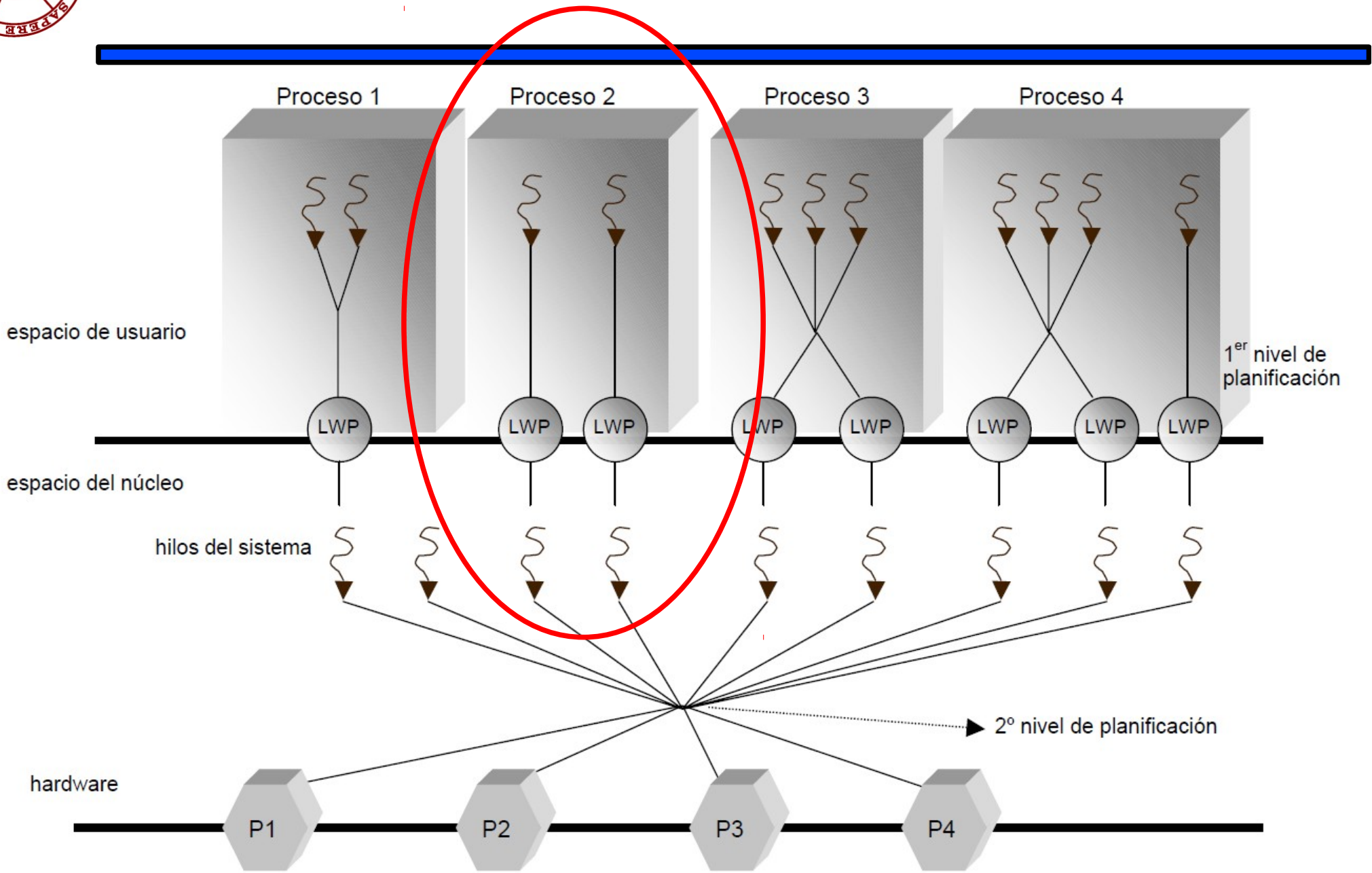
# 5. Procesos VS Hilos

- Hay principalmente tres técnicas distintas para hacer la **planificación de hilos**:

  2. **Un hilo por procesador lógico** También llamado como **modelo uno-a-uno**. Aquí se asigna un procesador lógico para cada hilo de usuario.

     - ✓ **Ventaja**: permite que muchos hilos se ejecuten simultáneamente en diferentes procesadores físicos.
     - ✗ **Desventaja**: la creación de hilos conlleva la creación de un procesador lógico y por tanto una llamada al sistema. Como cada procesador lógico toma recursos adicionales del núcleo, uno está limitado en el número de hilos que puede crear. Win32 y OS/2 utilizan este modelo. Cualquier MVJ basada en estas librerías también usa este modelo, así pues Java sobre Win32 lo usa.

