

## HILOS

### Definición

Son características que permite a una aplicación realizar varias tareas a la vez. Es una tarea que puede ser ejecutada en paralelo con otra tarea.

La mayoría de los SO modernos proporcionan procesos con múltiples secuencias o hilos de control en su interior.

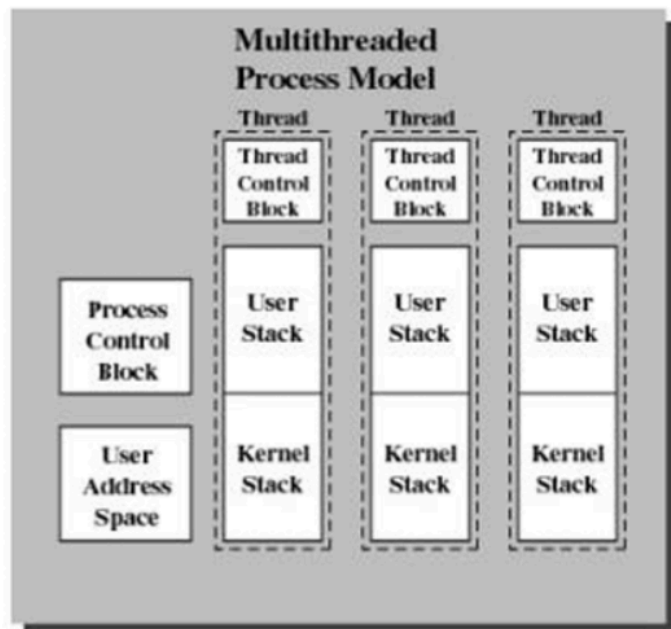
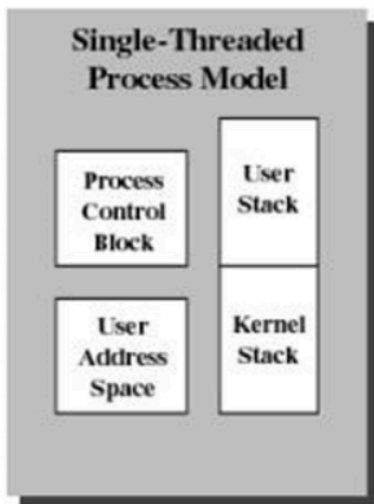
Se lo considera una unidad básica de utilización de la CPU.

Cada hilo posee: identificador del thread, controlador de programa, conjunto de registros y pila.

Comparten con el resto de los hilos del proceso: mapa de memoria, ficheros abiertos y señales, semáforos y temporizadores.

### Los mayores beneficios de los hilos provienen de las consecuencias del rendimiento:

- 1- Lleva mucho menos tiempo crear un nuevo hilo en un proceso existente que crear un proceso totalmente nuevo.
- 2- Lleva menos tiempo finalizar un hilo que un proceso.
- 3- Lleva menos tiempo cambiar entre dos hilos dentro del mismo proceso.
- 4- Los hilos mejoran la eficiencia de la comunicación entre diferentes programas que están ejecutando.



La imagen muestra un **modelo de proceso de sistema operativo**, comparando dos enfoques: el **modelo de proceso monohilo (Single-Threaded Process Model)** y el **modelo de proceso multihilo (Multithreaded Process Model)**. Es un tema común en **sistemas operativos**, relacionado con la **concurrency y paralelismo**.

## ♦ **Modelo Monohilo (Single-Threaded Process Model)**

- Cada proceso tiene:
  - **Process Control Block (PCB):** Contiene información sobre el proceso (estado, registros, prioridades, etc.).
  - **User Address Space:** Memoria del proceso (código, datos, etc.).
  - **User Stack:** Pila para ejecución del código en modo usuario.
  - **Kernel Stack:** Pila utilizada cuando el proceso entra al modo kernel (por ejemplo, en llamadas al sistema).
- Solo hay un **hilo de ejecución (thread)**, lo que significa que el proceso realiza una sola tarea a la vez.

