Arquitectura de Sistemas

Activación

Gustavo Romero López

Updated: 4 de abril de 2019

Arquitectura y Tecnología de Computadores

Índice

Lecturas recomendadas

J. Bacon	Operating Systems (4)
A. Silberschatz	Fundamentos de Sistemas Operativos (2)
W. Stallings	Sistemas Operativos (3, 4)
A. Tanuenbaum	Sistemas Operativos Modernos (2)

Motivación

- ¿Por qué introducir estados para las hebras si ya disponemos de estados para los procesos? $\longrightarrow$ eficiencia.
- Los estados de las hebras, ¿son necesarios o simplemente mejoran el control de las hebras?
- Si son necesarios, ¿cuáles utilizar?
- Por simplicidad, en este tema nos centraremos en el las hebras tipo núcleo.

Ventajas del uso de estados en las hebras

- A veces es necesario acceder a una hebra que espera por un evento pero no tenemos su identificador (TID)... ¿Cómo localizarla?
- Es más rápido buscar si están agrupadas por estados $\longrightarrow$ mejora el rendimiento del SO.
- **Imprescindibles** si **SMP** + **cola central** de hebras preparadas + política de planificación global.
  - *SO diferente para máquinas monoprocesador y multiprocesador.*

TCB de una hebra núcleo en un SMP

identificador (TID)
estado
puntero de instrucción (IP)
puntero de pila (SP)
banderas
afinidad

- El estado de la hebra como mínimo debe distinguir entre las que se están ejecutando y las que no.
- Una misma hebra **no** debería ejecutarse **en paralelo** en más de un procesador.
- Es útil añadir un campo para indicar cual fue el último procesador en el que se ejecutó (**afinidad**) porque mejora el rendimiento.

Estados de una hebra (1)

- La noción de *"estado de una hebra"* puede resultar un poco confusa porque una hebra en ejecución cambia su **estado interno** de ejecución con cada instrucción.
  - Esta clase de estado pertenece al contexto de una hebra (recordad el cambio de hebra).
- La noción de *"estado de una hebra"* que manejaremos aquí hace referencia al **estado externo** de una hebra en cuanto a su relación con el entorno:
  - uso de recursos
  - otras hebras
  - el sistema operativo

Estados de una hebra (2)

Definición: El estado de una hebra expresa la relación de una hebra con respecto a otras y al sistema.

- Una hebra "ejecutando" se está ejecutando sobre un procesador.
- Una hebra "preparada" es ejecutable, sólo está esperando a que quede un procesador libre.
- Una hebra "bloqueada" espera un evento...
  - el fin de una operación de E/S.
  - la llegada de un mensaje.
  - la llegada de una señal.
  - la liberación de un recurso ocupado.

Modelo de un estado

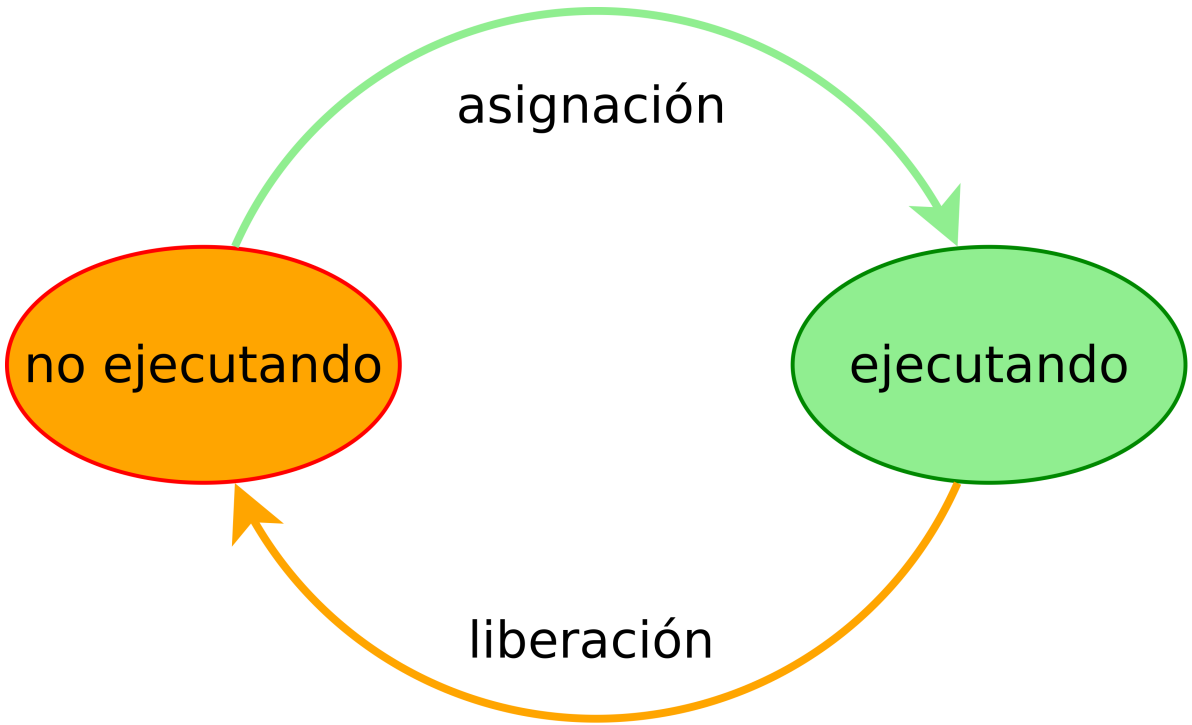
- No existe control explícito sobre las hebras: todas las hebras están activas pero sólo algunas de ellas progresan en su ejecución.
  - debido a la falta de procesadores
  - debido a la espera por ciertos eventos

Consecuencias del modelo de un estado

- Planificación:
  - Sólo los más simples son posibles o nos arriesgamos a una gran sobrecarga.
  - Ineficiente debido a los cambios de hebra superfluos.
- Sólo válido en sistemas con un único procesador.

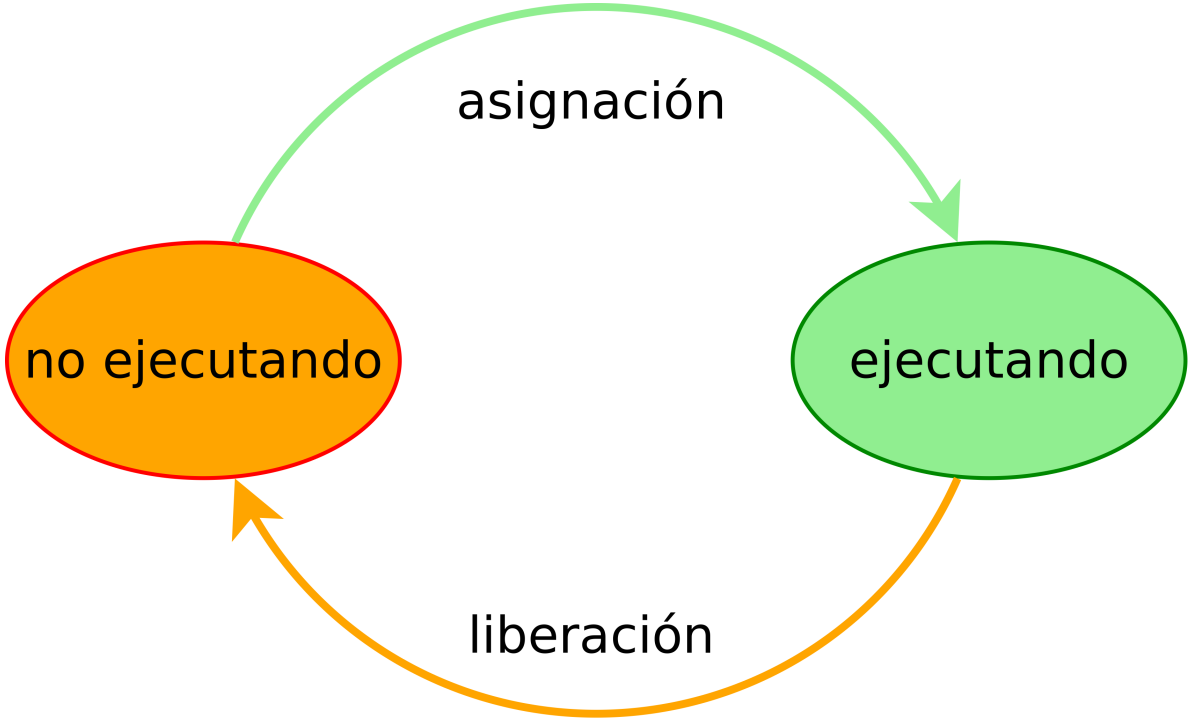
Modelo de dos estado

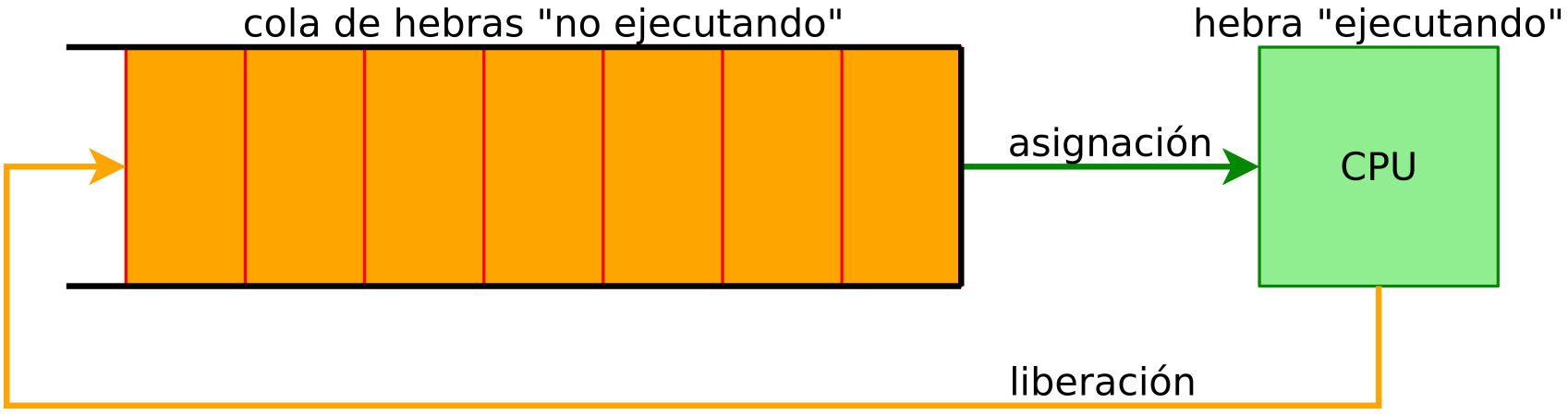
- Modelo muy primitivo.
- Cada hebra está en uno de los estados:
  - ejecutando
  - no ejecutando



