# Arquitectura de Sistemas

## Hebras

---

Gustavo Romero López

Updated: 28 de marzo de 2025

Departamento de Ingeniería de Computadores, Automática y Robótica

# Índice

# Lecturas recomendadas

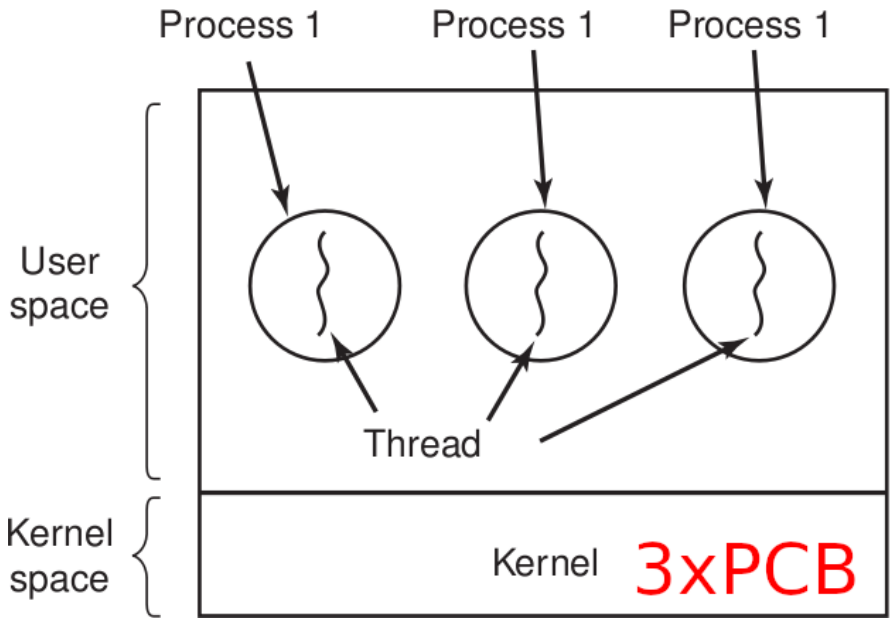
| | |
|---|---|
| Tanuenbaum | Sistemas Operativos Modernos (2.2) |
| Silberschatz | Fundamentos de Sistemas Operativos (4) |
| Stallings | Sistemas Operativos (4) |

# Definición

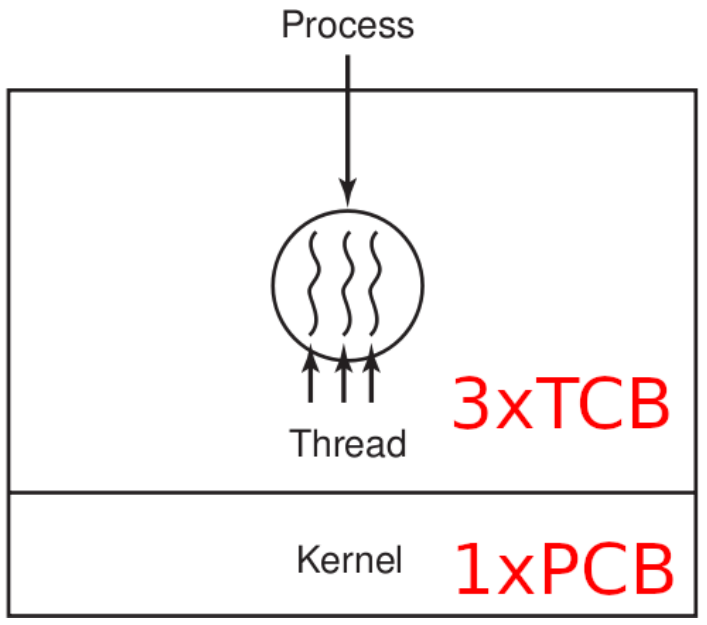
- Proceso en un SO *"tradicional"* = espacio de direcciones + hebra de control.
- Proceso = contenedor de recursos + hebra de ejecución.
- Una hebra se ejecuta **dentro** de un proceso.
- Proceso y hebra son conceptos diferentes y suelen ser tratados por separado.
  - La función de un proceso es **agrupar recursos**.
  - Las hebras son **entidades planificables** para su ejecución sobre un procesador.
- Las hebras añaden a los procesos la capacidad de ejecutar varios conjuntos de instrucciones de forma **concurrente** dentro de un mismo entorno de procesamiento.
- Ejecutar varias hebras en **paralelo** es parecido a ejecutar varios procesos salvo por que comparten sus recursos.
- También llamados **procesos ligeros**, **subprocesos** o **fibras**.

# Comparación procesos/hebras

- Como ejecutar 3 hebras en función de la relación del núcleo del SO con ellas:
  - Con su ayuda necesitamos 3 PCB[1] y 0 TCB[2].
  - Sin su ayuda necesitamos 1 PCB y 3 TCB.