## ♦ **Modelo Multihilo (Multithreaded Process Model)**

- Comparte el mismo:
  - **PCB**
  - **User Address Space**
- Pero tiene **varios hilos (threads)**, cada uno con su:
  - **Thread Control Block (TCB):** Información específica del hilo.
  - **User Stack y Kernel Stack:** Cada hilo tiene su propia pila.
- Permite ejecutar varias tareas simultáneamente dentro del mismo proceso (ej: descargar archivos y actualizar interfaz gráfica al mismo tiempo).

---

### ✅ **Ventajas del modelo multihilo:**

- Mejor uso de CPU en sistemas multiprocesador.
- Mayor eficiencia para tareas concurrentes.
- Compartición de memoria facilita la comunicación entre hilos (más rápido que entre procesos).

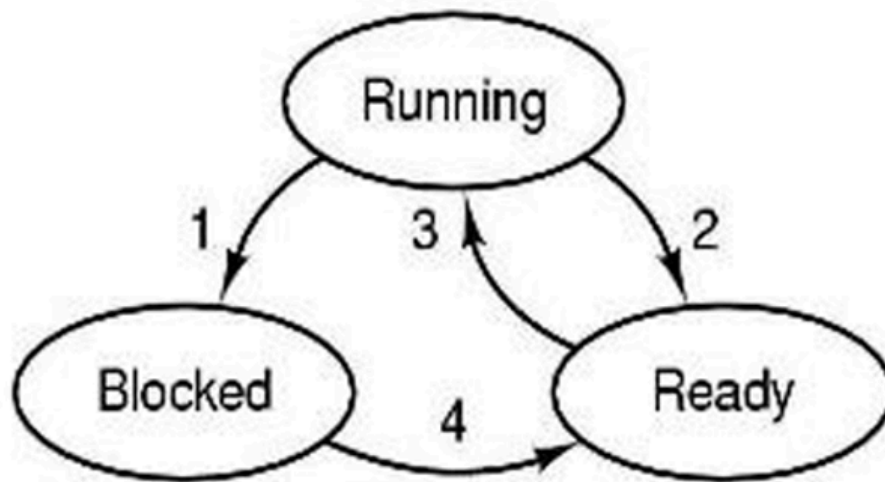
## ESTADOS DE UN HILO

Un thread puede estar en cualquiera de estos 3 estados:

**Ejecución**-> tiene en ese momento la CPU y esta activo.

**Bloqueado**-> espera que algún proceso lo desbloquee.

**Listo**-> esta planificado para ejecutarse y lo hace apenas llegue su turno.






# ¿Qué son los hilos (threads) de nivel usuario?

Los **hilos** son "sub-tareas" dentro de un proceso, que permiten hacer varias cosas a la vez sin tener que crear múltiples procesos. Cuando hablamos de **User-Level Threads (ULTs)** o **hilos a nivel usuario**, nos referimos a hilos que son completamente gestionados por la aplicación, **sin intervención del sistema operativo (núcleo/kernel)**.



---

## En resumen:

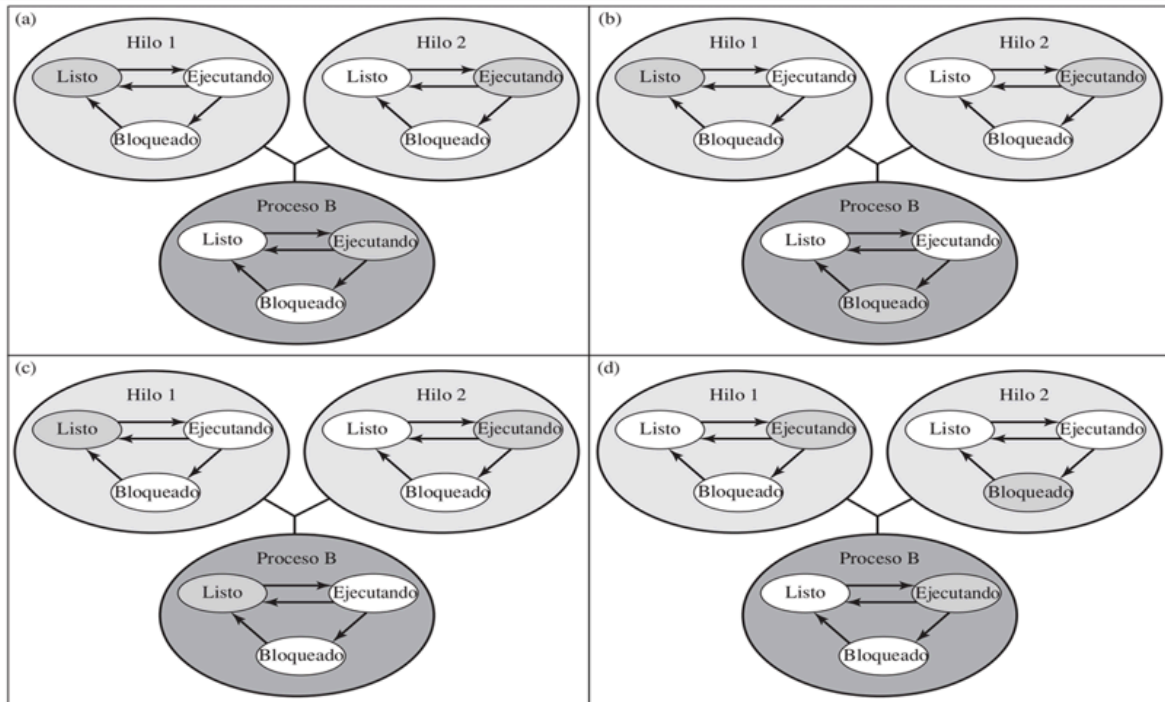
### Ventajas de los ULT:

-  **No necesitan acceso al núcleo** para cambiar de hilo → es rápido.
-  **El programador puede decidir** cómo se planifican los hilos.
-  **Funciona en cualquier sistema operativo**, no requiere soporte especial del núcleo.

### Desventajas:

-  Si un hilo hace una llamada al sistema (por ejemplo, leer un archivo), **se bloquean todos los hilos del proceso**, porque el núcleo no sabe que hay más hilos.
  -  Solo **un hilo puede ejecutarse a la vez por proceso**, porque el sistema operativo no puede asignar múltiples núcleos a los hilos del mismo proceso.
-

## Ejemplo de las relaciones entre los estados de los hilos de nivel usuario y los estados de proceso.



Esa imagen muestra cómo **los hilos pueden estar en distintos estados** (Listo, Ejecutando, Bloqueado), **independientemente del estado del proceso**.

Vamos ejemplo por ejemplo:

### Imagen (a):

- **Hilo 1 bloqueado**, Hilo 2 ejecutando.
- Pero el **proceso está bloqueado** por culpa del Hilo 1 (¡esto es una limitación!).
- Esto **demuestra que un hilo puede bloquear a todo el proceso**.

### Imagen (b):

- Similar a (a), pero ahora Hilo 1 está listo y Hilo 2 está ejecutando.
- Aún así, **Proceso B sigue bloqueado** (por alguna operación que afecta a todos).

### Imagen (c):

- Ambos hilos están ejecutando o listos.

Pero el proceso también está listo: esto **no debería pasar en ULT**, porque si hay una llamada bloqueante, **todos se bloquean**. Esto quizás muestra un ideal deseado.

---

### Imagen (d):

- Todo está ejecutando o listo, incluyendo el proceso. Esta es la mejor situación.
  - **Pero no siempre se puede alcanzar** con ULTs puros por las desventajas mencionadas.
- 

## ¿Por qué es importante esto?

Porque entender cómo se gestionan los hilos a nivel usuario (versus a nivel núcleo) **te permite diseñar aplicaciones más eficientes y evitar bloqueos innecesarios**. Es clave para:

- Sistemas operativos.
- Aplicaciones concurrentes (como navegadores, editores de video, etc.).
- Entender cómo se aprovechan los procesadores multinúcleo.

## ¿Qué son los hilos a nivel núcleo?

A diferencia de los hilos a nivel usuario, aquí **el sistema operativo (el núcleo o "kernel") se encarga de TODO**:

- Gestiona la creación, planificación (cuándo corre cada hilo), y eliminación de hilos.
  - El programador **no tiene que implementar la lógica de hilos**, solo usar una API (por ejemplo, POSIX Threads en C).
-