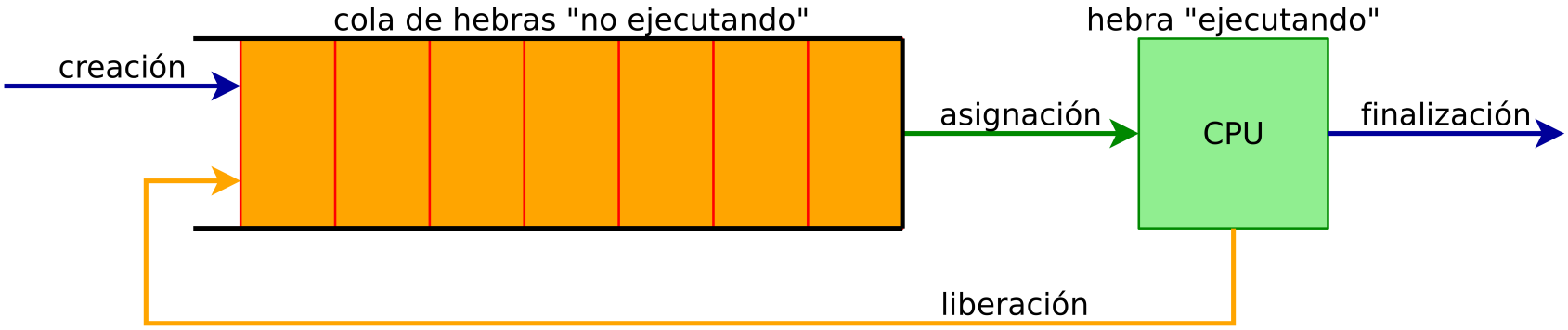
Modelo de dos estados

- ¿Cómo implementar este modelo?



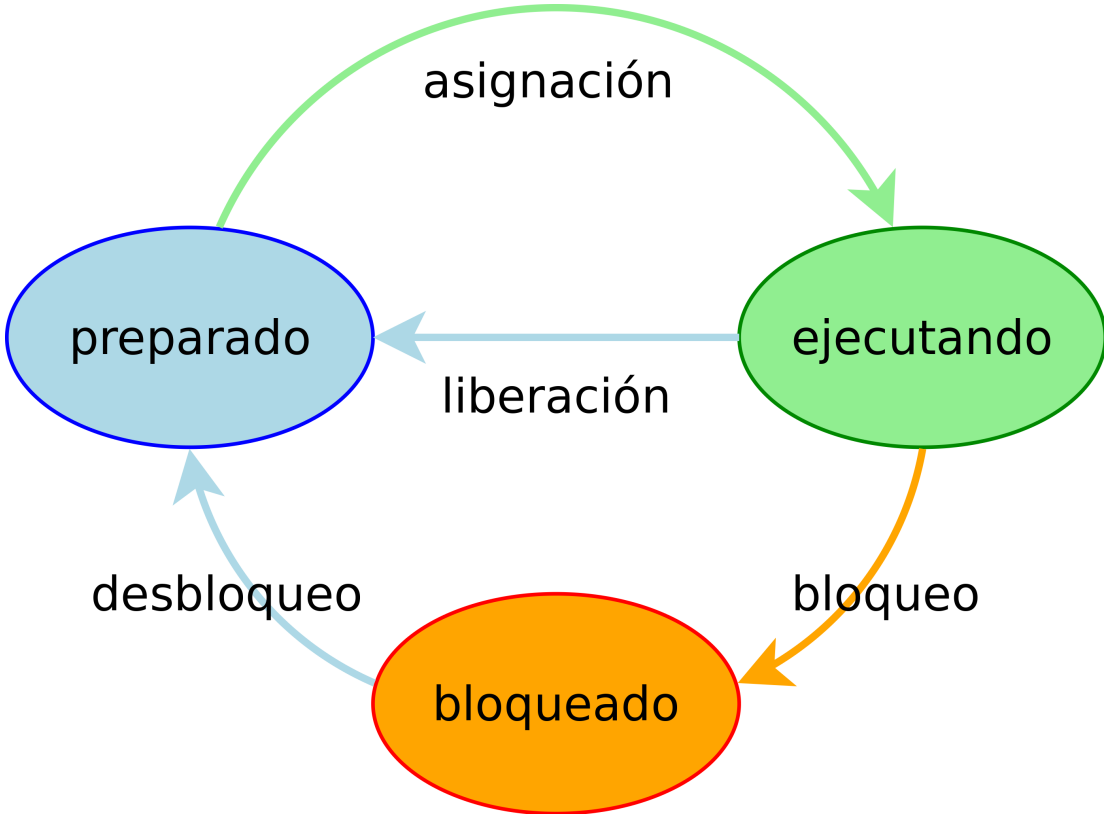


- Este modelo representa un sistema estático.



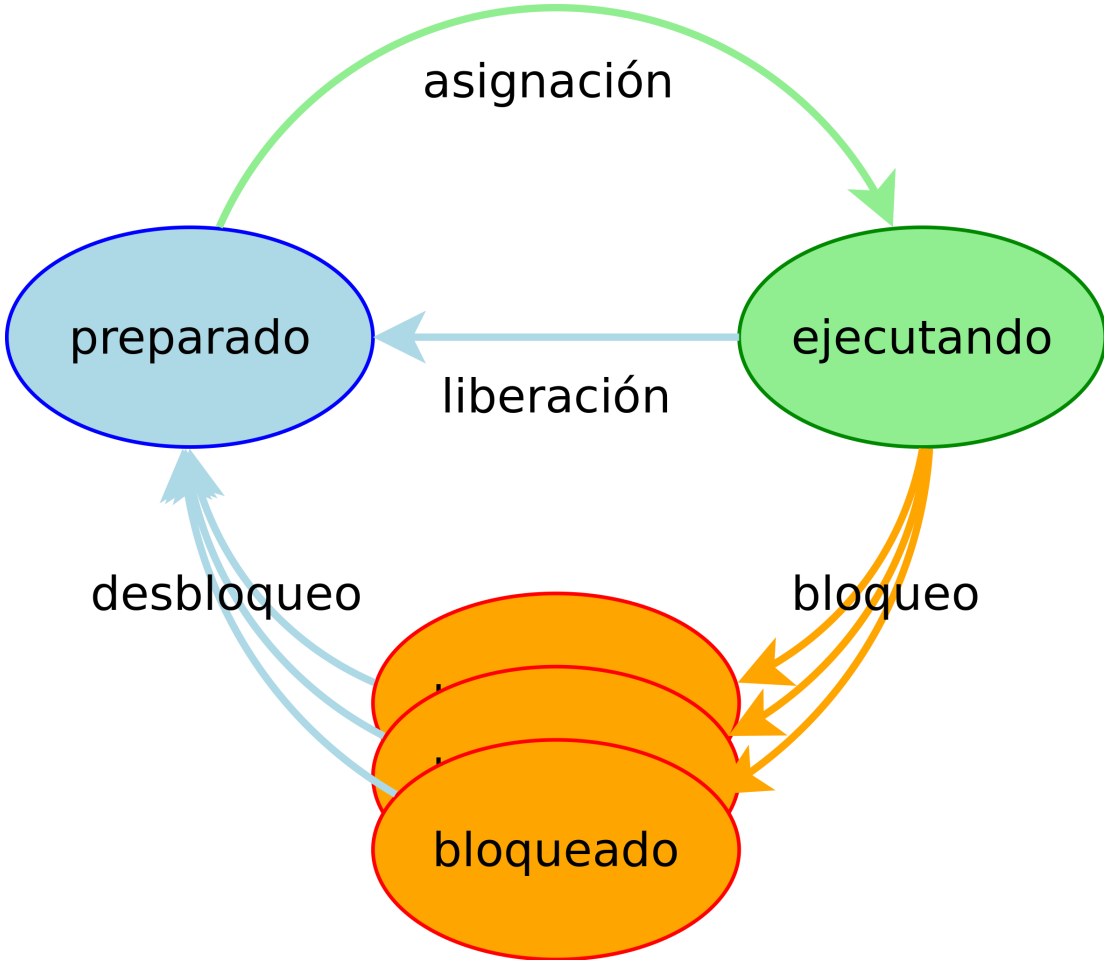
- Estos modelos son utilizado a menudo para analizar los efectos de la elección de los parámetros de diseño: velocidad del procesador, longitud de las colas y políticas de planificación.