3 procesos con 1 hebra



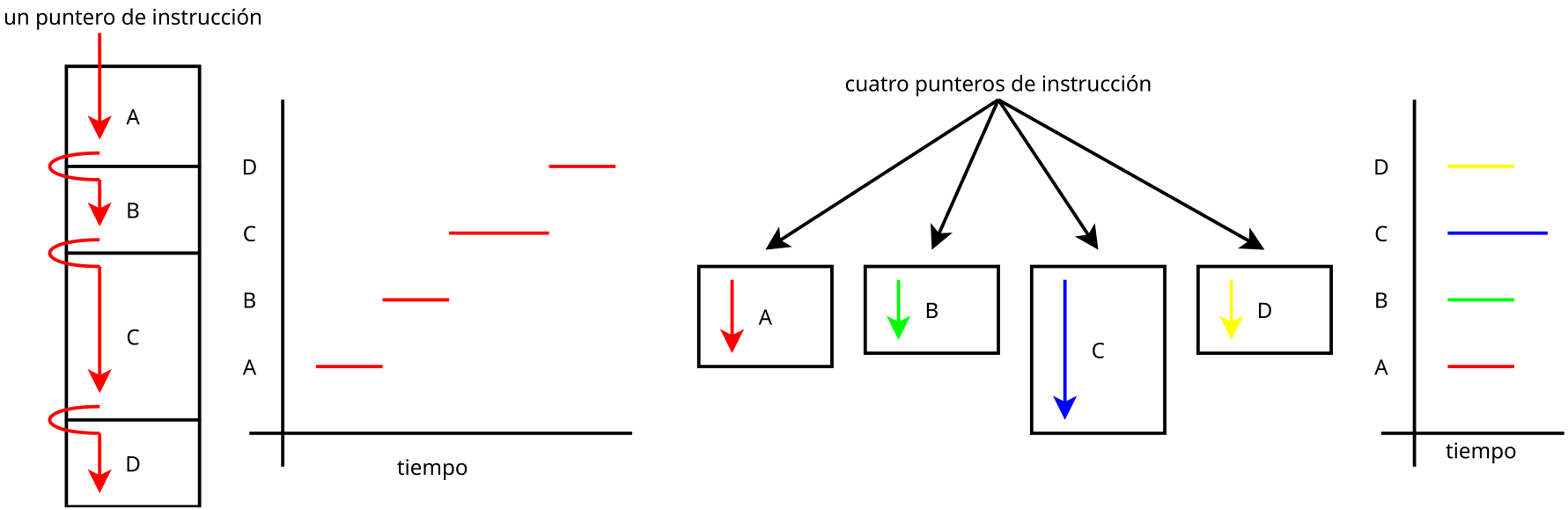

1 proceso con 3 hebras

[1] *Process Control Block*

[2] *Thread Control Block*

# Ejecución multihebra

- Multihebra = ejecución **concurrente** de hebras en el interior de un proceso.
  - En un sistema con un único procesador las hebras se **alternan** en el uso del procesador.
  - En un sistema multiprocesador tantas hebras como procesadores pueden ejecutarse en **simultáneamente**.



# Ventajas de la multihebra

- Mejora de rendimiento:
  - Las hebras se crean más rápido que los procesos.
  - Las hebras se finalizan más rápido que los procesos.
  - Cambiar de hebra puede ser más rápido que de proceso.
  - Es posible la comunicación al margen del núcleo.
- Ejemplos de uso:
  - Trabajo simultáneo en primer y segundo plano, ej: procesador de texto, servidor de ficheros,...
  - Procesamiento asíncrono, ej: copia de seguridad.
  - Velocidad de ejecución: descomposición de problemas y procesamiento paralelo, ej: solapamiento entre obtención de datos y cálculo.
  - Mejor Modularidad: fácil subdivisión de tareas complejas en otras más simples.

# Independencia/Cooperación

- Las diferentes hebras de un proceso **no** son **independientes** como las que se encuentran en diferentes procesos.
- Las hebras de un proceso **comparten** sus **recursos**: espacio de direcciones, variables globales, ficheros abiertos,...
- **No** hay **protección** entre hebras porque...
  - **es imposible**: comparten recursos.
  - **no es necesario**: diferentes procesos pueden pertenecer a diferentes usuarios, en cambio, todas las hebras de un proceso pertenecen al mismo.
- Podemos tener datos privados para cada hebra:

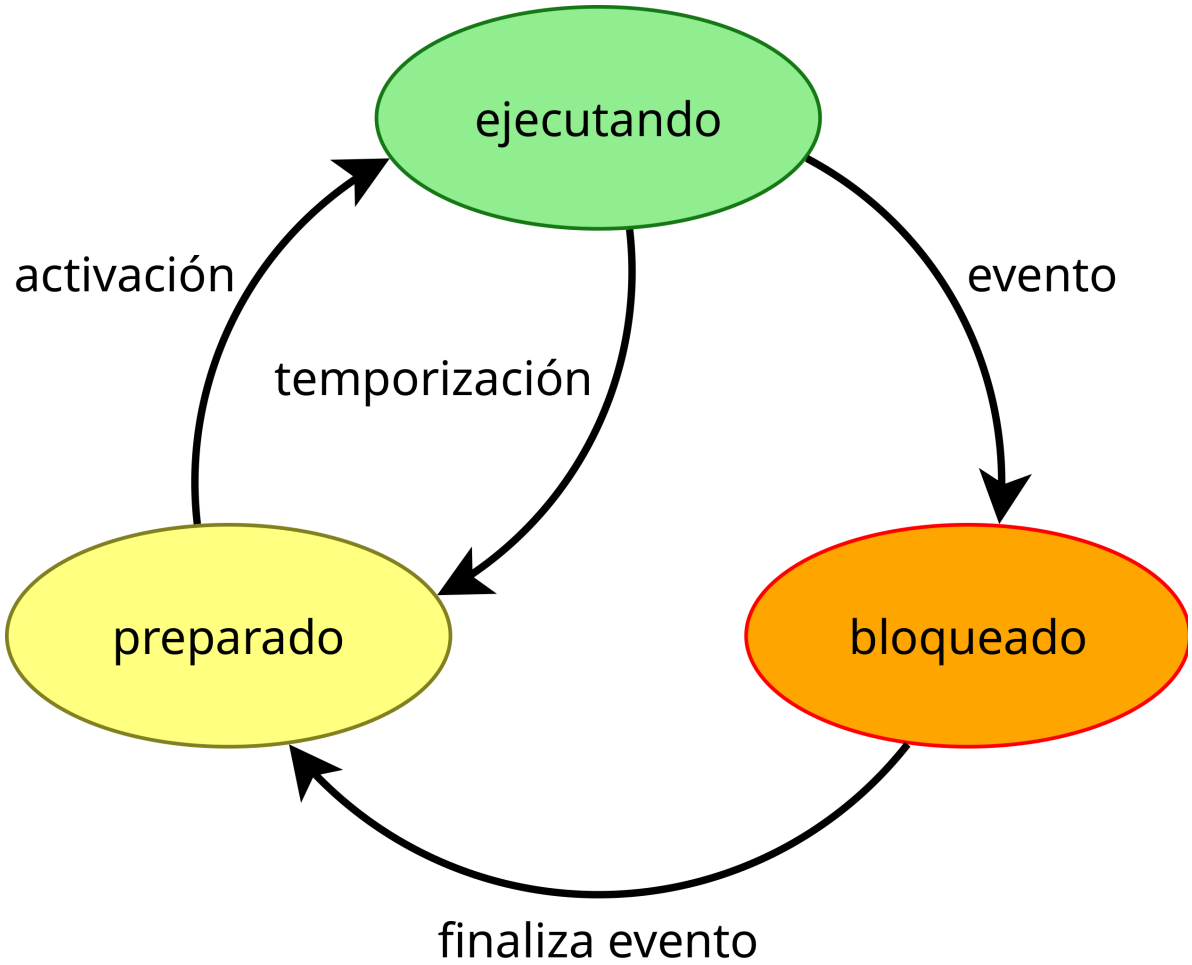
  C++11: `thread_local int x = 0;`

# Recursos necesarios

| **por proceso** | **por hebra** |
|---|---|
| identificador de proceso | |
| espacio de direcciones | identificador de hebra |
| variables globales | registros |
| ficheros abiertos | pila |
| procesos hijos | estado |
| alarmas pendientes | datos privados |
| señales y manejadores de señales | |
| información contable | |
| ... | |