## ◆ Ventajas de los KLT:

1. **Paralelismo real en múltiples procesadores** 🧠
    - El kernel puede hacer que **varios hilos del mismo proceso se ejecuten al mismo tiempo** en diferentes núcleos de CPU.
  2. **No se bloquea todo el proceso si un hilo se bloquea** 💡
    - Si un hilo llama al sistema (como leer disco), **otro hilo del mismo proceso puede seguir trabajando**.  
→ Esto **no es posible con hilos a nivel usuario** puros.
  3. **El propio núcleo puede usar hilos internamente** 🔄
    - Por ejemplo, para manejar múltiples conexiones de red al mismo tiempo.
- 

## ● Desventaja principal:

- **Cambio de contexto es más costoso** 📦
    - Cuando se cambia de un hilo a otro, el sistema operativo tiene que hacer un **"cambio de modo usuario a modo núcleo"**.
    - Esto consume más tiempo y recursos que un cambio entre hilos de nivel usuario.
- 

## 📌 Ejemplo de la vida real:

Imaginá que:

- En ULT, tenés **un solo jefe (el proceso)** que organiza sus empleados (hilos), pero si el jefe se duerme, **nadie trabaja**.
  - En KLT, tenés un **gerente general (el kernel)** que sabe qué hace cada empleado, y si uno se va al baño, **manda a otro a trabajar en su lugar**.
-

## ¿Cuándo se prefiere usar KLT?

- Cuando **necesitás verdadero paralelismo** (usar varios núcleos).
- Cuando no podés arriesgarte a que **una operación bloquee a todo el programa**.
- En sistemas modernos y robustos como Linux, Windows, macOS, donde **el sistema operativo ya ofrece esta gestión avanzada**.

## ¿Qué es un enfoque combinado ULT/KLT?




Es una técnica donde se **combinan las ventajas** de los hilos a nivel usuario (**ULT - User Level Threads**) y los hilos a nivel núcleo (**KLT - Kernel Level Threads**).

### ¿Cómo funciona?

- Los hilos se crean y gestionan en el espacio de usuario, como los ULT.
- Pero esos múltiples ULT **se asocian con un número menor (o igual) de KLT** que el núcleo del sistema operativo conoce y planifica.
- El cambio de contexto entre hilos aún puede requerir un cambio al modo núcleo, pero no necesariamente para todo.

---

## Ventajas del enfoque combinado:

1. **Mejor paralelismo** 
  - Los hilos pueden ejecutarse en **múltiples procesadores al mismo tiempo**, gracias a los KLT.
2. **Evita bloqueos totales** 
  - Si un hilo hace una llamada bloqueante (por ejemplo, lee de disco), **otros hilos del proceso aún pueden ejecutarse**, algo que no es posible con ULT puros.
3. **Optimización del rendimiento** 
  - Si está bien diseñado, el sistema **combina lo mejor de ambos mundos**:
    - Rapidez y ligereza de los ULT.
    - Paralelismo y tolerancia a bloqueos de los KLT.




# Ejemplo conceptual:

Supongamos que:

- Tenés 5 hilos ULT, pero solo 2 hilos KLT asignados.
- El sistema operativo gestiona esos 2 hilos KLT en los procesadores.
- Dentro de cada KLT, **la biblioteca de hilos en espacio de usuario decide cuál ULT corre en cada momento.**

Es decir, **un mismo KLT puede ejecutar distintos ULT** dependiendo de cómo lo gestione la aplicación.

 **En resumen:**

Característica	ULT	KLT	Combinado
Gestión de hilos	Usuario	Núcleo	Ambos
Cambio rápido entre hilos	✓	✗	Parcialmente ✓
Aprovecha múltiples CPUs	✗	✓	✓
Bloqueo de sistema afecta a todos los hilos	✓	✗	✗
Complejidad	Baja	Alta	Media/Alta

### 3. CREACIÓN DE HILOS CON Pthreads (POSIX threads)

#### Sintaxis general:

C

CopiarEditar

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*func)(void *), void *arg);
```

#### Parámetros explicados:

- **pthread\_t \*thread**  
→ Puntero a una variable donde se guarda el **ID del hilo creado**. Sirve como "manejador" del hilo.  
Ejemplo: `pthread_t hilo1;`
- **const pthread\_attr\_t \*attr**  
→ Permite especificar atributos especiales del hilo, como tamaño de pila, prioridad, etc.  
Si no necesitas nada especial, se pone `NULL` y se usan los atributos por defecto.

**void \*(\*func)(void \*)**

→ Función que se ejecutará en el nuevo hilo. Debe recibir un `void *` como parámetro y devolver un `void *`.

Ejemplo:

C

CopiarEditar