Modelo de tres estados



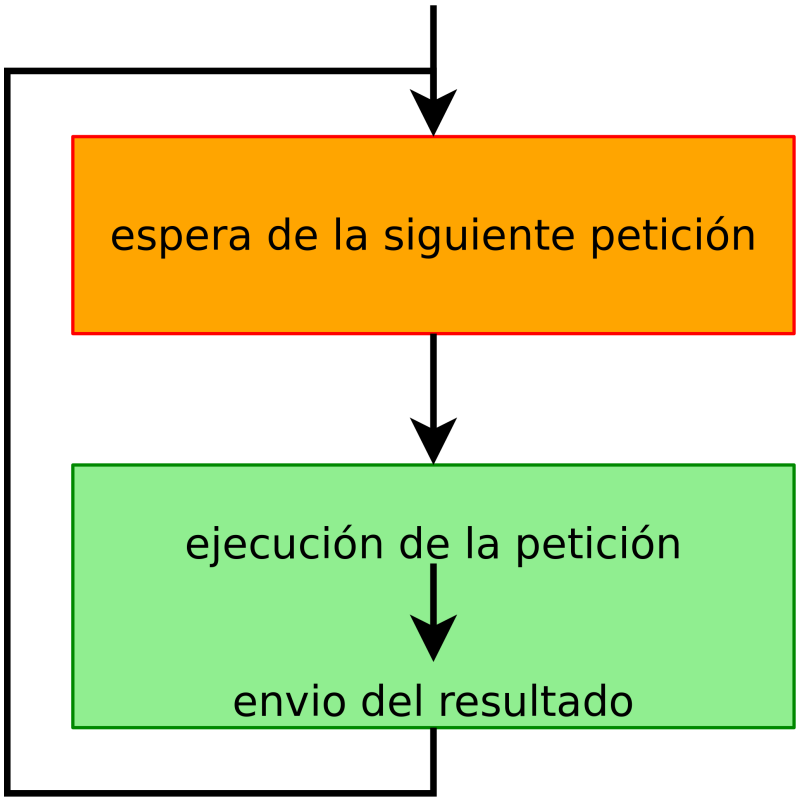
- Es cuestión de diseño y no de implementación si existe un único estado bloqueado para todos los tipos de eventos...

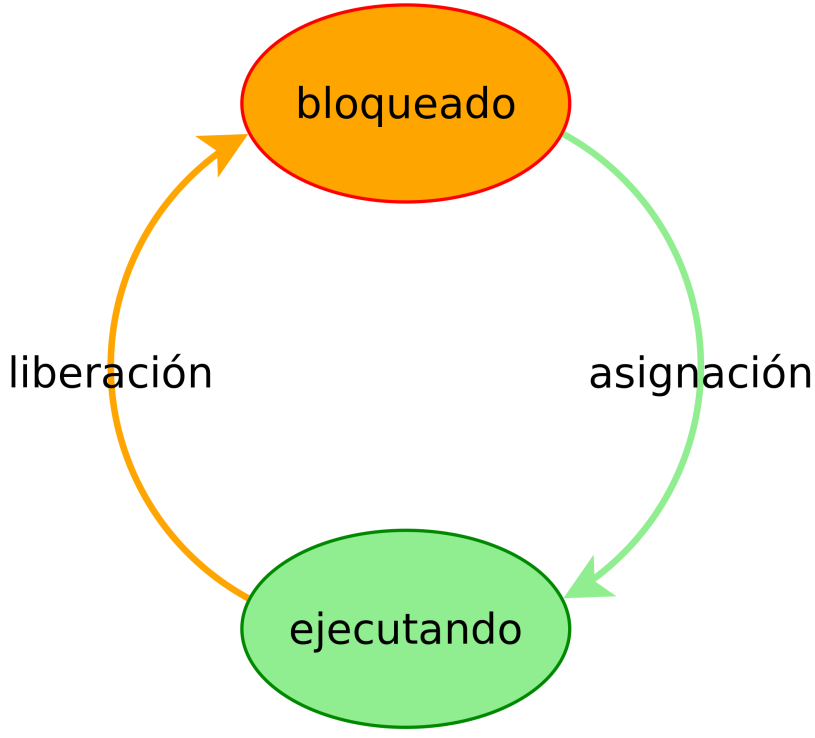
Modelo de tres estados



- ... o si existe un estado bloqueado diferente por cada tipo de evento.

Modelando las actividades de E/S como hebras





- ¿Por qué no se incluye el estado **preparado**? → no es necesario ni deseable.

nuevo

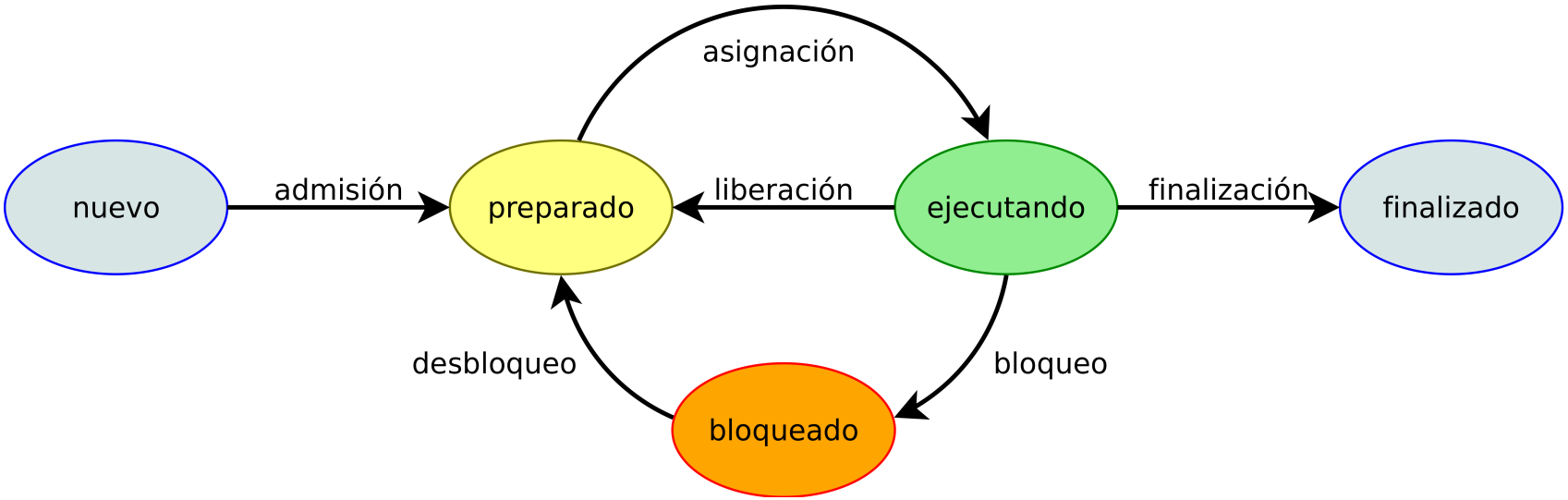
- Cuando el SO crea una hebra tipo núcleo...
  - crea un identificador de hebra único.
  - crea un TCB para gestionar la hebra.
  - crea un espacio de direcciones (tablas de páginas).
  - crea e inicializa otras estructuras de control de recursos.
- ... pero todavía no hace que la hebra se pueda ejecutar, no es **admitida**[1], debido a que...
  - los recursos son escasos.
  - existe alguna restricción de tiempo.

[1] algunos autores defienden que todo SO necesita control de admisión

finalizado

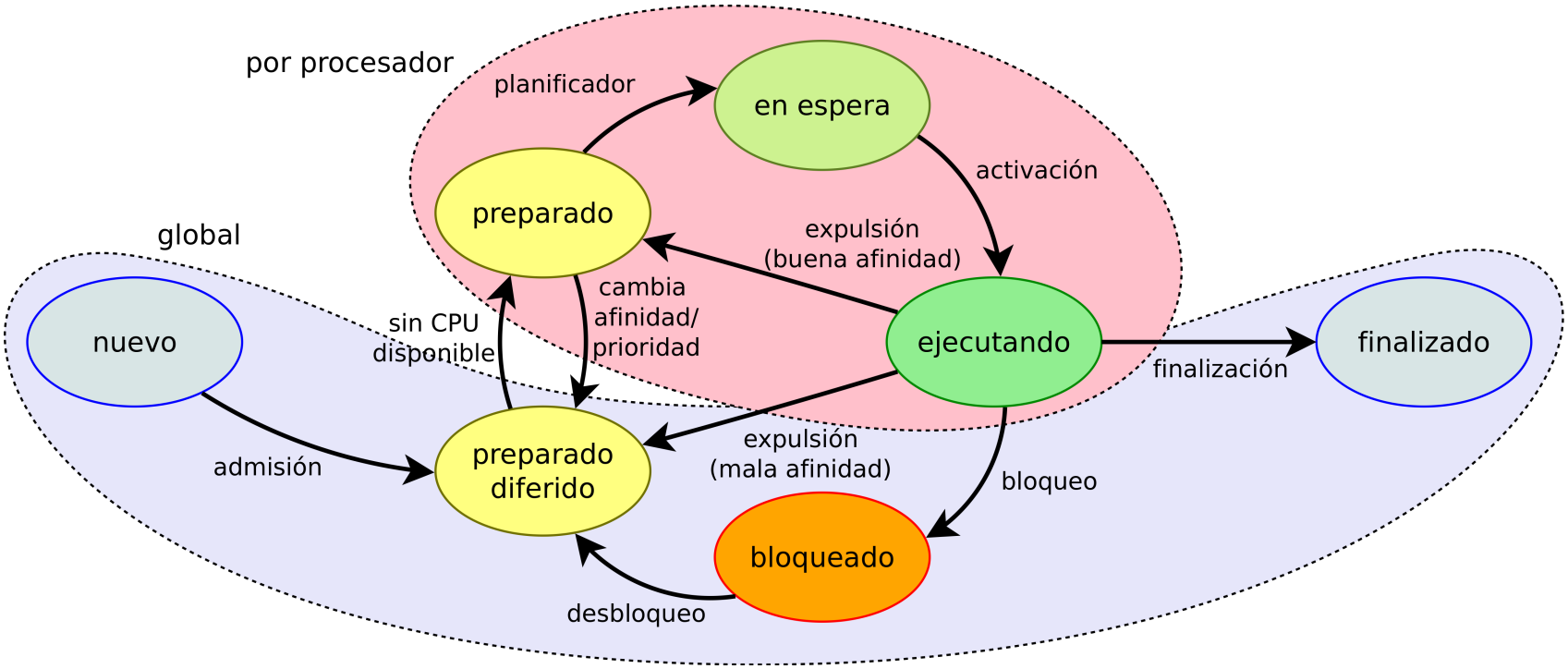
- La hebra deja de ser seleccionable para ejecución.
- El TCB y otras subtablas se guardan para ciertas tareas:
  - programas de contabilidad que acumulan el uso de recursos con el fin de facturar a clientes.
  - facilitar la creación y ejecución de hebras nuevas.
  - depuración (tras un fallo).
- El TCB puede ser **borrado** cuando ya no es necesario.
  - ¿Cuándo? ⟶ no necesito su **contenido** ni reutilizar la propia **estructura**.