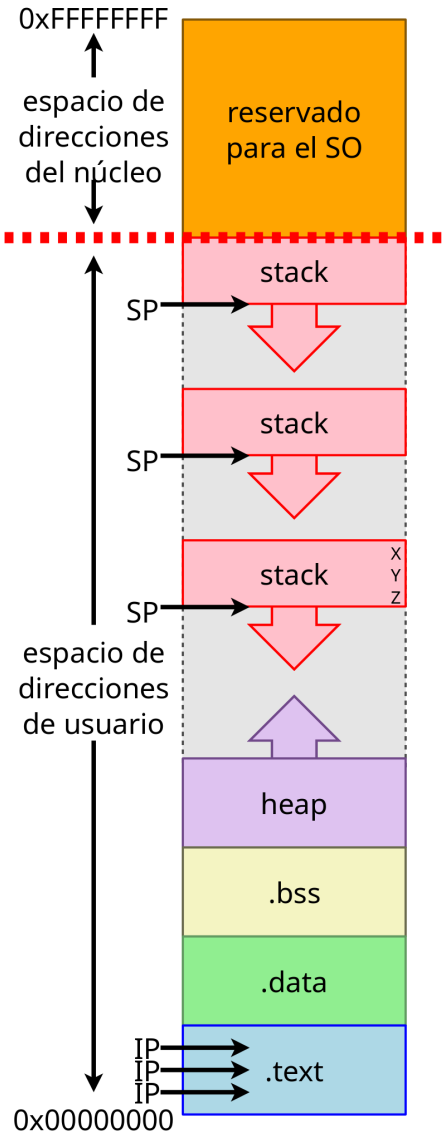
# Estado de un hebra

- Al igual que los procesos, una hebra puede pasar por diferentes estados a lo largo de su ejecución.



# Espacio de direcciones

- Cada hebra necesita su **propia pila**.
- Cada pila contiene un **marco de función** por cada función llamada y de la que todavía no se ha retornado.
- Si en un hebra ejecutamos el procedimiento `X`, este llama al `Y`, y este llama `Z`, durante la ejecución de `Z` la pila contendrá los marcos de `X`, `Y` y `Z`.



# Gestión de hebras

- Para gestionar las hebras, al igual que sucede con los procesos, necesitamos una serie de **llamadas al sistema**.
- Los procesos multihebra comienzan su ejecución con una **única hebra** y crean más en caso de necesitarlas: `pthread_create()`/`std::thread::thread()`.
- Las hebras pueden mantener una relación jerárquica o ser creadas todas iguales.
- Cada hebra suele tener un identificador único.
- Es necesaria un llamada al sistema para terminar con una hebra: `pthread_exit()`/`std::thread::~thread()`.
- Al igual que los procesos, podemos hacer que las hebras sincronicen su funcionamiento (equivalente a `wait()`): `pthread_join()`/`std::thread::join()`.

# Gestión de hebras

- Otra llamada habitual es `pthread_yield()` / `std::this_thread::yield()`, que permite a una hebra ceder voluntariamente el procesador.
  - Muy importante porque puede que la interrupción de reloj no las afecte o no se pueda imponer el tiempo compartido.
  - Las hebras **deben** cooperar.
- Las hebras introducen nuevos problemas en el sistema:
  - `fork()`: si un padre tiene varias hebras, ¿debería el hijo tenerlas también? ⟶ no
  - Si un proceso tiene una hebra bloqueada en espera de una entrada desde el teclado, ¿debería tenerla también un hijo?
  - Este mismo problema puede darse con otros recursos: ficheros, conexiones de red, gestión de memoria,...
  - solución más rápida y sin problemas ⟶ no duplicar.

# Motivos para el uso de hebras

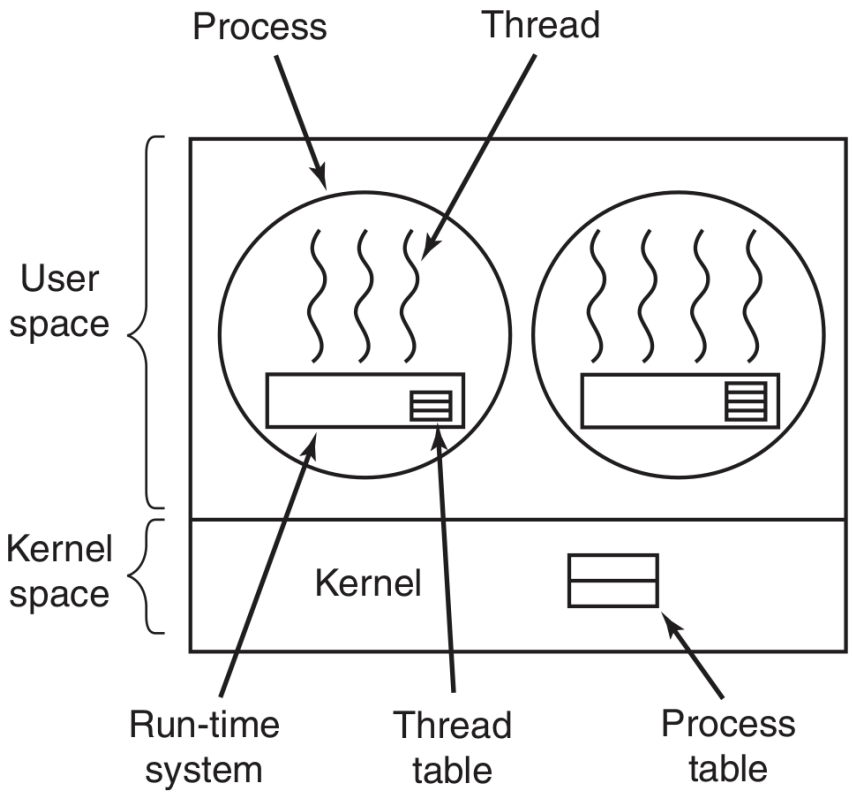
- **Compartir** de recursos, ej: servidor web.
  - evolución: contenido estático (+E/S) $\longrightarrow$ dinámico (+CPU).
- **Facilidad** (economía) de creación y destrucción.
- Mejora de **rendimiento**:
  - gran mejora en los limitados por E/S.
  - pequeña mejora en los limitados por CPU.
- En sistemas con múltiples procesadores es posible un verdadero **paralelismo**.
- **Simplicidad** de programación, ej: procesador de texto.