```
void *miFuncion(void *arg) {
    // Código del hilo
    return NULL;
}
```

- **void \*arg**  
→ Un solo parámetro (opcional) que se le pasa a la función `func`. Si no se pasa nada, se usa `NULL`.

#### Valor de retorno:


- `0` si se creó correctamente
  - Otro valor si hubo un error.
-

## 4. ESPERA Y TERMINACIÓN DE HILOS

 `int pthread_join(pthread_t thread, void **value);`

Sirve para esperar a que un hilo específico termine su ejecución.


- `pthread_t thread`: identificador del hilo que queremos esperar.
- `void **value`: si no es `NULL`, guarda el valor que devolvió el hilo al terminar (`pthread_exit`).

 Es **bloqueante**: el hilo actual se detiene hasta que el otro finalice.

---

 `void pthread_exit(void *value);`

Finaliza un hilo de forma explícita y opcionalmente **devuelve un valor** (usualmente un puntero).

 No se debe devolver el puntero a una variable local, porque dejará de existir al terminar el hilo.

---

## 5. IDENTIFICACIÓN DE HILOS

 `pthread_t pthread_self(void);`

Devuelve el **identificador del hilo actual**. Muy útil si querés saber quién está ejecutando el código.

---

## 6. ATRIBUTOS DE HILOS

Los atributos (`pthread_attr_t`) permiten configurar el comportamiento del hilo **antes de crearlo**.

---

 `int pthread_attr_init(pthread_attr_t *attr);`

Inicializa una estructura de atributos para usarla.

---

 `int pthread_attr_destroy(pthread_attr_t *attr);`

Libera los recursos asociados a esa estructura una vez que no la necesites más.

---

 `int pthread_attr_setstacksize(pthread_attr_t *attr, int stackSize);`

Permite definir el **tamaño de la pila** (stack) del hilo. Útil si sabés que el hilo va a usar mucha memoria local o muy poca.

---

 `int pthread_attr_getstacksize(pthread_attr_t *attr, int *stackSize);`

Permite **consultar el tamaño actual de la pila** asignada a un hilo.

Ejemplo completo y comentado en C usando **Pthreads**. Este código:

- Crea dos hilos.
- A cada hilo se le pasa un número diferente como parámetro.
- Cada hilo imprime un mensaje, espera 1 segundo y termina.
- El hilo principal espera que ambos terminen con `pthread_join`.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h> // para sleep

// Función que ejecutará cada hilo
void *hilo_funcion(void *arg) {
    int numero = *((int *)arg); // Convertimos el void* a int*
    printf("Hola desde el hilo %d\n", numero);
    sleep(1); // Simulamos trabajo
    printf("Hilo %d finalizado\n", numero);
    pthread_exit(NULL); // Fin del hilo sin valor de retorno
}
```

```
int main() {
    pthread_t hilo1, hilo2;          // Identificadores de hilo
    int arg1 = 1, arg2 = 2;         // Argumentos a pasar a cada hilo

    // Crear el primer hilo
    if (pthread_create(&hilo1, NULL, hilo_funcion, &arg1)) {
        perror("Error al crear el hilo 1");
        return 1;
    }

    // Crear el segundo hilo
    if (pthread_create(&hilo2, NULL, hilo_funcion, &arg2)) {
        perror("Error al crear el hilo 2");
        return 1;
    }
}
```

```
    // Esperar a que ambos hilos terminen
    pthread_join(hilo1, NULL);
    pthread_join(hilo2, NULL);

    printf("Todos los hilos han terminado.\n");
    return 0;
}
```



### Explicación clave:

- `pthread_create(&hilo1, NULL, hilo_funcion, &arg1);`  
→ Crea un hilo y le pasa `&arg1` como argumento.
- `void *hilo_funcion(void *arg)`  
→ Es la función que ejecuta cada hilo, recibe el argumento como `void *` y lo convierte a `int *` para usarlo.
- `pthread_join(hilo1, NULL);`  
→ Espera a que `hilo1` termine antes de seguir.