Modelo de 5 estados



- Existen motivos para incluir estados adicionales, sin embargo hay que evitar modelos demasiado complejos.
- 1ª Regla de diseño: mantener simples las cosas simples.

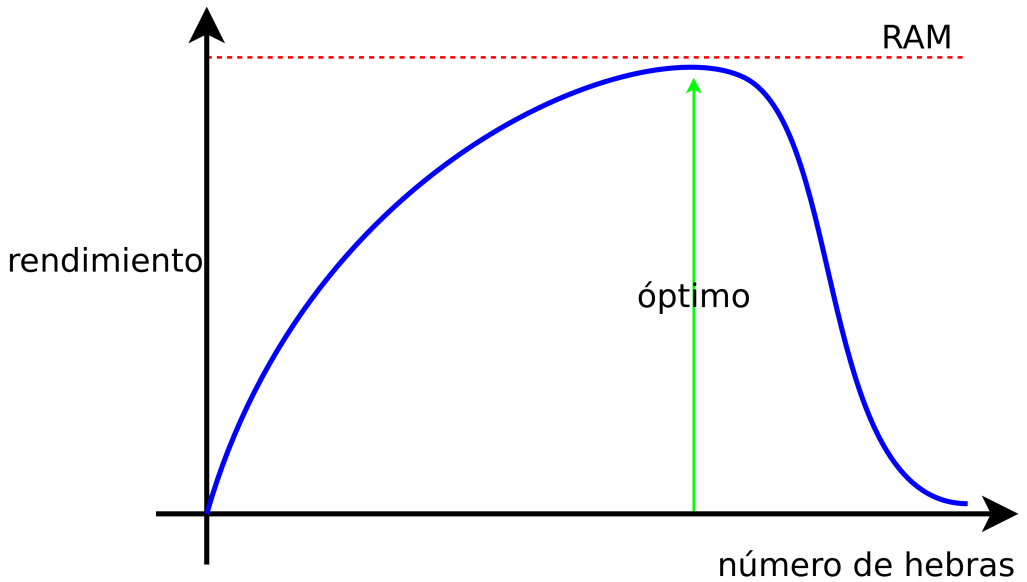
Modelo de 7 estados de Windows



- preparado = preparado + preparado diferido + en espera.
- Los nuevos estados no son imprescindibles.
- Razones para la subdivisión:
  - cambiar de hebra más fácil y rápidamente (escalable).
  - mejorar la gestión de afinidades y prioridades.

La necesidad del **intercambio** (“*swapping*”)

- En la mayoría de los SO las hebras residen **siempre en memoria principal** → no disponen del estado **suspendido**.
- Cuando demasiadas hebras son admitidas el rendimiento disminuye considerablemente debido al fenómeno conocido como **hiperpaginación** (“*thrashing*”).
- La hiperpaginación se produce cuando la sobrecarga del intercambio prima en el sistema sobre el trabajo útil.



Implementación del estado una hebra

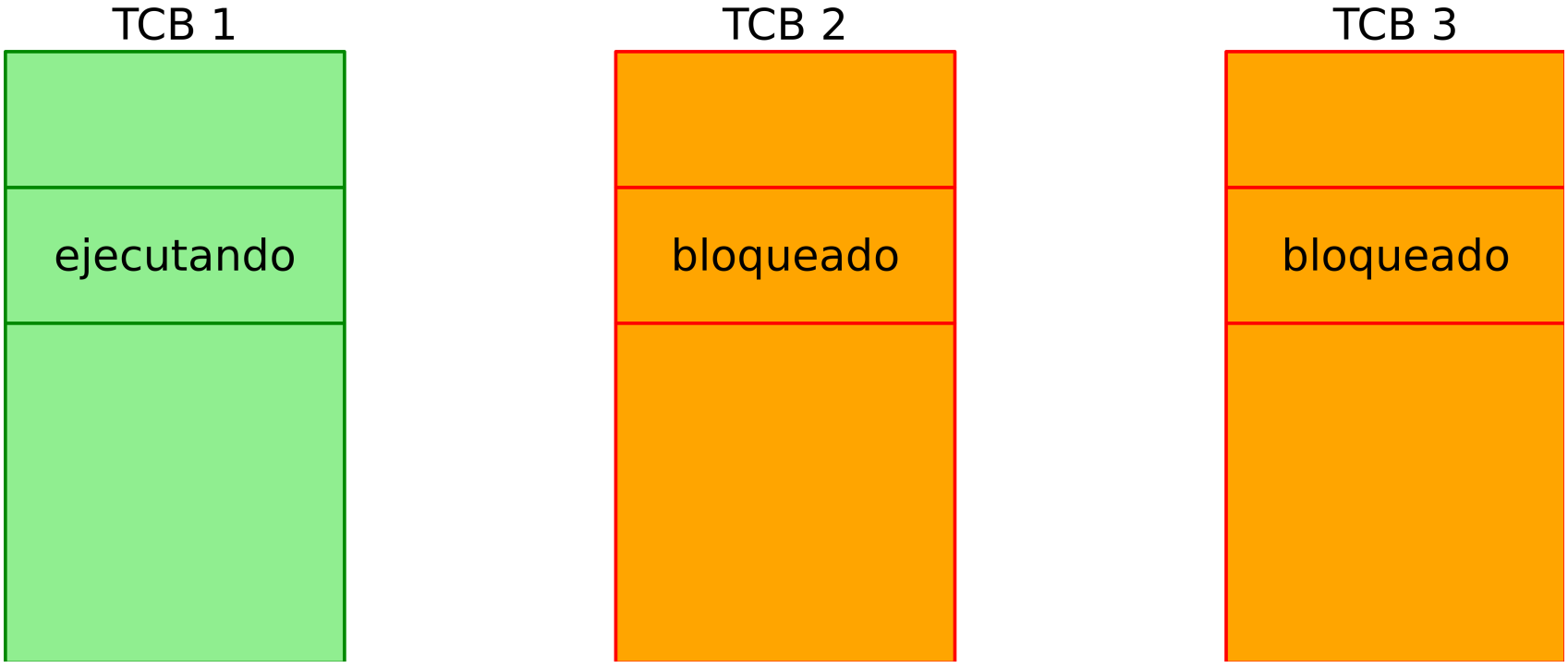
Método de implementación:

- Explícito: nuevo atributo en el TCB.
- Implícito: enlazando los TCB.
  - vector
  - lista
  - lista doblemente enlazada
  - árbol

En algunos sistemas se usan ambos... ¿Por qué?

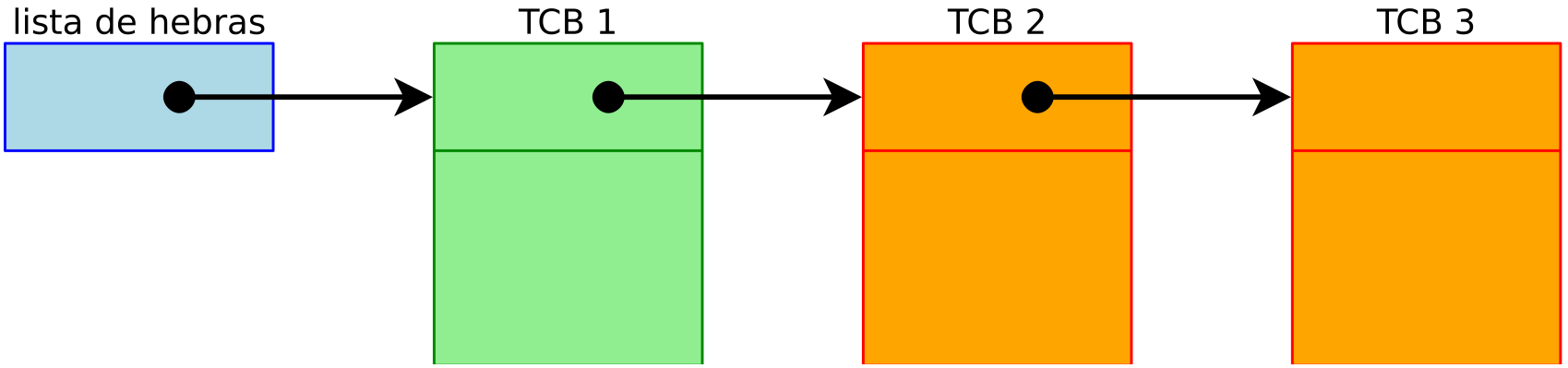
- por redundancia: mejora la robustez del sistema.
- por eficacia: transiciones perezosas, reducción de la sobrecarga.

Estado como atributo en el TCB



Hebra siguiente = búsqueda lineal $\Longrightarrow$ eficiencia: O(n).

Estado como estructura de datos



Hebra siguiente = primera hebra de la lista de preparadas $\Longrightarrow$ eficiencia: $O(1)$.

Análisis (1)

Implementación:

- 1001 hebras, todas ellas en una única lista.
- **Sin atributo** "estado de hebra" dentro del TCB.
- **Sin estructura de datos** específica para hebras ejecutables.
- Sólo la hebra actual es ejecutable, las demás esperan algún evento.

Sobrecarga de planificación justa:

- supongamos un coste de cambio de hebra de $1\mu s$.

Resultado:

Realizamos 1000 cambios de hebra en vano hasta que la única hebra ejecutable es planificada de nuevo para su ejecución.

$$\textbf{sobrecarga} = 1000\ \mu s = 1\text{ms}$$

Análisis (2)

Implementación:

- 1001 hebras, todas ellas en una única lista.
- **Atributo** "estado de hebra" dentro del TCB.
- **Sin estructura de datos** específica para hebras ejecutables.
- Sólo la hebra actual es ejecutable, las demás esperan algún evento.

Sobrecarga de planificación justa:

- el coste del cambio de hebra es $1\mu s$
- el coste de comparar 2 entradas de la lista es $0.1\mu s$

Resultado:

Realizamos 1000 comparaciones hasta dar con la única hebra ejecutable

$$\textbf{sobrecarga} = 101\mu s = 0{,}101\textbf{ms}$$

Análisis (3)

Implementación:

- 1001 hebras repartidas en varias listas según su estado
- **listas** específicas para hebras ejecutables y no ejecutables
- Sólo la hebra actual es ejecutable, las demás esperan por eventos

Sobrecarga de planificación justa:

- el coste del cambio de hebra es $1\mu s$.
- el coste de comparar 2 entradas de la lista es $0.1\mu s$.