# 5. Procesos VS Hilos



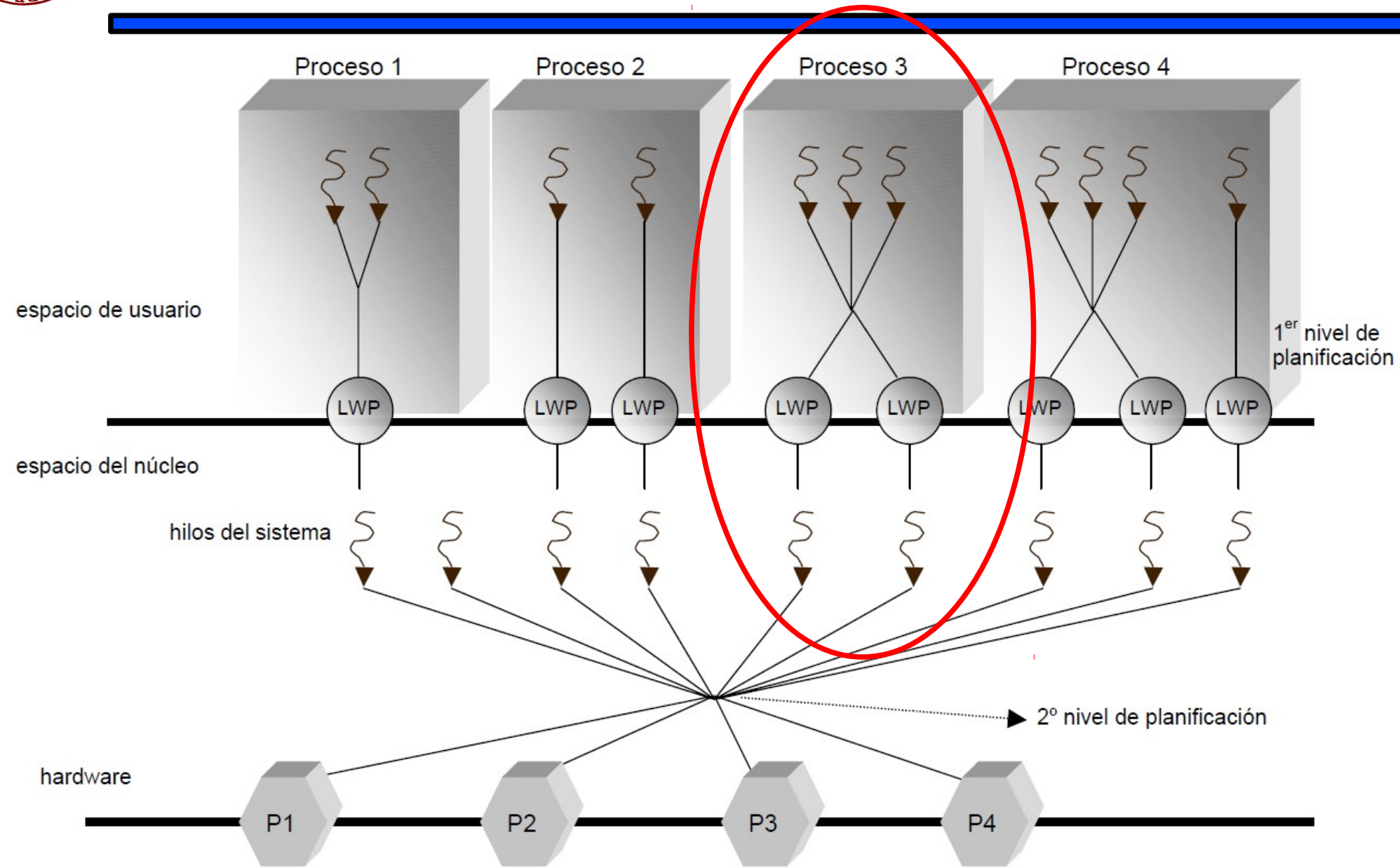
# 5. Procesos VS Hilos

- Hay principalmente tres técnicas distintas para hacer la **planificación de hilos** :

  3. **Muchos hilos en muchos procesadores lógicos**. Llamado modelo **muchos-a-muchos (estricto)**. Un número de hilos es multiplexado en un número de procesadores lógicos igual o menor.

     Numerosos hilos pueden correr en paralelo en diferentes CPUs y las llamadas al sistema de tipo bloqueante no bloquean al proceso entero.

