1. Las ventajas son:

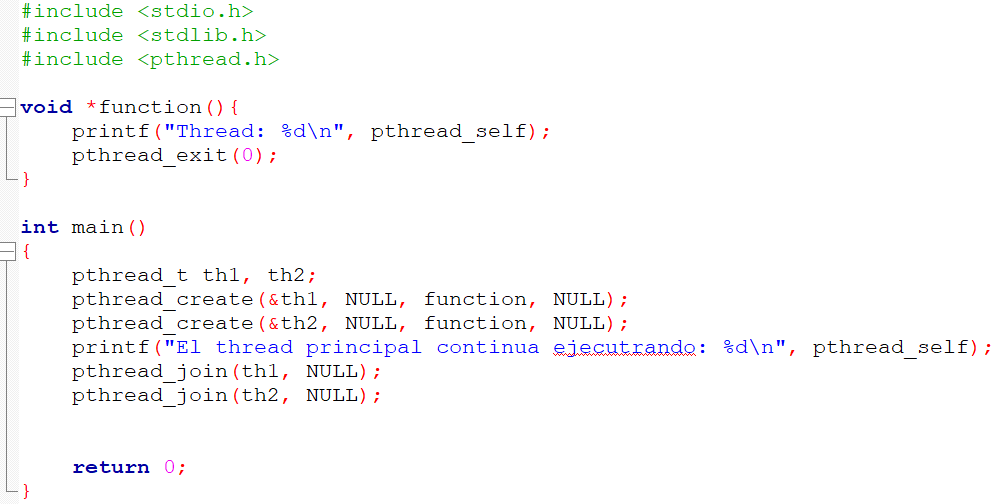
-Es mas veloz crear un nuevo hilo que un nuevo proceso.

-Es mas rápido finalizar un hilo que un proceso,

-Lleva menos tiempo cambiar entre dos hilos dentro del mismo proceso.

-Los hilos mejoran la eficiencia en la comunicación entre distintos programas que se están ejecutando

1. No mejora ya que los threads a nivel usuario no tiene acceso al núcleo (kernel) se maneja todo desde la aplicación a nivel usuario y el sistema operativo los desconoce, solo tiene acceso a un hilo que es el proceso en si, además en caso de bloquearse un hilo se bloquea todo el proceso (desventaja del ULT). Tendría que utilizar threads a nivel núcleo (KLT), que son gestionados directamente por el sistema operativo.
2. Un programa multihilo puede hacer que el procesador ejecute màs de una operación al mimso tiempo, aunque en un hilo se ejecute una operación bloqueante el resto de los hilos pueden seguir operando normalmente, en caso de que esto suceda en un sistema monohilo el procesador quedarìa parado hasta tanto se pueda volver a ejecutar ese proces

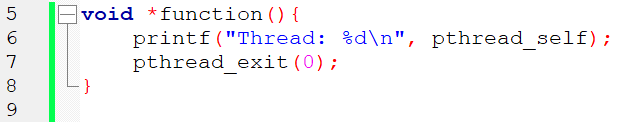


Este código en C crea y ejecuta dos **hilos** usando la biblioteca pthread, además del hilo principal. A continuación te lo explico **línea por línea y parte por parte**:



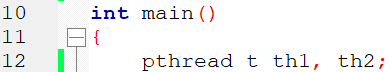
**Inclusión de bibliotecas necesarias**:

* + pthread.h: para trabajar con hilos POSIX (pthreads).



**Definición de la función que ejecutarán los hilos**:

* + void \*funcion(): es la función que será ejecutada por cada hilo.
  + pthread\_self(): devuelve el ID del hilo que está ejecutando la función.
  + printf(...): imprime el ID del hilo.
  + pthread\_exit(0): termina la ejecución del hilo de forma explícita. El 0 indica que finaliza exitosamente.