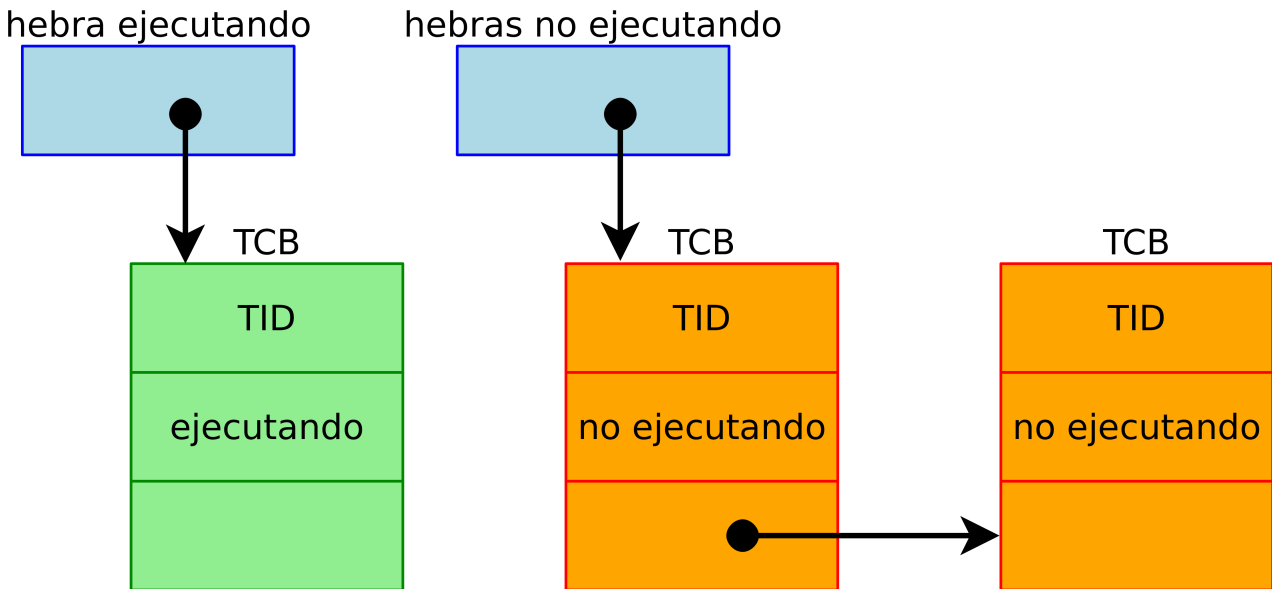
Resultado:

1 búsqueda en la cabeza de la lista nos indica que no hay otra hebra preparada $\Longrightarrow$ no realizamos ningún cambio de hebra

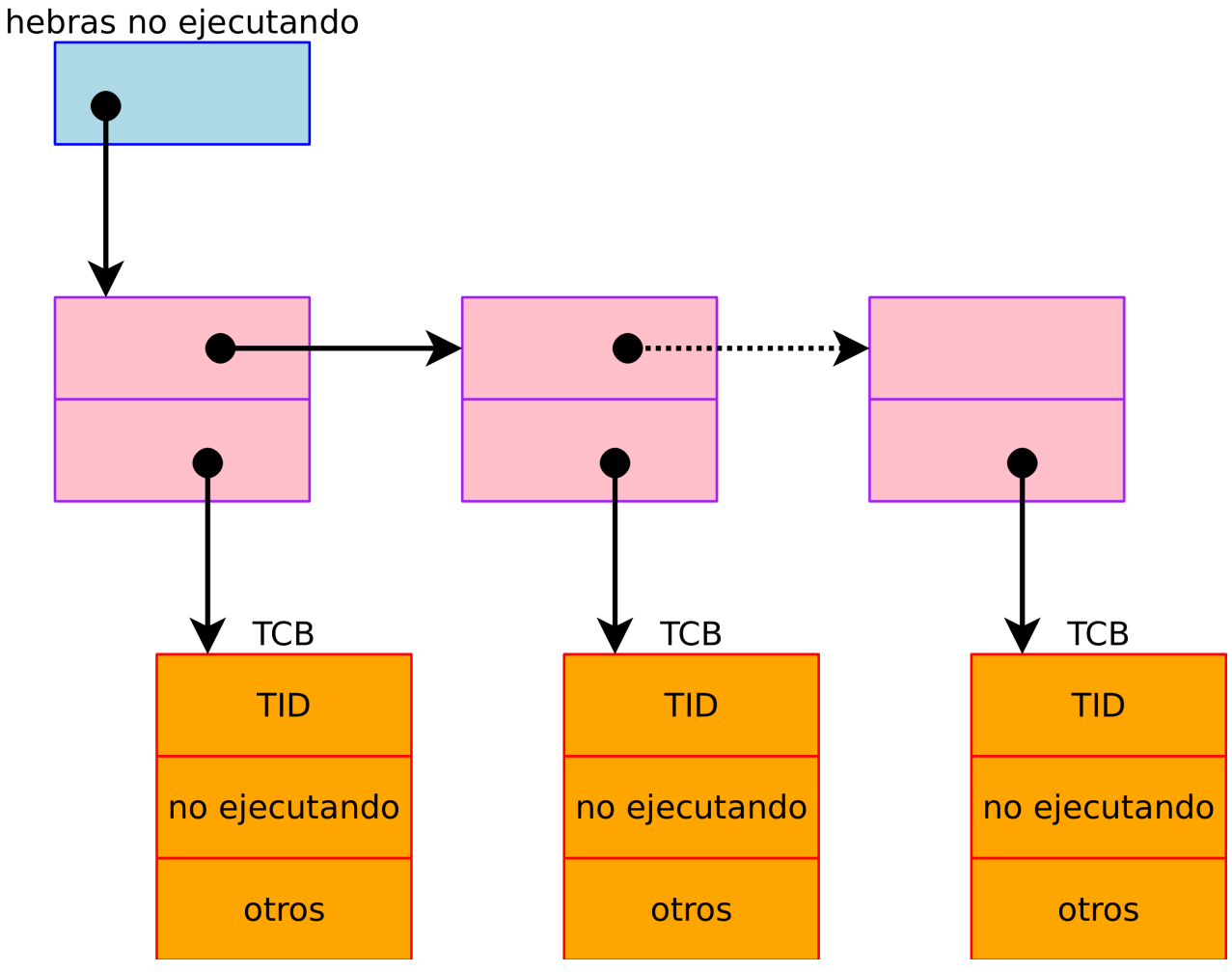
$$\textbf{sobrecarga} = 0.1\ \mu s = 100\textbf{ns}$$

Punteros dentro del TCB

- Para el modelo de 2 estados son necesarias:
  - 2 estructuras de datos $\longrightarrow$ una lista enlazada no suele ser una buena elección.
  - 2 apuntadores a dichas estructuras de datos.
- Estado ejecutando: un apuntador por procesador.
- Estado no ejecutando: una única cola global/por procesador.



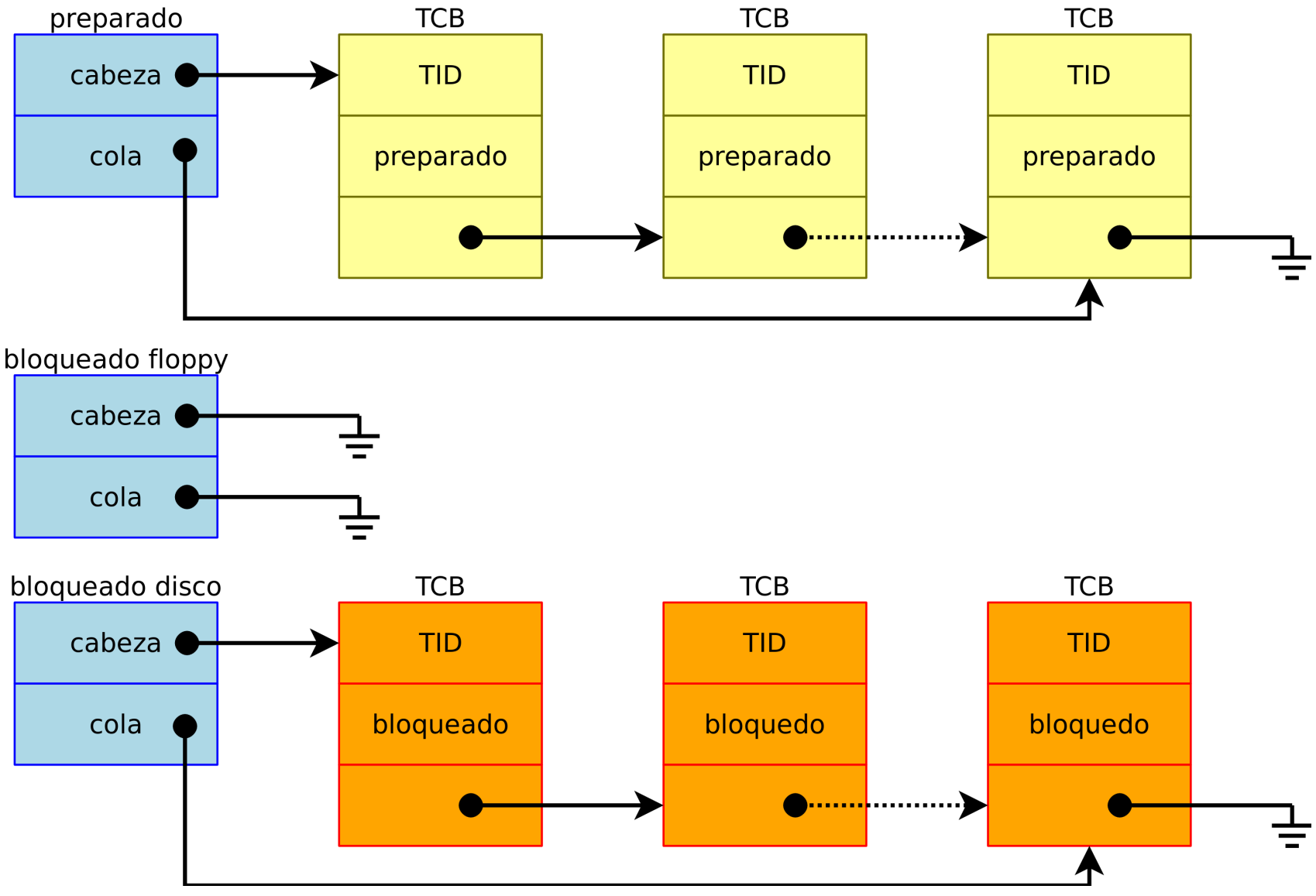
Punteros fuera del TCB ¿de qué quieres llenar tu caché?