# 5. Procesos VS Hilos



# 5. Procesos VS Hilos

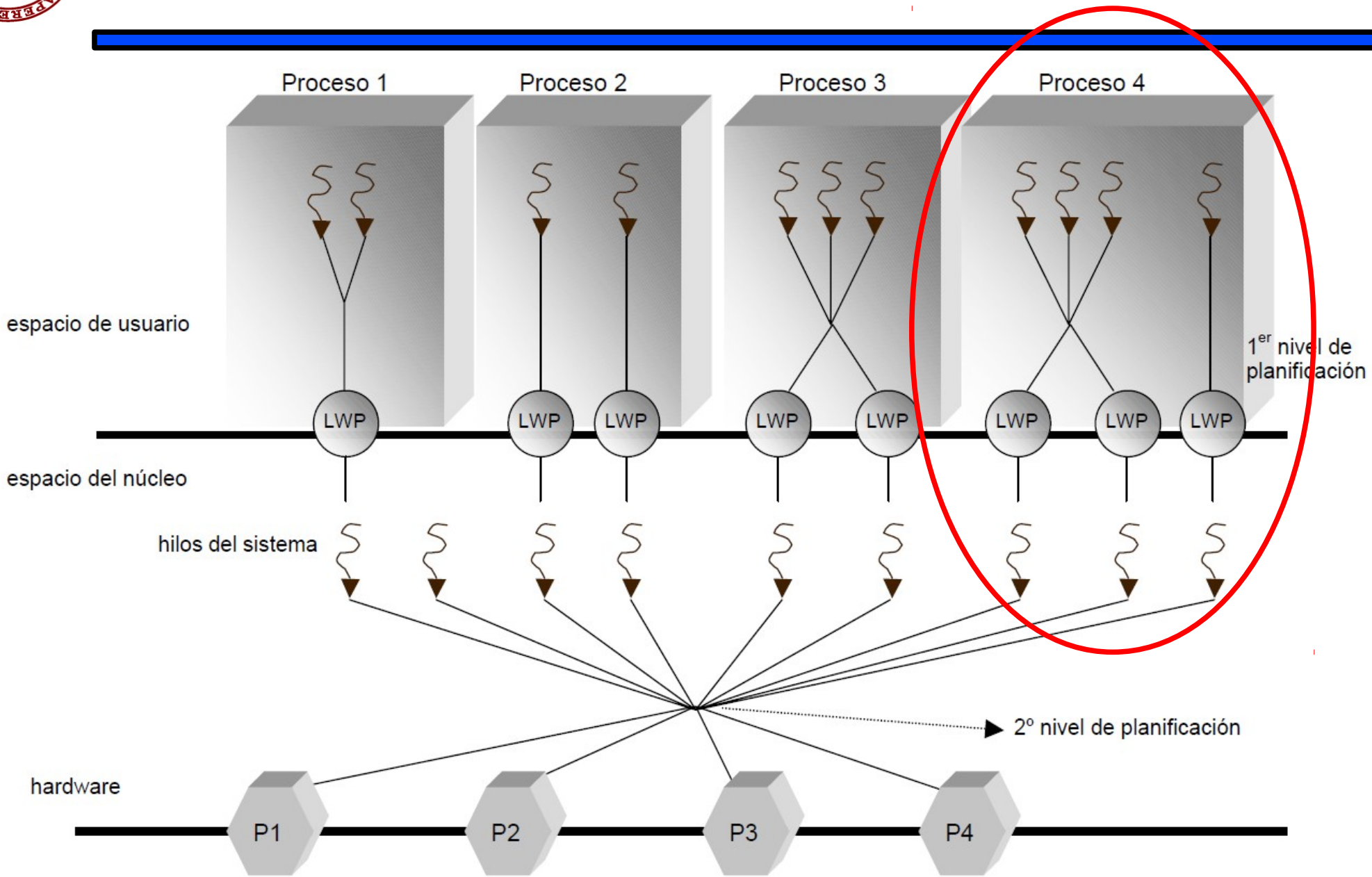
- Hay principalmente tres técnicas distintas para hacer la **planificación de hilos** :

  3. **El modelo de dos niveles**. Llamado modelo **muchos-a-muchos (no estricto)**. Es como el anterior pero ofrece la posibilidad de hacer un enlace de un hilo específico con un procesador lógico.

     Probablemente sea el mejor modelo. Varios sistemas operativos usan este modelo (Digital UNIX, Solaris, IRIX, HP-UX, AIX).

     En el caso de Java, las MVJ sobre estos SSOO tienen la opción de usar cualquier combinación de hilos enlazadas o no. La elección del modelo de hilos es una decisión al nivel de implementación de los escritores de la MVJ. Java en sí mismo no tiene el concepto de procesador lógico.