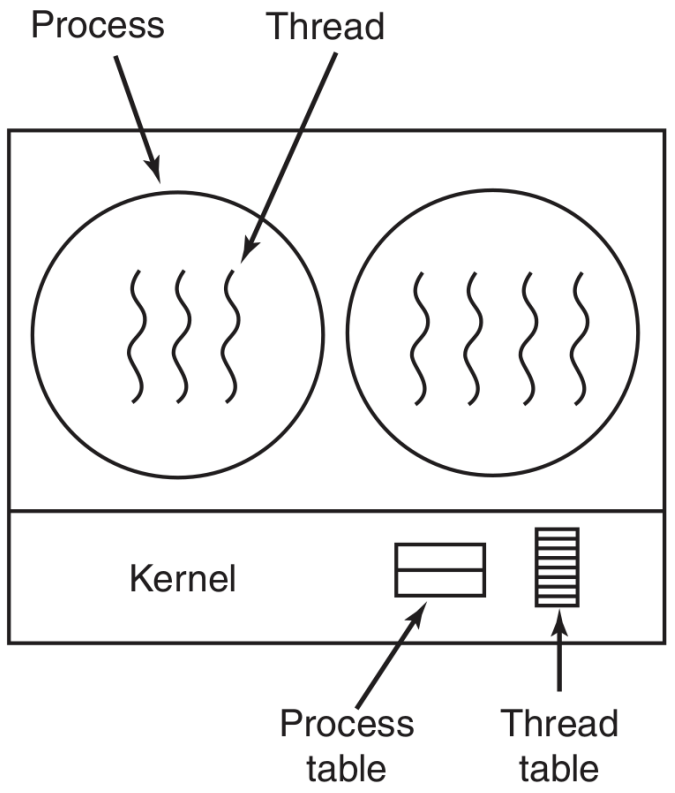
# Ejemplos de uso de hebras

- Procesador de textos: una hebra por tarea.
  - Interacción con el usuario.
  - Corrector ortográfico/gramatical.
  - Guardar automáticamente y/o en segundo plano.
  - Mostrar resultado final en pantalla (`WYSIWYG`).
- Hoja de cálculo:
  - Interacción con el usuario.
  - Actualización en segundo plano.
- Servidor web: dos tipos de hebras.
  - Interacción con los clientes (navegadores).
  - Gestión del caché de páginas.
- Navegador: varios tipos de hebras.
  - Interacción con el usuario.
  - Interacción con los servidores (web, ftp, ...).
  - Dibujo de la página (1 hebra por pestaña).

# Tipos de hebras

- Hebras tipo usuario.
- Hebras tipo núcleo.
- Hebras híbridas.



# Hebras tipo usuario

- El sistema de hebras se coloca por completo en el espacio de usuario.
- El núcleo no sabe nada de ellas y por lo que a él respecta se dedica a planificar procesos ordinarios.
- Las primeras implementaciones eran de este tipo.
  - la mayoría de SOs han evolucionado hacia el uso de hebras tipo núcleo.
- Cada proceso necesita una tabla de hebras.
  - Análoga a las tablas de procesos.
  - Gestionada por el sistema de ejecución de hebras.
  - Implementado por las bibliotecas de los lenguajes de programación.

# Ventajas de las hebras tipo usuario

- Pueden implementarse sobre **cualquier SO**.
- La **conmutación** entre hebras es varios órdenes de magnitud más **rápida** que la de procesos.
- Cuando una hebra termina de ejecutarse (`pthread_exit()` / `pthread_yield()`) puede cambiarse a otra **sin involucrar al núcleo** con lo que ahorraremos cambios de proceso y modo.
- Cada proceso puede utilizar un algoritmo de **planificación personalizado** para sus hebras.

| operación | hebras usuario | procesos | p/u |
|---|---|---|---|
| proceso nulo | $34\mu s$ | $11300\mu s$ | $\times$**333** |
| signal/wait | $37\mu s$ | $1840\mu s$ | $\times$**50** |

# Inconvenientes de las hebras usuario

- **Llamadas al sistema bloqueantes**: detener una hebra implica detener el proceso completo, ej: lectura de teclado, fallo de página. Soluciones:
  - modificar el SO para hacer sus llamadas no bloqueantes $\Rightarrow$ no queremos/podemos hacerlo.
  - comprobar si las llamadas bloqueantes tendrán éxito antes de efectuarlas, ej: `select/read` $\Rightarrow$ método engorroso.
- Como no existe interrupción de reloj no podremos ejecutar una hebra hasta que otra deje libre el procesador **voluntariamente** (cooperación obligatoria).
  - solución: solicitar interrupción periódica $\Rightarrow$ pérdida de rendimiento.
- En sistemas multiprocesador no podremos asignar más de **un procesador** a cada proceso.

# Hebras tipo núcleo

- El núcleo conoce la existencia de las hebras y se encarga de gestionarlas.
- No se necesita un sistema de gestión de hebras ni tablas de hebras en el interior de cada proceso.
- En el interior del núcleo ya disponemos de las tablas de procesos, las tablas de hebras en realidad son parte de ellas.
- Algunos sistemas no distinguen entre procesos y hebras $\Rightarrow$ Linux usa tareas (*task*).
- La **gestión** de hebras se hace a través de **llamadas al sistema** $\Rightarrow$ mayor coste de operación[3].