En resumen...

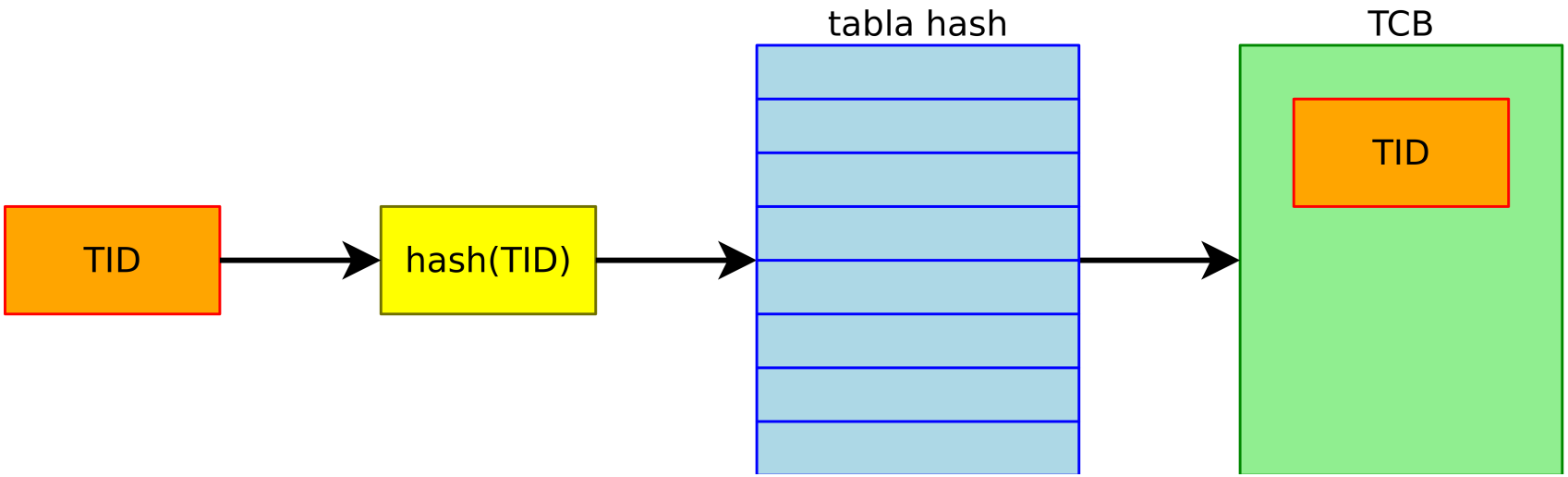
- No escoger estructuras de datos adecuadas para operaciones muy frecuentes $\longrightarrow$ pobre rendimiento.
- Lo que puede ser bueno para un pequeño número de hebras puede ser una misión imposible para un número mayor debido a la falta de escalabilidad.
  - ejemplo: algoritmos de búsqueda.
- Si insertar/borrar en cualquier posición de las listas es necesario, una lista enlazada será una mala elección.
- Un buen diseñador de sistemas considerará no sólo los requisitos actuales sino también los futuros.

Implementación algo más realista



Del TID al TCB

- Algunas llamadas al sistema necesitan el TID como parámetro.
- ¿Cómo encontramos el TCB de una hebra a partir de su TID?



Relación entre estados de procesos y hebras tipo **núcleo**

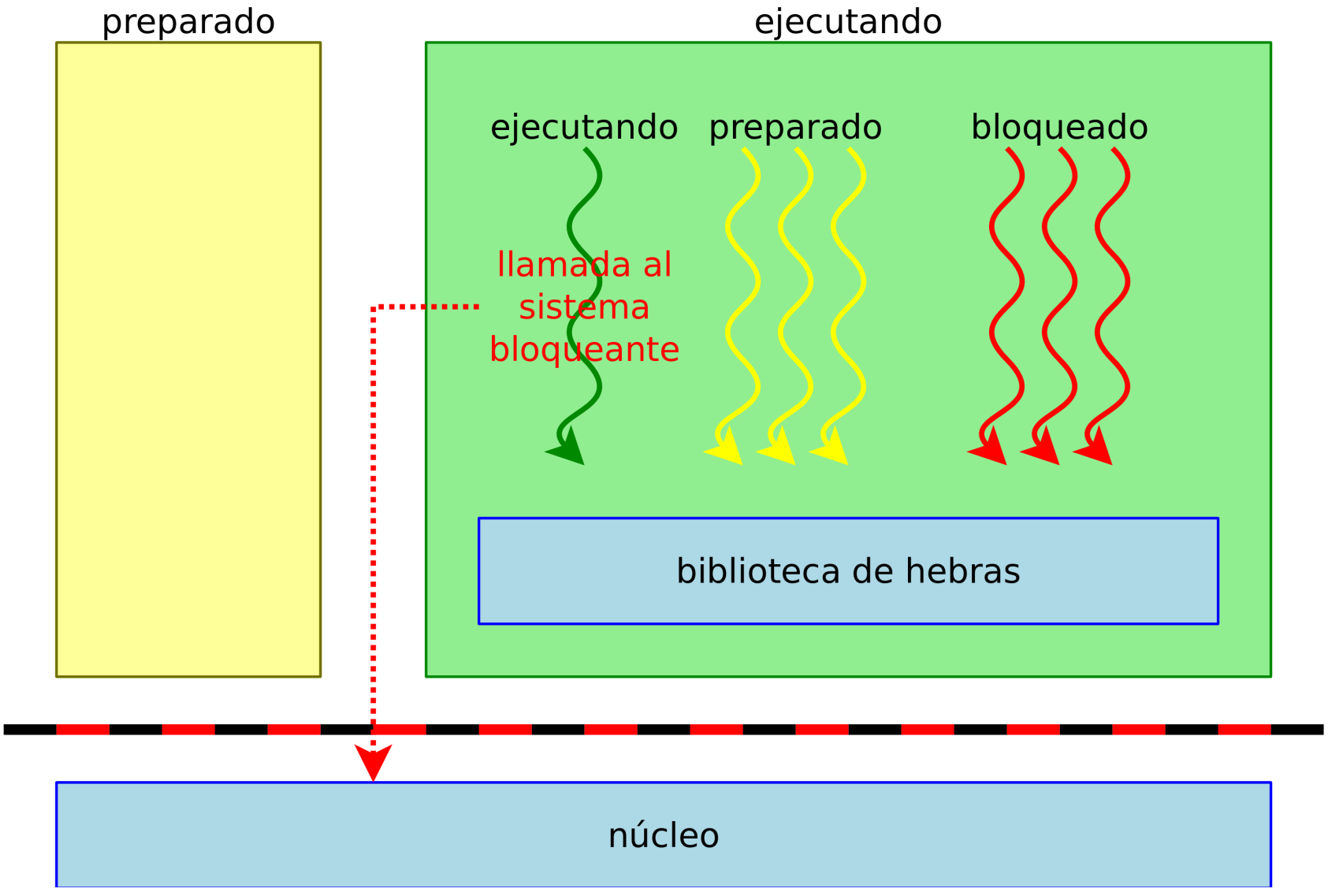
- Supongamos que un proceso tiene **t** hebras de las que **t-1** están bloqueadas y sólo **1** está preparada o ejecutándose.
- ¿En que estado está el proceso: ejecutándose, preparado o bloqueado?
- Establezcamos la siguiente relación en base al procesador:

  ejecutando ≥ **preparado** ≥ **bloqueado**
- Mientras una **hebra** esté **ejecutándose** el **proceso** permanecerá en el estado **ejecutando** independientemente de cuántas hebras posea en otros estados.