# 5. Procesos VS Hilos

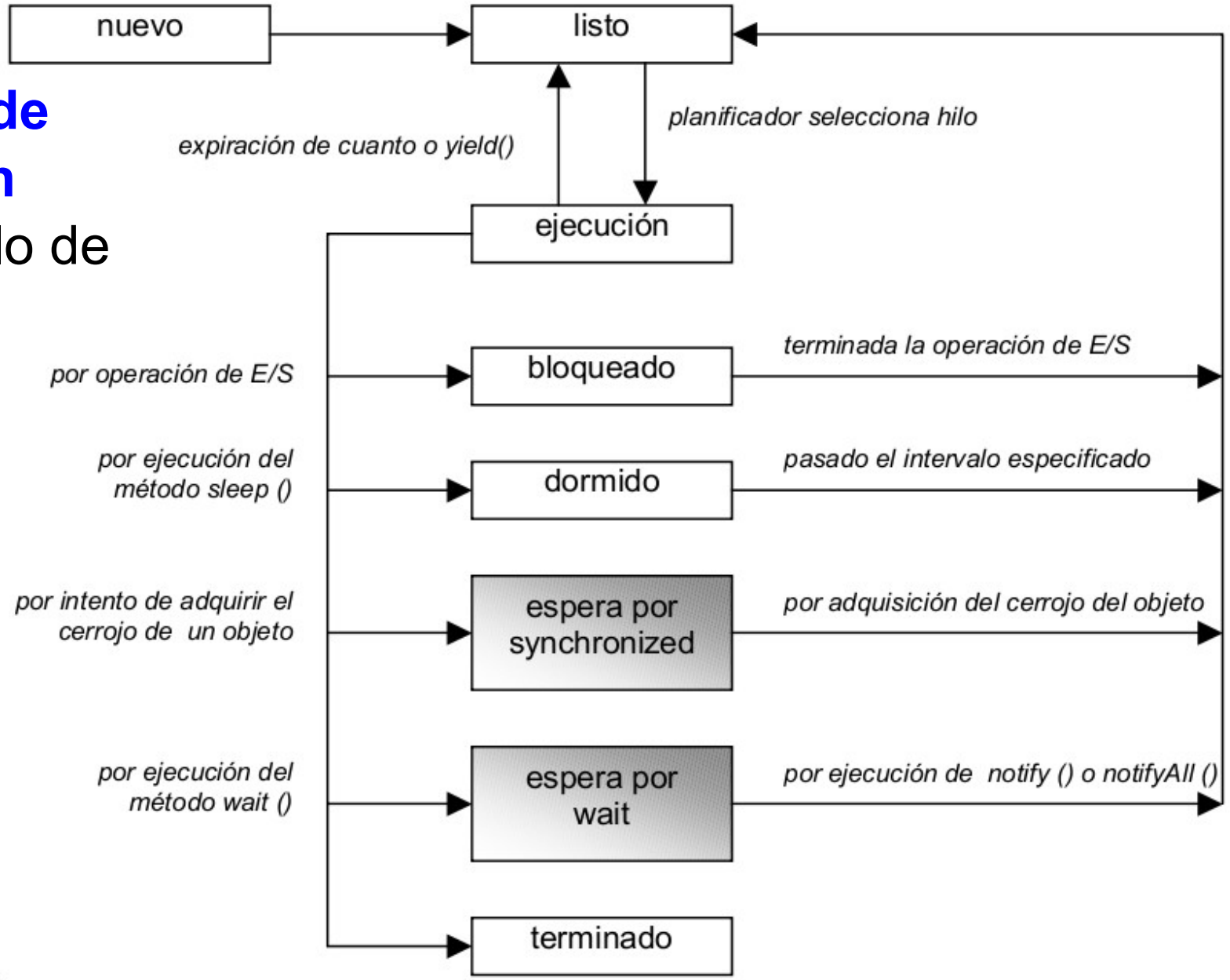


# 5. Procesos VS Hilos

- **Java** fue diseñado para obtener una gran portabilidad. Este objetivo hipoteca el modelo de hilos a usar en Java pues se deseaba que fuera fácil de soportar en cualquier plataforma y, como se ha visto, cada plataforma hace las cosas de una forma diferente. Esto hace que el **modelo de hilos** sea demasiado generalista. Sus principales **características** son:
  - Todos los hilos de Java tienen una **prioridad** y se supone que el planificador dará preferencia a aquel hilo que tenga una prioridad más alta. Sin embargo, no hay ningún tipo de garantía de que en un momento determinado el hilo de mayor prioridad se esté ejecutando.
  - Las rodajas de tiempo pueden ser aplicadas o no. Dependerá de la gestión de hilos que haga la librería sobre el que se implementa la máquina virtual java.

# 5. Procesos VS Hilos

- **Estados de un hilo en Java** (Ciclo de Vida)



# 6. Especificaciones de ejecución concurrente

- ¿Qué se puede ejecutar concurrentemente?

```
x = x + 1;
y = x + 2;
```

```
x = 1;
y = 1;
z = 1;
```

- A. J. Bernstein (1966) definió unas condiciones para determinar si dos conjuntos de instrucciones $S_i$ y $S_j$ se pueden ejecutar concurrentemente.

# 6. Especificaciones de ejecución concurrente

- Condiciones de Berstein:
  - *Sea* $L(S_k) = \{a_1, a_2, ..., a_n\}$ *el* ***conjunto de lectura*** *del conjunto de instrucciones* $S_k$*, formado por las variables que son leidas durante la ejecución del conjunto de instrucciones* $S_k$.
  - *Sea* $E(S_k) = \{b_1, b_2, ..., b_n\}$ *el* ***conjunto de escritura*** *del conjunto de instrucciones* $S_k$*, formado por las variables que son escritas durante la ejecución del conjunto de instrucciones* $S_k$.

# 6. Especificaciones de ejecución concurrente

- Para que dos conjuntos de instrucciones $S_i$ y $S_j$ se puedan ejecutar concurrentemente debe cumplirse:

1. $L(S_i) \cap E(S_j) = \emptyset$
2. $E(S_i) \cap L(S_j) = \emptyset$
3. $E(S_i) \cap E(S_j) = \emptyset$

# 6. Especificaciones de ejecución concurrente

- Ejemplo:

$$S1 \rightarrow a = x + y; \quad L(S_1) = \{x, y\} \quad E(S_1) = \{a\}$$
$$S2 \rightarrow b = z - 1; \quad L(S_2) = \{z\} \quad E(S_2) = \{b\}$$
$$S3 \rightarrow c = a - b; \quad L(S_3) = \{a, b\} \quad E(S_3) = \{c\}$$
$$S4 \rightarrow w = c + 1; \quad L(S_4) = \{c\} \quad E(S_4) = \{w\}$$

- Comprobamos las condiciones por pares :

*Entre S1 y S2*

1. $L(S_1) \cap E(S_2) = \emptyset$
2. $E(S_1) \cap L(S_2) = \emptyset$
3. $E(S_1) \cap E(S_2) = \emptyset$