**Función principal**:

* + pthread\_t th1, th2;: declara dos variables de tipo pthread\_t que almacenarán los identificadores de los hilos que se van a crear.



**Creación de los hilos**:

* + pthread\_create(...): crea un nuevo hilo.
    - &th1 / &th2: dirección donde se guarda el ID del nuevo hilo.
    - NULL: atributos por defecto del hilo.
    - funcion: función que ejecutará el hilo.
    - NULL: argumento que se pasa a la función; en este caso, ninguno.

****

**Mensaje del hilo principal**:

* + Imprime el ID del hilo principal usando pthread\_self().



**Esperar a que los hilos terminen**:

* + pthread\_join(th1, NULL): el hilo principal espera que th1 termine.
  + pthread\_join(th2, NULL): luego espera a que th2 termine.
  + Esto asegura que el programa no termine hasta que ambos hilos hayan finalizado.

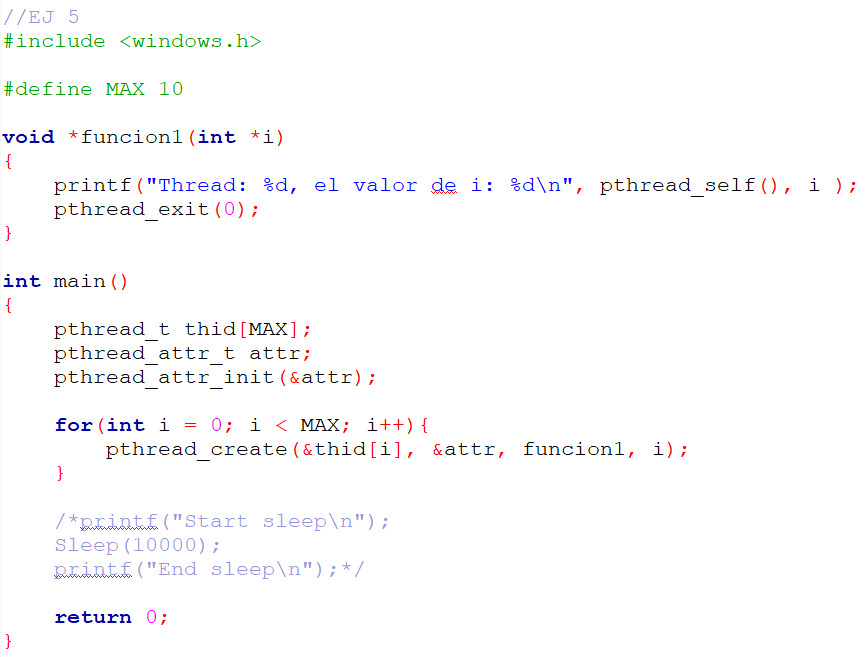
**a) ¿Siempre terminan todos los threads?**

**Sí, siempre terminan todos los threads cuando se incluyen las llamadas pthread\_join(...).**

Esto se debe a que el **hilo principal se queda esperando** que los hilos secundarios (th1 y th2) terminen antes de finalizar el programa. Como resultado, tenés una ejecución ordenada y completa.

**b) ¿Qué pasa si se comentan los pthread\_join?**

Si ejecutas el programa varias veces, notarás que **a veces no se imprimen los mensajes de todos los hilos**. Sin pthread\_join, el hilo principal **no espera** a que los hilos secundarios terminen. Esto implica que el **programa puede finalizar antes de que los hilos terminen de ejecutarse**. Como los hilos están asociados al proceso principal, **cuando el proceso termina, se terminan todos sus hilos activos abruptamente**.