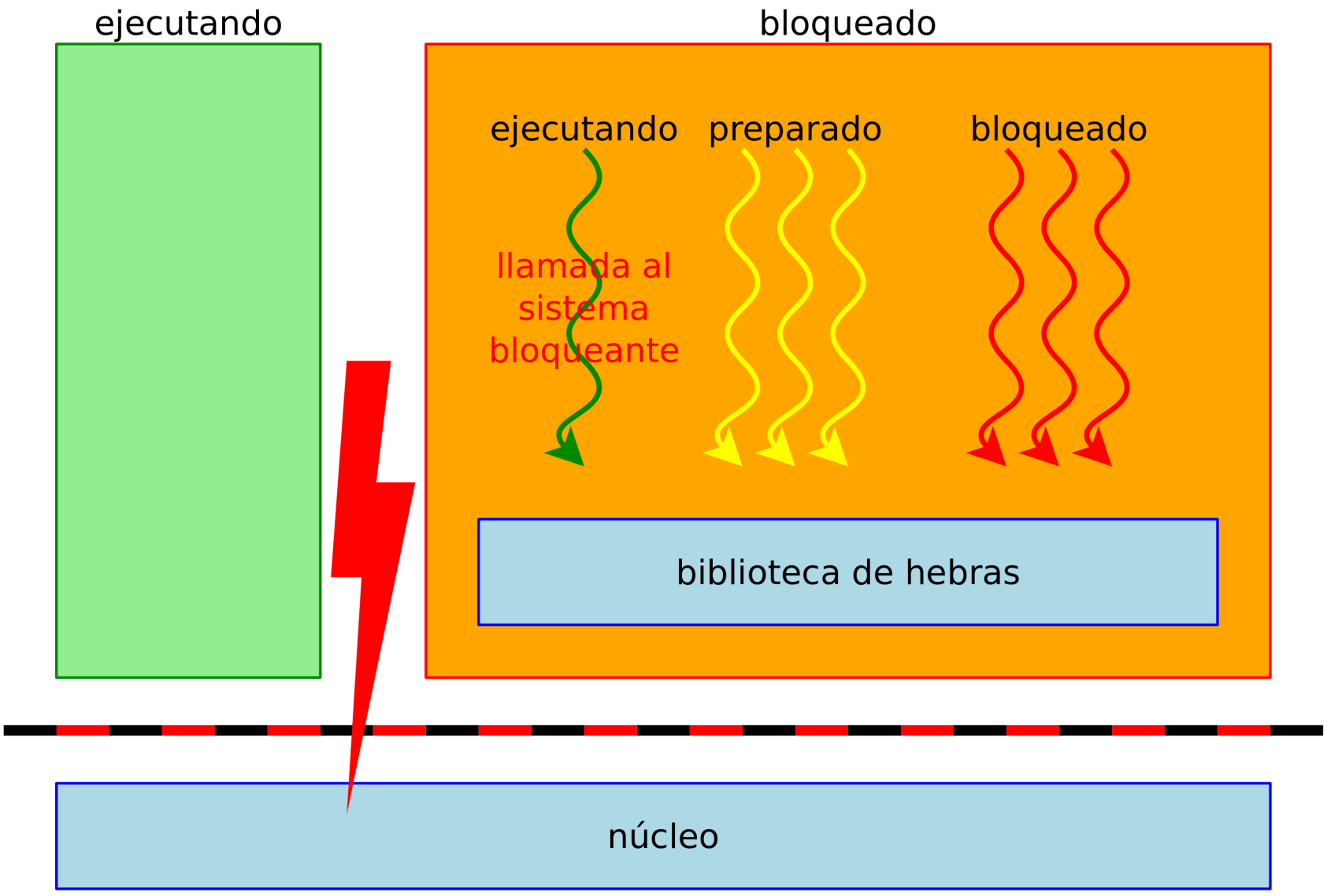
Procesos y hebras tipo **usuario**

- Aunque el núcleo no es consciente de la existencia de las hebras de usuario es responsable de gestionar la actividad de su proceso anfitrión.
- Ejemplo: cuando una hebra de usuario realiza una llamada al sistema bloqueante el proceso que la contiene debe bloquearse dentro del núcleo.
- Desde el punto de vista de la biblioteca de hebras, la hebra llamadora sigue en estado ejecutando, al menos virtualmente a nivel de usuario.
- Conclusión: el **estado** de una **hebra** tipo usuario es **independiente** del estado de su **proceso** anfitrión.

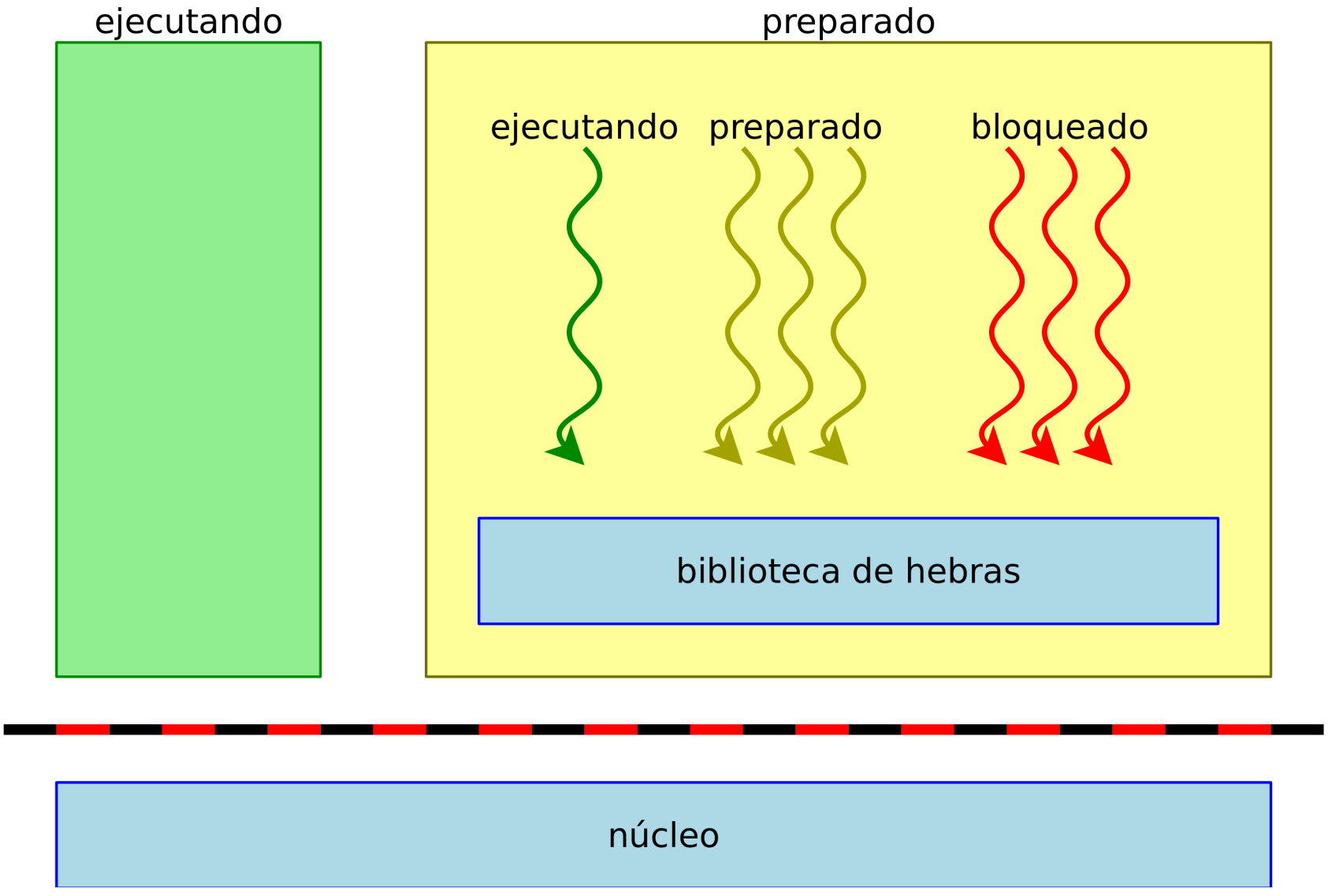
Independencia entre estados de hebras y procesos (1)



Independencia entre estados de hebras y procesos (2)



Independencia entre estados de hebras y procesos (3)



¿Cómo se bloquea una hebra de usuario?

- Hay funciones de la biblioteca de hebras que permiten bloquear y desbloquar una hebra:
  - c++11: `join()`, `yield()`.
  - pthread: `phread_join()`, `pthread_yield()`.
  - java: `wait()`, `notify()`.
- La llamada a una de estas funciones bloquea sólo a la hebra llamadora y provoca la ejecución del planificador de la biblioteca para que seleccione otra hebra que ejecutar.
- ¿Qué hacer si no existe ninguna otra hebra preparada? $\Longrightarrow$ devolver el control al núcleo para que pueda ejecutar otro proceso con `sched_yield()`.

Expulsión de hebras tipo usuario

¿Cómo prevenir que una hebra tipo usuario acapare el procesador?

- **Cooperación:**
  - cada hebra debe devolver el control a la biblioteca periódicamente.
  - mecanismo ya estudiado: `yield()`
- **Expulsión** (*"preemtion"*):
  - La biblioteca de hebras solicita al núcleo que le envíe una señal periódicamente.
  - La señal devuelve el control a la biblioteca para que pueda escoger otra hebra para ejecutar.

- Ventajas:
  - El núcleo conoce que existen y puede gestionarlas eficazmente.
  - Puede asignar diferente prioridad a cada una.
  - Puede cambiar entre hebras de un mismo proceso.
- Inconvenientes:
  - Su gestión requiere de llamadas sistema.
  - Las operaciones serán más costosas.

- Ventajas:
  - Las operaciones son entre 10 y 100 veces más rápidas.
  - El estado de las hebras es muy pequeño: procesador y pila.

- Inconvenientes:
  - Cualquier evento en cualquier hebra bloquea el proceso completo.
  - No podemos utilizar varios procesadores de forma simultánea.
    - El núcleo sólo conoce un contexto de ejecución.
  - El núcleo no puede tomar buenas decisiones de planificación:
    - podría planificar un proceso con todas sus hebras bloqueadas u ociosas.
    - podría quitar el procesador a una hebra que posea un cerrojo.

Resumen

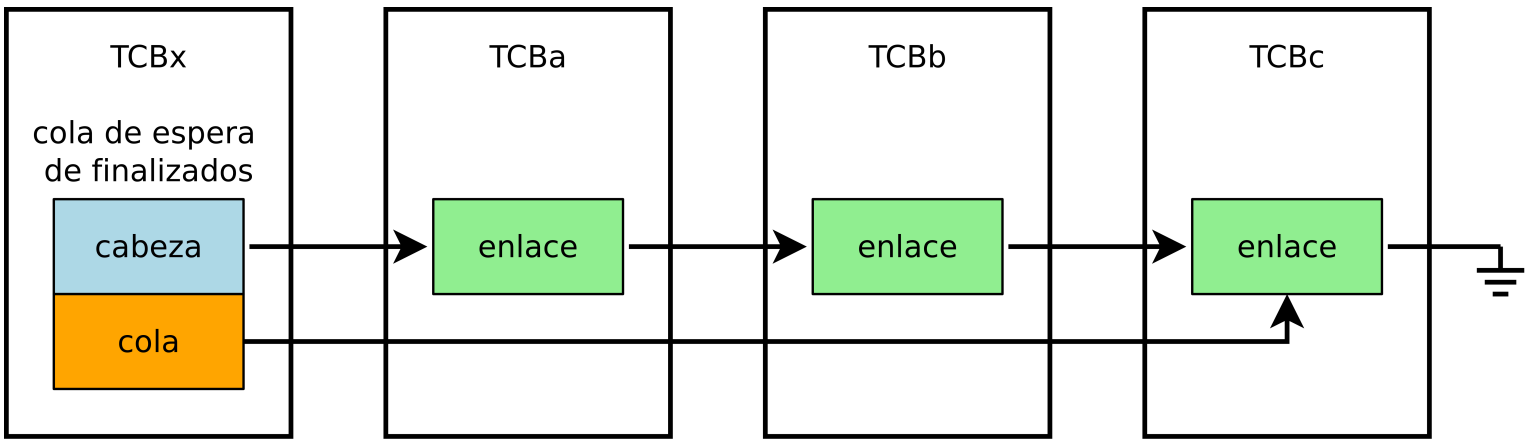
- Establecer **estados** si y sólo si estos son **útiles**.
- Los estados de las **hebras** y de los **procesos** son **diferentes**.
- Una **hebra tipo usuario** puede estar en estado **ejecutando** mientras el **proceso** que la contiene está **bloqueado**.
- Una **hebra tipo núcleo** puede estar **bloqueada** mientras otras hebras tipo núcleo del mismo **proceso** están **ejecutándose**.