*Entre S1 y S3*

1. $L(S_1) \cap E(S_3) = \emptyset$
2. $E(S_1) \cap L(S_3) = a \neq \emptyset$
3. $E(S_1) \cap E(S_3) = \emptyset$

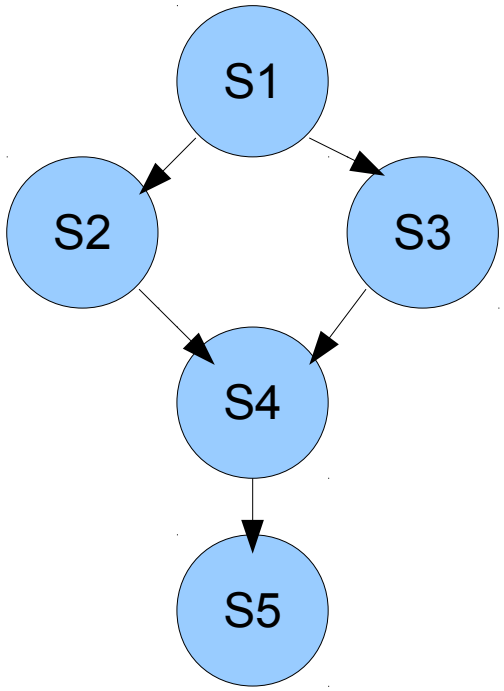
# 6. Especificaciones de ejecución concurrente

- Podemos recoger en una tabla las sentencias que pueden ejecutarse de forma concurrente:

| | S1 | S2 | S3 | S4 |
|---|---|---|---|---|
| S1 | - | SI | NO | SI |
| S2 | - | - | NO | SI |
| S3 | - | - | - | NO |
| S4 | - | - | - | - |

# 6. Especificaciones de ejecución concurrente

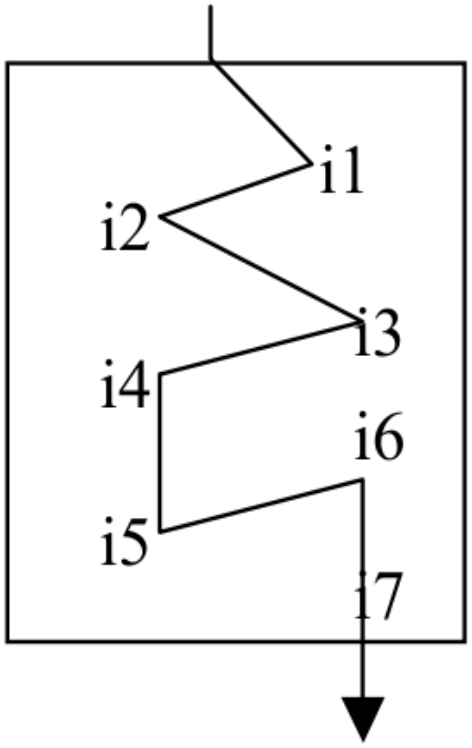
- **Grafos de Precedencia**. Es un grafo acíclico dirigido. Cada nodo representa un conjunto de instrucciones. Una flecha de A hacia B representa que B debe ejecutarse después de A. Dos flechas paralelas equivalen a ejecución concurrente.




```
begin
 S1
 cobegin
   S2
   S3
 coend
 S4
 S5
end
```

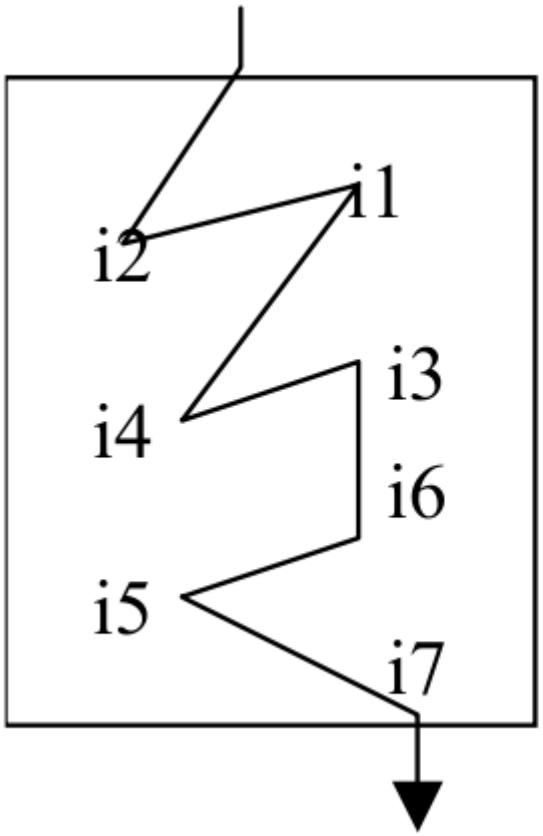
# 7. Características de los sistemas concurrentes