[3] ¿Conoce la diferencia de coste entre llamada a función y llamada al sistema?

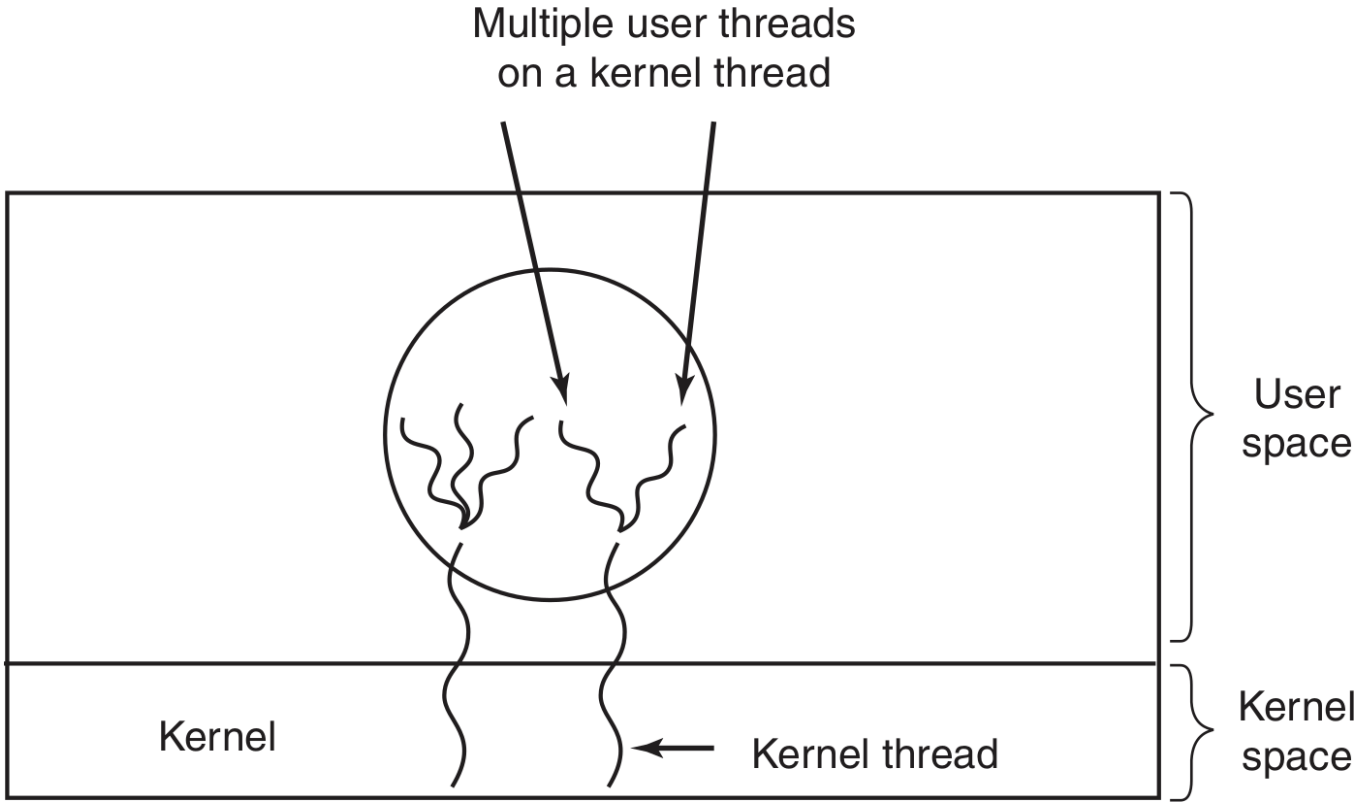
# Ventajas e inconvenientes de las hebras del núcleo

- Ventajas:
  - No requieren llamadas al sistema **no bloqueantes**.
  - Si una hebra se bloquea o excede su tiempo de ejecución otra puede ser ejecutada.
  - En sistemas multiprocesador tantas hebras de un mismo proceso como procesadores existan pueden ejecutarse en **paralelo**.
- Inconvenientes:
  - **Coste** de operación mucho mayor que hebras tipo usuario.
  - Importante el **reciclaje** de hebras: finalizado $\Rightarrow$ nuevo.

| operación | hebras usuario | hebras núcleo (n/u) | procesos (p/n) |
|---|---|---|---|
| proceso nulo | $34\mu s$ | $948\mu s$ **×28** | $11300\mu s$ **×12** |
| signal/wait | $37\mu s$ | $441\mu s$ **×12** | $1840\mu s$ **×4** |

# Hebras híbridas

- Intentan reunir las ventajas y evitar los inconvenientes.
- Implementación:
  - Es necesario disponer de hebras tipo núcleo en el SO.
  - Una o más hebras tipo usuario se reparten el tiempo de procesador de una hebra tipo núcleo.



# Activaciones del planificador

- Método propuesto por Thomas E. Anderson (1992).
- El núcleo asigna cierto número de **procesadores virtuales** o **procesos ligeros** a cada proceso.
  - Inicialmente suele asignarse sólo uno.
  - Podemos solicitar más y/o devolverlos.
  - El núcleo puede concederlos y/o retirarlos.
- Cuando el núcleo detecta que una hebra se bloquea avisa al sistema de gestión de hebras en espacio de usuario mediante una llamada directa (*"upcall"*).
- Una vez avisado el planificador a nivel usuario puede seleccionar otra hebra para ejecutar.
- Cuando la hebra se desbloquea de nuevo el núcleo avisa al sistema de gestión de hebras en espacio de usuario.