### Tuve que agregar el int en el paramaetro de funcion1 porque si no no andaba, junto con inicializar el arreglo de hilos y el conjunto de atributos, lo que hace es crear 10 hilos con el bucle for,

### a) ¿Siempre terminan todos los threads?

**NO necesariamente.**

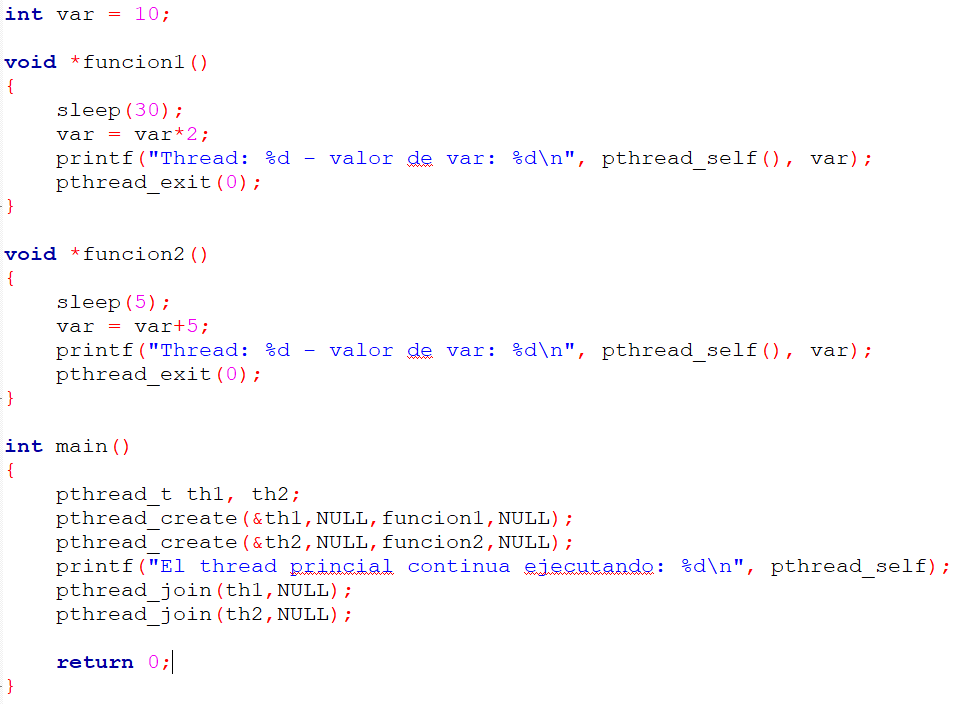
Si **no se usa pthread\_join** ni se deja al main esperando con Sleep(...), es muy probable que **el hilo principal termine antes de que los hilos hijos hayan finalizado**, y eso hace que **se "maten" los hilos restantes** porque el proceso termina.

### b) ¿Qué efecto tiene descomentar Sleep(10000)?

Hace que el hilo principal **espere 10 segundos antes de finalizar**.

Esto **da tiempo suficiente para que todos los hilos hijos terminen su ejecución** y se vea la salida completa.

Actúa como una **sustitución temporal de pthread\_join**, aunque no garantiza sincronización precisa.



Crea **dos threads**:

* 1. Uno que **espera 30 segundos** y luego **duplica el valor de var**.
  2. Otro que **espera 5 segundos** y luego **le suma 5** a var.

Ambos imprimen el valor de var después de modificarlo. El hilo principal imprime su propio ID y espera a que ambos hilos terminen usando pthread\_join.

### a) ¿Siempre terminan todos los threads?

**Sí, siempre**, **si se mantienen las líneas pthread\_join(...)**. Estas llamadas hacen que el hilo principal **espere a que terminen los hilos secundarios** antes de finalizar el programa.

### b) ¿Qué pasa si comento pthread\_join()?

Si comentás estas dos líneas. Entonces el main() puede terminar **antes de que los hilos secundarios completen su ejecución**. Esto puede causar:

* Que **el programa termine sin que los threads lleguen a ejecutar el printf()**.
* Que **veas solo parte de la salida** o incluso **ninguna salida de los threads**.

Esto sucede porque **cuando el hilo principal finaliza, el proceso completo termina**, y los hilos hijos son abortados.