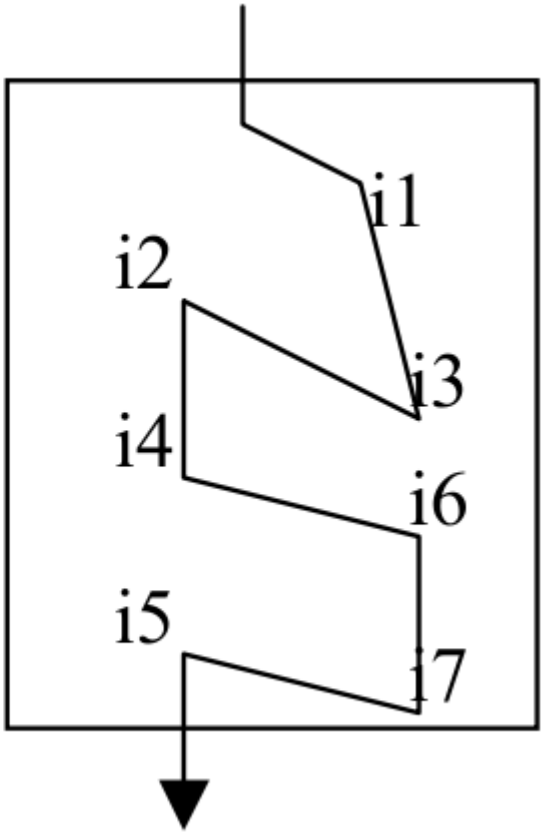
- **Orden de ejecución de las instrucciones**:
  - **Orden total**. Ocurre en los programas secuenciales. Dado un conjunto de datos de entrada, se conoce la ejecución del programa (flujo de programa).



# 7. Características de los sistemas concurrentes

- **Orden de ejecución de las instrucciones**:
  - **Orden parcial**. Ante el mismo conjunto de datos de entrada no se puede saber cual será el flujo de ejecución.



# 7. Características de los sistemas concurrentes

- **Indeterminismo**. Debido al orden parcial, los programas pueden diferentes resultados al ejecutarse sobre el mismo conjunto de datos de entrada.

```
program incognita
  var x: integer;

process P1;
  var i: integer;
begin
  for i:=1 to 5 fo x:=x+1;
end;
process P2;
  var j: integer;
begin
  for j:=1 to 5 fo x:=x+1;
end;
```

```
begin
x:=0;
cobegin
  P1;
  P2;
coend;
end.
```

El valor final de x puede ser 5,6,7,8,9,10.

# 8. Problemas inherentes a la programación concurrente

- **Exclusión mutua**. La ejecución concurrente se produce entre las instrucciones de bajo nivel, las generadas por el compilador.
  - La instrucción `x = x + 1;` se transforma en:

```
LOAD X R1
ADD R1 1
STORE R1 X
```

  - Cada una de ellas es indivisible, dado que se ejecuta en un ciclo de reloj, pero cualquier intercalación entre ellas es posible.

# 8. Problemas inherentes a la programación concurrente

- **Exclusión mutua**.
  - Si dos procesos ejecutan concurrentemente esta instrucción
    `x = x + 1;`

```
P1
LOAD X R1
ADD R1 1
STORE R1 X
```

```
P2
LOAD X R1
ADD R1 1
STORE R1 X
```

  - Suponiendo que inicialmente x vale 0, puede devolver como resultado que x valga 1 o 2.
  - Esto, además viene determinado por las condiciones de Berstein.

# 8. Problemas inherentes a la programación concurrente

- **Exclusión mutua**.
  - Si dos procesos ejecutan concurrentemente esta instrucción
    `x = x + 1;`

```
P1
(1) LOAD X R1
(2) ADD R1 1
(3) STORE R1 X
```

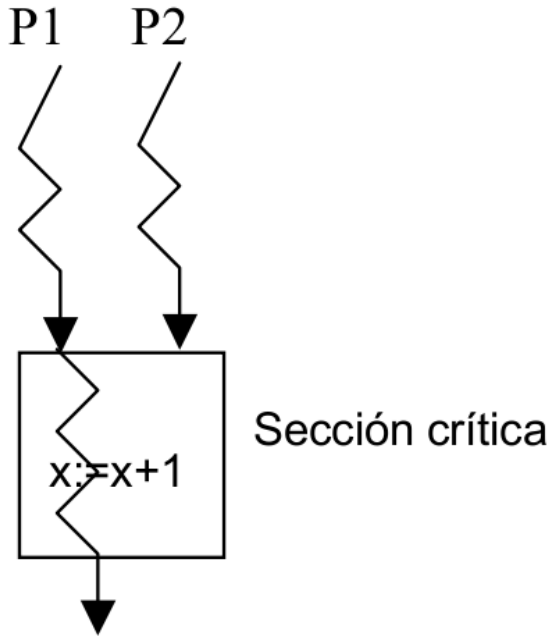
```
P2
(1) LOAD X R1
(2) ADD R1 1
(3) STORE R1 X
```

| x | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| P1 | 1 | 2 | | | 3 | | |
| P2 | | | 1 | 2 | | 3 | |