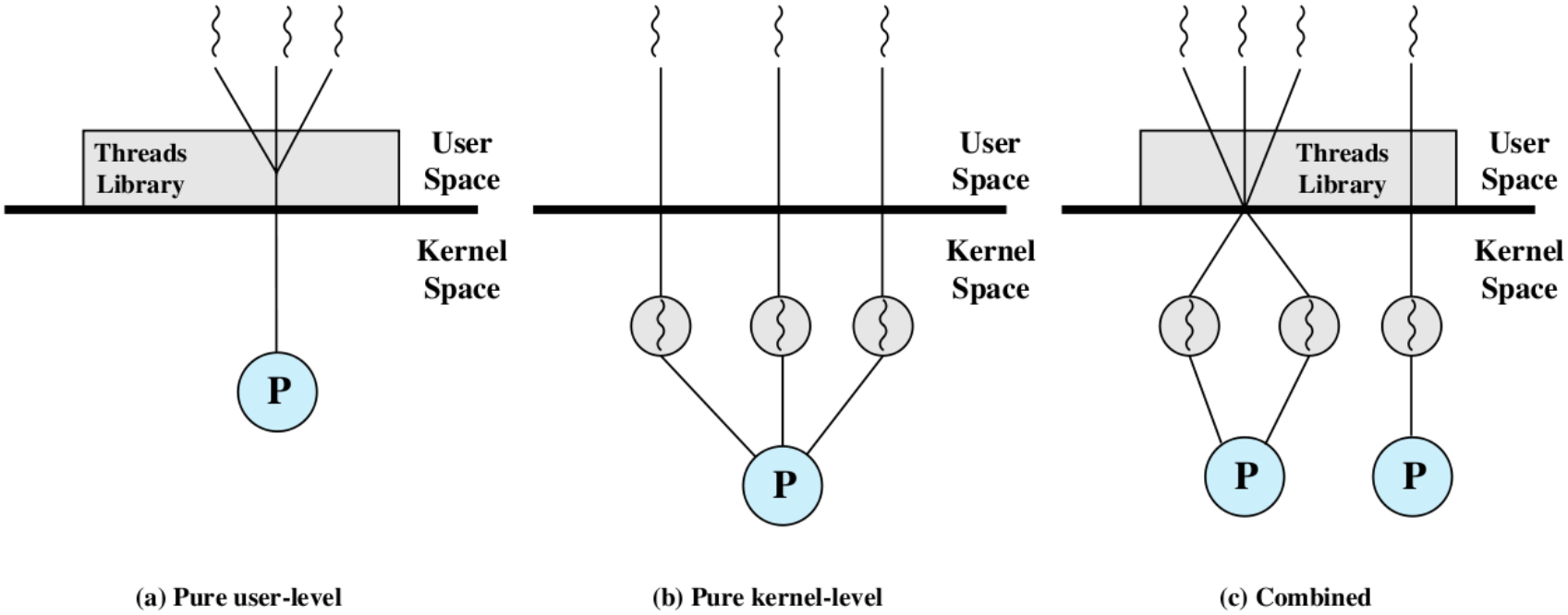
# Ventajas e inconvenientes del enfoque híbrido

- Ventajas:
  - Dos formas de manejar las llamadas al sistema bloqueantes:
    - Una llamada bloqueante en una hebra de usuario sólo bloquea su correspondiente hebra del núcleo hasta que finalice la causa del bloqueo mientras que el resto de hebras del proceso continúan ejecutándose.
    - La hebra del núcleo almacena la causa del bloqueo y avisa al planificador a nivel usuario para que cambie a otra hebra de usuario hasta que finalice la causa del bloqueo.
- Inconvenientes:
  - El SO debe disponer de hebras tipo núcleo.
  - Hay dos niveles de planificación:
    - El núcleo planifica las hebras del núcleo.
    - Las hebras de usuario son planificadas por una biblioteca.
    - Ambos planificadores deben cooperar.
  - Problema: las llamadas directas violan los principios de los sistemas por capas.

# En resumen...

- Hebras tipo usuario:
  - Rápidas y fáciles de implementar.
  - No planificables por el núcleo.
  - Se bloquean ante cualquier evento.
  - Sólo pueden usar un procesador.
  - Requieren cooperación entre hebras.
- Hebras tipo núcleo:
  - Involucran al SO, fraccionamiento del tiempo.
  - Operaciones de cambio y sincronización más costosas.
  - Los eventos no bloquean el proceso completo.
  - Pueden utilizarse varios procesadores en paralelo.
- Hebras híbridas:
  - Lo mejor de cada tipo, al menos en teoría.
  - Balanceo de carga: equilibrar el número de hebras tipo usuario que se ejecutan sobre cada procesador.