Creación de hebras

- `thread()`, `pthread_create()` no es igual que la llamada `fork()` de UNIX.
  - `fork()` crea un nuevo proceso así que tendrá que crear un nuevo espacio de direcciones .
- Crear una hebra es como una **llamada a procedimiento asíncrona**.
  - ejecuta un procedimiento en otra hebra.
  - la hebra llamadora no debe esperar a que finalice.
  - si quiere esperar debe hacerlo de forma explícita: `join`, `pthread_join()`.
- ¿Y si la hebra quiere finalizar?
  - `~thread()`, `pthread_exit()` y `exit()` hacen básicamente lo mismo.

Espera de la finalización de una hebra

- Una hebra puede esperar a que otra finalice mediante `join()`, `pthread_join()`.
  - La hebra cuya finalización se espera debe ser colocada en la lista de espera de la hebra que llama a `join()`.
- ¿Cuál es el sítio más lógico para colocar esta cola de espera?
  - El interior del TCB de la hebra que ejecutó `join()`.



- Muy parecido a la llamada al sistema `wait()` de UNIX.
  - Permite esperar la finalización de descendientes.

Comparación hebra/procedimiento

- Ejemplo: llamada a `b()` desde `a()`:

```
a() { b(); }
b() { trabajo útil; }
```

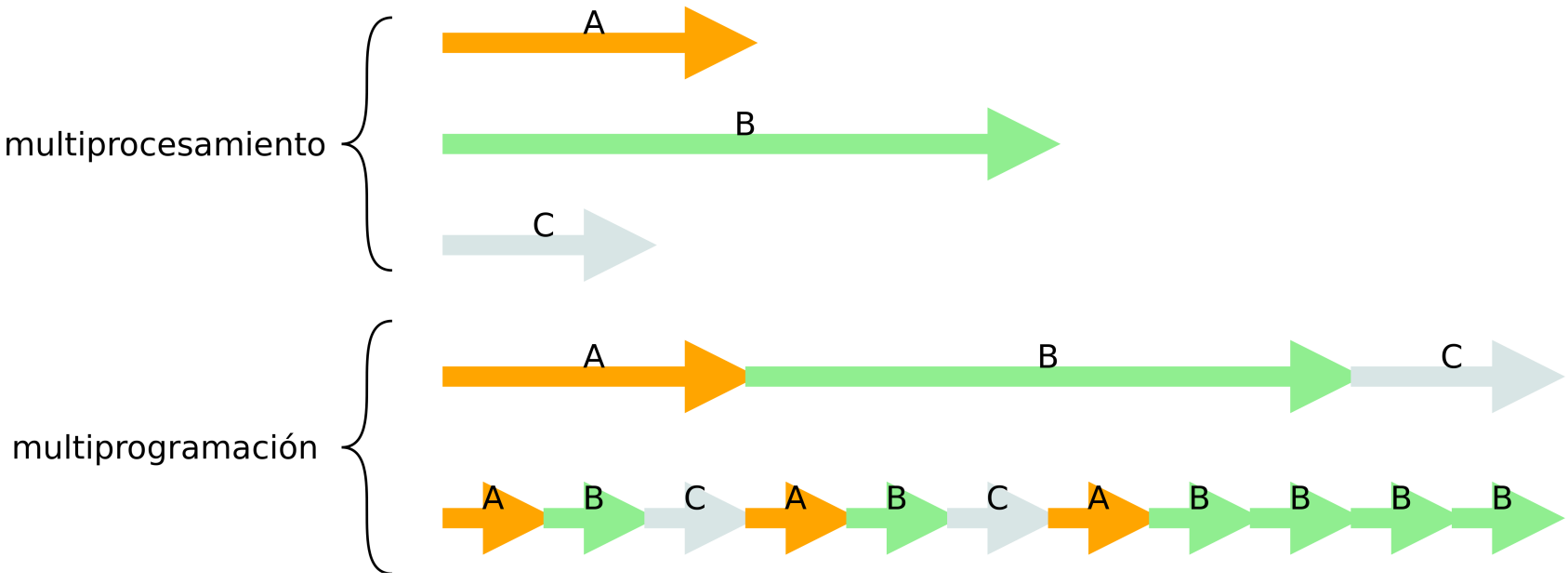
- Podemos transformar `a()` en `a'()` de la siguiente forma:

```
auto a' = std::sync(a);
-----------------------
a'()
{
    pthread_create(id, b);
    pthread_join(id);
}
```

- ¿Por qué no hacemos esto para cada procedimiento?
  - Por la sobrecarga del cambio de contexto (tiempo).
  - Por la sobrecarga del uso de memoria para pilas (espacio).
  - Por la sobrecarga necesaria para sincronización (comunicaciones).

Modelos de actividad

- Recordemos las definiciones de...
  - Multiprocesamiento: múltiples procesadores.
  - Multiprogramación: múltiples procesos.
  - Multihebra: múltiples hebras por proceso.
- ¿Qué significa ejecutar dos hebras concurrentemente?
  - El planificador puede ejecutar las hebras en cualquier orden.



Hebras, concurrencia y corrección

- Si el activador puede ejecutar las hebras en cualquier orden, los programas deben funcionar en cualquier caso:
  - ¿Sabrías comprobar esto?
  - ¿Cómo puedes saber si tu programa funciona?
- Las hebras **independientes**...
  - no comparten su estado con otras.
  - son **deterministas** → las entradas determinan las salidas.
  - son **reproducibles** → reproduciendo las condiciones iniciales obtendremos siempre los mismos resultados.
  - el orden de planificación no importa.
- Las hebras **cooperativas**...
  - comparten su estado.
  - son **no-deterministas**.
  - son **no-reproducibles**.
- El no-determinismo y la no-reproducibilidad propiciarán la aparición de errores y dificultarán su resolución.

Las interacciones complican la depuración

- ¿Es todo proceso de verdad independiente?
  - Todo proceso comparte los recursos del sistema: ficheros, red...
  - Ejemplo: un controlador defectuoso puede hacer que una hebra A haga fallar a otra hebra independiente B.
- Probablemente no os habréis dado cuenta de lo mucho que dependemos de la reproducibilidad para encontrar fallos.
  - `x = y / rand(); // mejor con srand(0)`
- Los errores no deterministas son realmente difíciles de encontrar $\longrightarrow$ `fib-mh2.cc (práctica 4)`.
  - mapa de memoria del núcleo y los programas de usuario.
    - depende de la planificación y la versión del SO.
  - E/S peculiar:
    - forma de teclear de como clave de identificación.

¿Por qué permitir que las hebras cooperen?

- La gente coopera, así que es normal que los ordenadores lo hagan también.
  - El no-determinismo y la no-reproducibilidad de las personas es un problema notable.
- 1ª ventaja: **recursos compartidos**.
  - Un ordenador, muchos usuarios.
  - Una cuenta bancaria, muchos cajeros automáticos → ¿qué pasaría si los cajeros sólo se sincronizasen una vez al día?
- 2ª ventaja: **aumento de velocidad**.
  - Permite solapar E/S y cómputo.
  - Multiprocesadores: división del trabajo para su ejecución paralela.
- 3ª ventaja: **modularidad**.
  - Es más importante de lo que pueda pensarse.
  - Dividir una problema en partes más simples.

Ejemplo: servidor web

- Un servidor web tiene que atender muchas peticiones.
- Versión **no cooperativa**:

```
while(true)
{
    conexión = aceptar_conexión();
    if (fork() == 0)
    {
        servir_página(conexión);
        exit(0);
    }
}
```

- ¿Qué ventajas y desventajas tiene esta técnica?

Ejemplo: servidor web multihebra

- Versión **cooperativa**:

```
while(true)
{
    conexión = aceptar_conexión();
    pthread_create(servir_página, conexión);
}
```

- Parece lo mismo, pero tiene muchas ventajas:
  - Pueden compartir la caché de ficheros en memoria, scripts,...
  - Las hebras son más baratas de crear así que habrá una menor sobrecarga por petición.
- ¿Tendría sentido utilizar hebras tipo usuario?
  - ¿Qué pasará cuando una hebra intente leer una página web desde el disco?
- ¿Y los efectos de Slashdot y los ataque de denegación de servicio?

Ejemplo: servidor web con una “piscina” de hebras (1)

- Las versiones anteriores tienen un problema: peticiones ilimitadas.
  - Cuando un sitio web se vuelve demasiado famoso su rendimiento decae.
- Solución: alojar un número máximo de hebras trabajadoras que representarán el máximo grado de multiprogramación.

```
maestra()
{
  for(int i = 0; i < N; ++i)
    pthread_create(trabajadora);

  while(true)
  {
    conexión = aceptar_conexión();
    cola.meter(conexión);
  }
}


trabajadora()
{
  while(true)
  {
    cola.sacar(conexión);
    servir_página(conexión);
  }
}
```

Ejemplo: servidor web con una “piscina” de hebras (2)

Ventajas:

- Crear muchas hebras consume mucho espacio y tiempo.
- Destruir hebras consume mucho tiempo.
- La lenta creación de nuevas hebras incrementa el tiempo de espera por petición.
- La lenta destrucción de hebras puede impedir el normal progreso de otros procesos por falta de recursos.