Tiempo

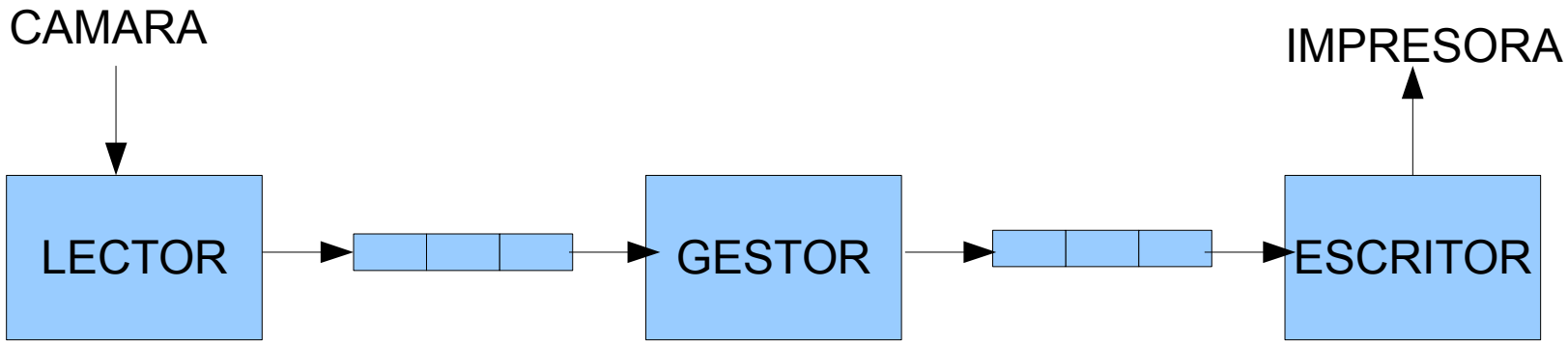
# 8. Problemas inherentes a la programación concurrente

- **Exclusión mutua**.
  - La sección de código que no puede ejecutarse de forma concurrente es conocida como **Sección Crítica**.
  - Hay que asegurarse de que las secciones críticas se ejecutan en **exclusión mutua**, es decir, que un solo proceso puede ejecutar dicha sección crítica a la vez.
  - La programación concurrente debe proporcionar mecanismos para especificar que partes del código deben ejecutarse bajo exclusión mutua.

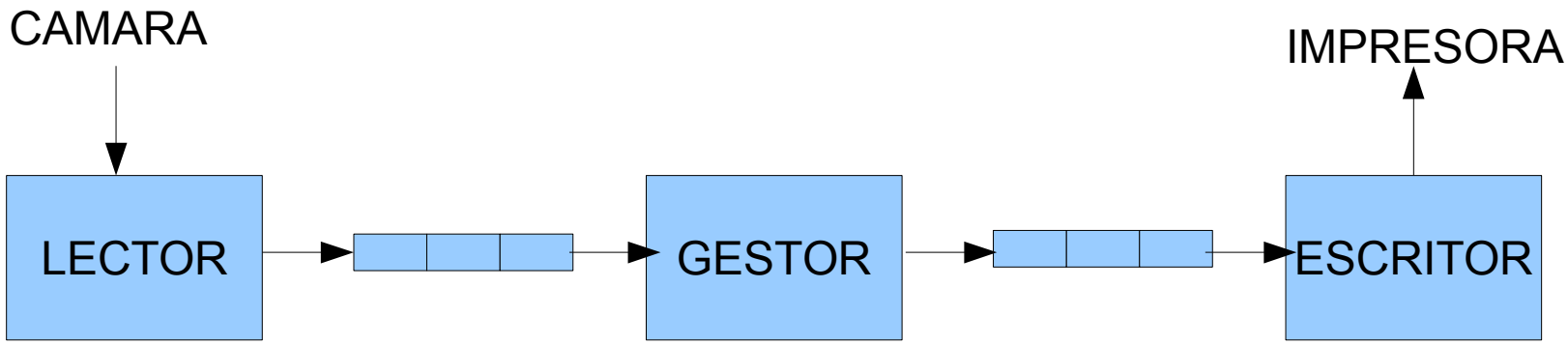


# 8. Problemas inherentes a la programación concurrente

- **Condición de sincronización**.
  - Hay ocasiones en que los procesos no pueden hacer uso de un recurso compartido hasta que éste no se encuentre en un determinado estado.
  - La programación concurrente debe proporcionar mecanismos para bloquear procesos que no pueden hacer algo en espera de un evento que los desbloquee.



# 8. Problemas inherentes a la programación concurrente




```
process LECTOR;
begin
 repeat
  capturar imagen;
  almacena en buffer;
 forever
end;
```

```
process GESTOR;
begin
 repeat
  saca imagen buffer;
  trata imagen;
  almacena en cola;
 forever
end;
```

```
process LECTOR;
begin
 repeat
  lee imagen cola;
  imprime imagen;
 forever
end;
```

- ¿Qué ocurre cuando se trata de escribir en las colas y no hay sitio?
- ¿Qué ocurre cuando se trata de leer de las colas y están vacías?

# 9. Corrección de programas concurrentes

- El **orden parcial** e **indeterminismo** hace que la **corrección** de un programa concurrente sea más difícil de conseguir que la de un programa secuencial.
- Para que un programa concurrente sea **correcto**, además de cumplir las especificaciones funcionales que deba cumplir, debe satisfacer una serie de **propiedades inherentes a la concurrencia**. Podemos agrupar esas propiedades en:
  - **Propiedades de seguridad**: son aquellas que aseguran que nada malo va a pasar durante la ejecución del programa.
  - **Propiedades de viveza**: son aquellas que aseguran que algo bueno pasará eventualmente durante la ejecución del programa.