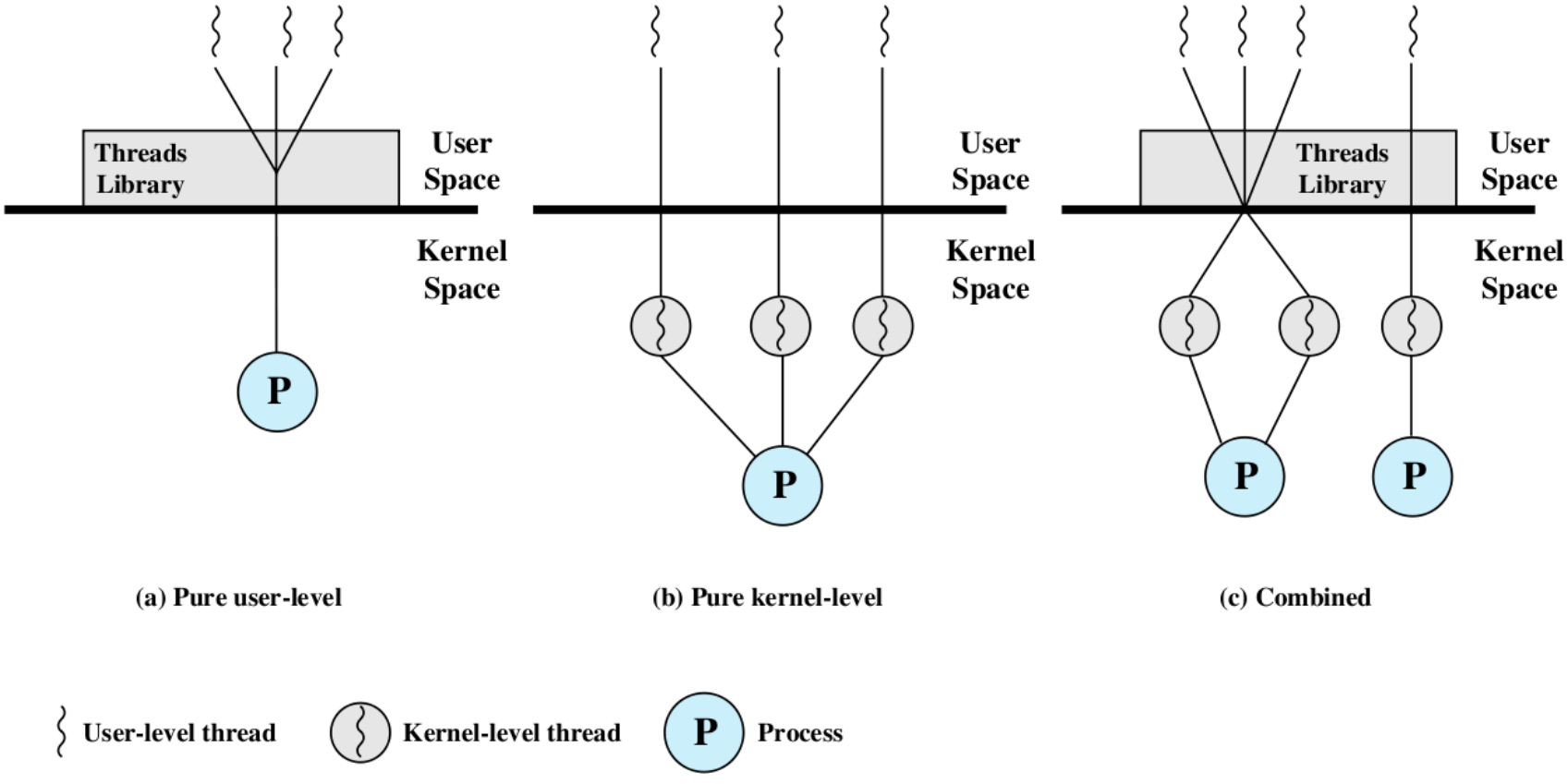
# Comparativa entre hebras y procesos




| operación\tipo | hebras usuario | hebras híbridas | hebras núcleo | procesos |
|---|---|---|---|---|
| proceso nulo | 34 | 37 | 948 | 11300 |
| signal/wait | 37 | 42 | 441 | 1840 |

Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. ACM Transactions on Computer Systems, Vol. 10, No. 1, February 1992. (Tiempos en $\mu$s)

# Relación entre tipos de hebras




| modelo | nº hebras (u/n) | ejemplos |
|---|---|---|
| muchos a uno | N:1 | GNU Portable Threads, Green Threads (Solaris) |
| uno a uno | 1:1 | Linux, MacOS X, Solaris (>8), Windows (>NT) |
| muchos a muchos | N:M | Solaris(<9), Fiber (Windows) |
| dos niveles | N:M + 1:1 | HP-UX, IRIX, Solaris (<9), Tru64 UNIX |

# Representación de hebras

- Para poder gestionar hebras necesitamos estrucuras de datos que las representen $\Longrightarrow$ **bloque de control de hebra** (**TCB**).

| **TCB mínimo** |
| --- |
| identificador de hebra (TID) |
| puntero de instrucción (IP) |
| puntero de pila (SP) |
| estado (flags) |

# Linux 6.13.8 processor.h thread_struct (28/03/2025) I

```
429 struct thread_struct {
430     /* Cached TLS descriptors: */
431     struct desc_struct    tls_array[GDT_ENTRY_TLS_ENTRIES];
432 #ifdef CONFIG_X86_32
433     unsigned long         sp0;
434 #endif
435     unsigned long         sp;
436 #ifdef CONFIG_X86_32
437     unsigned long         sysenter_cs;
438 #else
439     unsigned short        es;
440     unsigned short        ds;
441     unsigned short        fsindex;
442     unsigned short        gsindex;
443 #endif
444
445 #ifdef CONFIG_X86_64
446     unsigned long         fsbase;
```

# Linux 6.13.8 processor.h thread_struct (28/03/2025) II

```
447         unsigned long       gsbase;
448     #else
449         /*
450          * XXX: this could presumably be unsigned short.  Alternatively,
451          * 32-bit kernels could be taught to use fsindex instead.
452          */
453         unsigned long fs;
454         unsigned long gs;
455     #endif
456
457         /* Save middle states of ptrace breakpoints */
458         struct perf_event   *ptrace_bps[HBP_NUM];
459         /* Debug status used for traps, single steps, etc... */
460         unsigned long           virtual_dr6;
461         /* Keep track of the exact dr7 value set by the user */
462         unsigned long           ptrace_dr7;
463         /* Fault info: */
464         unsigned long       cr2;
465         unsigned long       trap_nr;
```

# Linux 6.13.8 processor.h thread_struct (28/03/2025) III

```
466         unsigned long           error_code;
467     #ifdef CONFIG_VM86
468         /* Virtual 86 mode info */
469         struct vm86             *vm86;
470     #endif
471         /* IO permissions: */
472         struct io_bitmap        *io_bitmap;
473
474         /*
475          * IOPL. Privilege level dependent I/O permission which is
476          * emulated via the I/O bitmap to prevent user space from disabling
477          * interrupts.
478          */
479         unsigned long           iopl_emul;
480
481         unsigned int            iopl_warn:1;
482         unsigned int            sig_on_uaccess_err:1;
483
484         /*
```

# Linux 6.13.8 processor.h thread_struct (28/03/2025) IV

```
485         * Protection Keys Register for Userspace.  Loaded immediately on
486         * context switch. Store it in thread_struct to avoid a lookup in
487         * the tasks's FPU xstate buffer. This value is only valid when a
488         * task is scheduled out. For 'current' the authoritative source of
489         * PKRU is the hardware itself.
490         */
491        u32             pkru;
492
493    #ifdef CONFIG_X86_USER_SHADOW_STACK
494        unsigned long           features;
495        unsigned long           features_locked;
496
497        struct thread_shstk     shstk;
498    #endif
499
500        /* Floating point and extended processor state */
501        struct fpu          fpu;
502        /*
503         * WARNING: 'fpu' is dynamically-sized.  It *MUST* be at
```

# Linux 6.13.8 processor.h thread_struct (28/03/2025) V

```
504         * the end.
505         */
506    };
```

# Atributos de una hebra

- Identificador de hebra (TID).
- Contexto:
  - Registros de usuario.
- Planificación.
- Pila.
- Recursos privados (opcional):
  - Datos privados.
- Otros:
  - Eventos relacionados: esperando E/S.
  - Prioridades.
  - Comunicación entre procesos.
  - Información de mapeado.
  - Propiedad y utilización de recursos.