# 9. Corrección de programas concurrentes

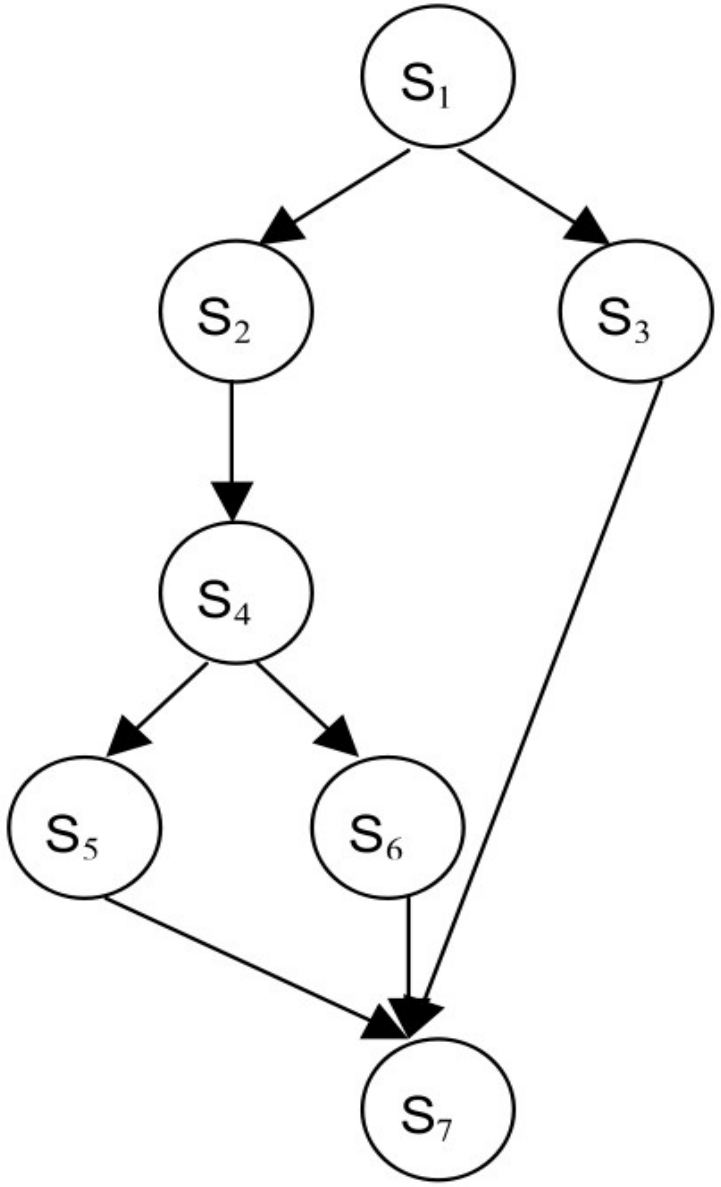
- Propiedades de **seguridad**.
  - **Exclusión mutua**. Hay recursos en el sistema que deben ser accedidos en exclusión mutua . Cuando esto ocurre,hay que garantizar que si un proceso adquiere el recurso, otros procesos deberán **esperar a que sea liberado**. De lo contrario, el resultado puede ser imprevisto.
  - **Condición de sincronización**. Hay situaciones en las que un proceso debe **esperar por la ocurrencia de un evento** para poder seguir ejecutándose. Cuando esto ocurre, hay que garantizar que el proceso no prosigue hasta que no se produce el evento. De lo contrario, el resultado puede ser imprevisto.

# 9. Corrección de programas concurrentes

- Propiedades de **seguridad**.
  - **Interbloqueo** (deadlock, bloqueo mutuo). Se produce cuando todos los procesos están esperando porque ocurra un evento que nunca se producirá.
- Propiedades de **vivacidad**.
  - **Interbloqueo activo** (livelock). Se produce cuando un sistema ejecuta una serie de instrucciones sin hacer ningún progreso. Su detección es muy complicada.
  - **Inanición** (starvation). Se produce una situación de este tipo cuando el sistema en su conjunto hace progresos, pero existe un grupo de procesos que nunca progresan pues no se les otorga tiempo de procesador para avanzar.

# Ejercicios

- Construir un programa concurrente que se corresponda con el grafo de precedencia de la siguiente figura utilizando el par cobegin/coend



# Ejercicios

- Dado el siguiente trozo de código obtener su grafo de precedencia correspondiente.

```
s0;
cobegin
  s1;
  begin
    s2;
    cobegin
      s3;s4
    coend;
    s5
  end;
  s6
coend;
s7
```

# Ejercicios

- Usando las condiciones de Bernstein, construir el grafo de precedencia del siguiente trozo de código y el programa concurrente correspondiente usando el par cobegin/coend.

```
S1: cuad := x*x;

S2: m1 := a*cuad;

S3: m2 := b*x;

S4: z := m1 + m2;

S5: y := z + c;
```

# Ejercicios

- Dado el siguiente conjunto de instrucciones, utilice las condiciones de Bernstein para establecer el grafo de precedencias que le corresponde.

```
S1: a = c + b;
S2: d = c / e + a;
S3: c = x + 3;
S4: e = f * (d + 2);
S5: h = c * e;
S6: x = e / h;
S7: y = 2 * e / h;
```

# Ejercicios

- Solución