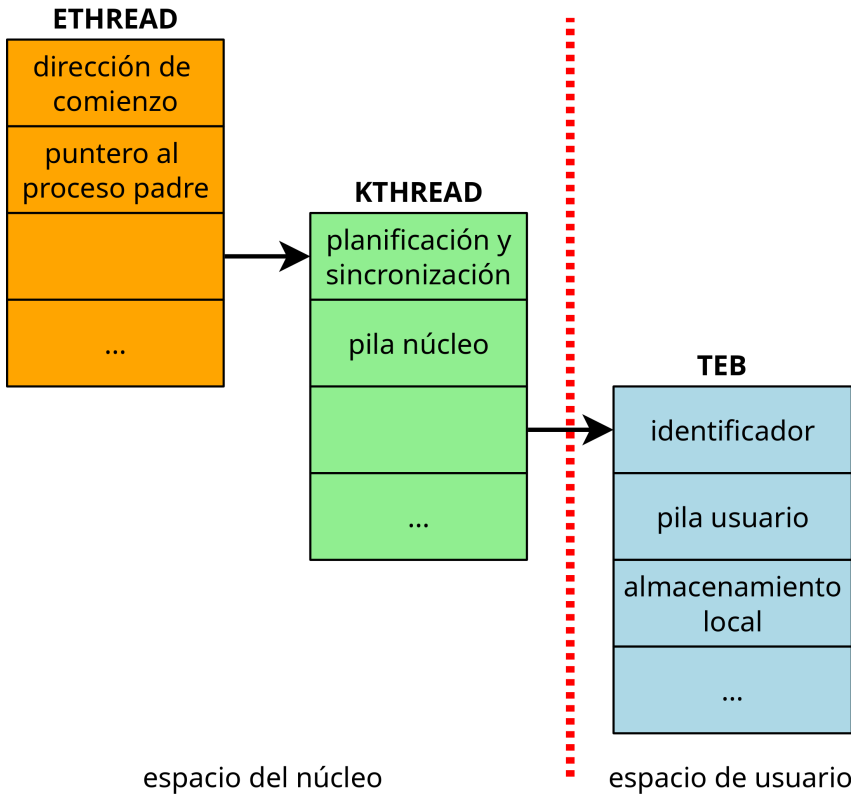
# Implementación de los bloques de control de hebra

- ¿Dónde colocar los TCBs?
  - No colocarlos aleatoriamente.
  - Tenga en cuenta la TLB y la caché.
  - Diseñarlos con sumo cuidado, tanto o más que los PCBs.
- Posibles estructuras de datos:
  - Contiguas:
    - vector
    - vector de punteros.
  - No contiguas:
    - lista.
- Posibles localizaciones de los TCBs.
  - Espacio de usuario.
  - Espacio del núcleo.
  - Repartido entre ambos: TCB = UTCB + KTCB.

# Windows (95/98/NT/2000/XP)

- Implementación de la API Win32.
- Cada proceso contiene una o más hebras.
- Implementa hebras tipo usuario y núcleo.
  - La biblioteca Fiber añade hebras híbridas.
- Componentes: identificador, conjunto de registros, pilas de usuario y núcleo, área de almacenamiento privada

- Localización de las estructuras de datos de las hebras:



# Linux

- No distingue entre procesos y hebras: **tareas** (*task*).
- Utiliza la llamada al sistema `clone()`, base de `fork()`, pero que permite un control más preciso sobre los recursos.
- Estructura de datos para representar tareas: `task_struct`.
  - Formada por punteros para optimizar el poder compartir.

```
int clone(int (*fn)(void *_Nullable), void *stack, int flags,
          void *_Nullable arg, ...  /* pid_t *_Nullable parent_tid,
                                       void *_Nullable tls,
                                       pid_t *_Nullable child_tid */ );
```

| parámetro | significado |
|---|---|
| CLONE_FS | compartir sistema de ficheros |
| CLONE_VM | compartir memoria virtual |
| CLONE_SIGHAND | compartir manejadores de señales |
| CLONE_FILES | compartir los ficheros abiertos |

# Bibliotecas de hebras

- Pthreads (POSIX threads/estándar IEEE 1003.1c).
  - API para la creación y sincronización de hebras.
  - Especificación ≠ implementación.
  - Disponible en la mayoría de los SOs.
- Hebras Win32.
- Hebras Java.
- Hebras de C++: std::thread/std::jthread y corutinas.
- Hebras de Boost: boost::thread y boost::fiber.

# pthread

cabecera: `#include <pthread.h>`

`pthread_create(id, attr, func, val)`: crea una hebra que ejecuta el código de la función `func()`.

`pthread_exit(val)`: finaliza un hebra devolviendo un valor.

`pthread_join(id, val)`: espera la finalización de una hebra y recupera el valor que esta devuelve.

`pthread_self()`: devuelve el identificador de una hebra.

`pthread_yield()`: cede el procesador voluntariamente.

# Ejemplo en C usando pthread

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

void* quien_soy(void*)
{
    printf("[%lu]: hola\n", pthread_self());
    return NULL; // pthread_exit(NULL);
}

int main()
{
    pthread_t id;
    pthread_create(&id, NULL, quien_soy, NULL);
    pthread_join(id, NULL);
}
```

# Ejemplo en C++ usando `std::thread`

```cpp
#include <iostream>
#include <thread>

void hola() { std::cout << "hola!" << '\n'; }
auto mundo = [] { std::cout << "mundo!" << '\n'; };

int main(int argc, char *argv[])
{
    std::thread h(hola), m(mundo);
    h.join();
    m.join();
}
```

# Ejemplo en C++ con std::jthread

```
std::jthread c([] { cout << "cogito, "; }),
             e([] { cout << "ergo "; }),
             s([](std::stop_token st)
             {
                 cout << "sum!";
                 while (!st.stop_requested())
                 {
                     cout << " ." << flush;
                     std::this_thread::sleep_for(75ms);
                 };
                 cout << '\n';
             });
std::this_thread::sleep_for(1s);
s.request_stop();
```