T5: Multithreading

SISTEMAS OPERATIVOS
GRADO EN INGENIERÍA INFORMÁTICA

Autores: Professors de Sistemes Operatius del
Departamento de Arquitectura de Computadores
(UPC- BarcelonaTech)

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Licencia

«By Universitat Politècnica de Catalunya - BarcelonaTech (UPC), Any 2025»

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es

Indice

- Threads vs procesos
- Librerías de gestión de threads
- Comunicación entre threads: memoria compartida
- Pthreads sobre Linux

THREADS VS PROCESSOS

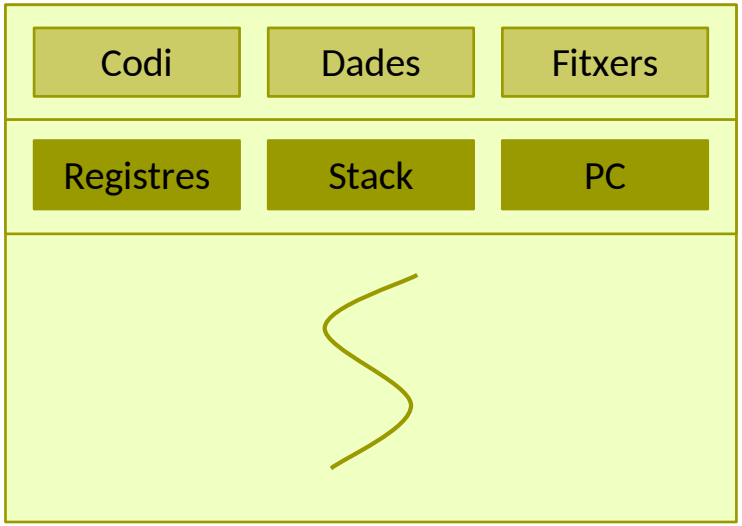
Hilos de ejecución - Threads

- Recordemos: un **proceso** es la representación del SO **de un programa en ejecución**.
- Hasta el momento, hemos visto que cada proceso solamente tiene un "hilo de ejecución", es decir, que solamente está ejecutando una cosa a la vez, pero:

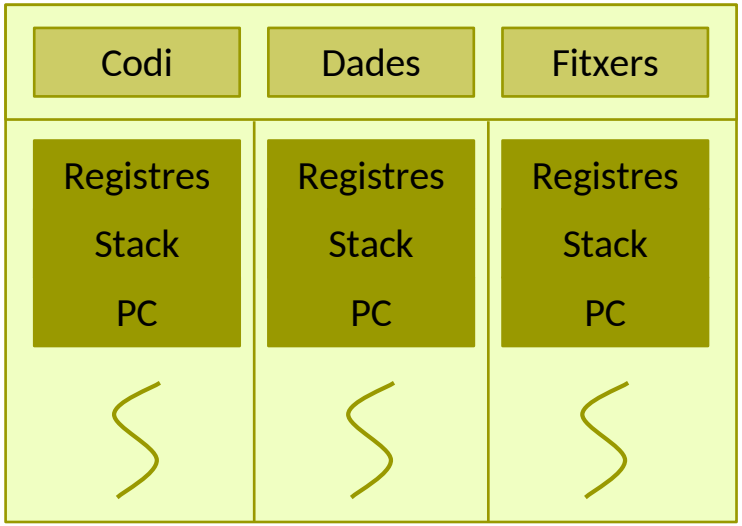
- Entre los recursos que puede gestionar un proceso, están los **hilos de ejecución (threads).**
  - Un thread es una instancia o flujo de ejecución de un proceso, y es la unidad mínima de ejecución y planificación del SO (la unidad mínima a la que se le puede asignar tiempo de CPU)
    - Cada parte del código que se puede ejecutar de forma independiente podría asignarse a un thread
  - Un thread tiene asignado el contexto necesario para representar un **flujo de ejecución de instrucciones**:
    - Identificador (Thread ID: TID)
    - Puntero a la pila (Stack Pointer)
    - Puntero a la siguiente instrucción (Program Counter)
    - Registros (Register File)
    - Variables locales del thread (por ejemplo, errno).

Hilos de ejecución - Threads

- Todos los threads de un mismo proceso comparten los recursos de este:
  - Mismo PCB, y en consecuencia…
  - Misma memoria
  - Mismos dispositivos de entrada/salida, ficheros abiertos, signal handlers, etc..



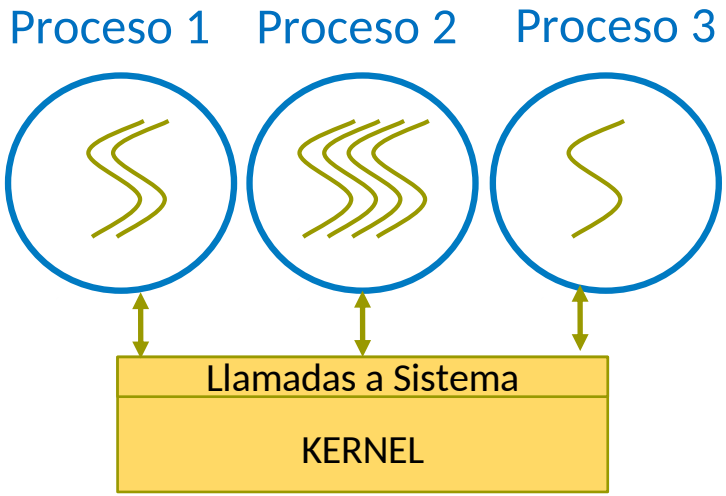
Proceso single thread



Proceso multi-thread

Hilos de ejecución - Threads

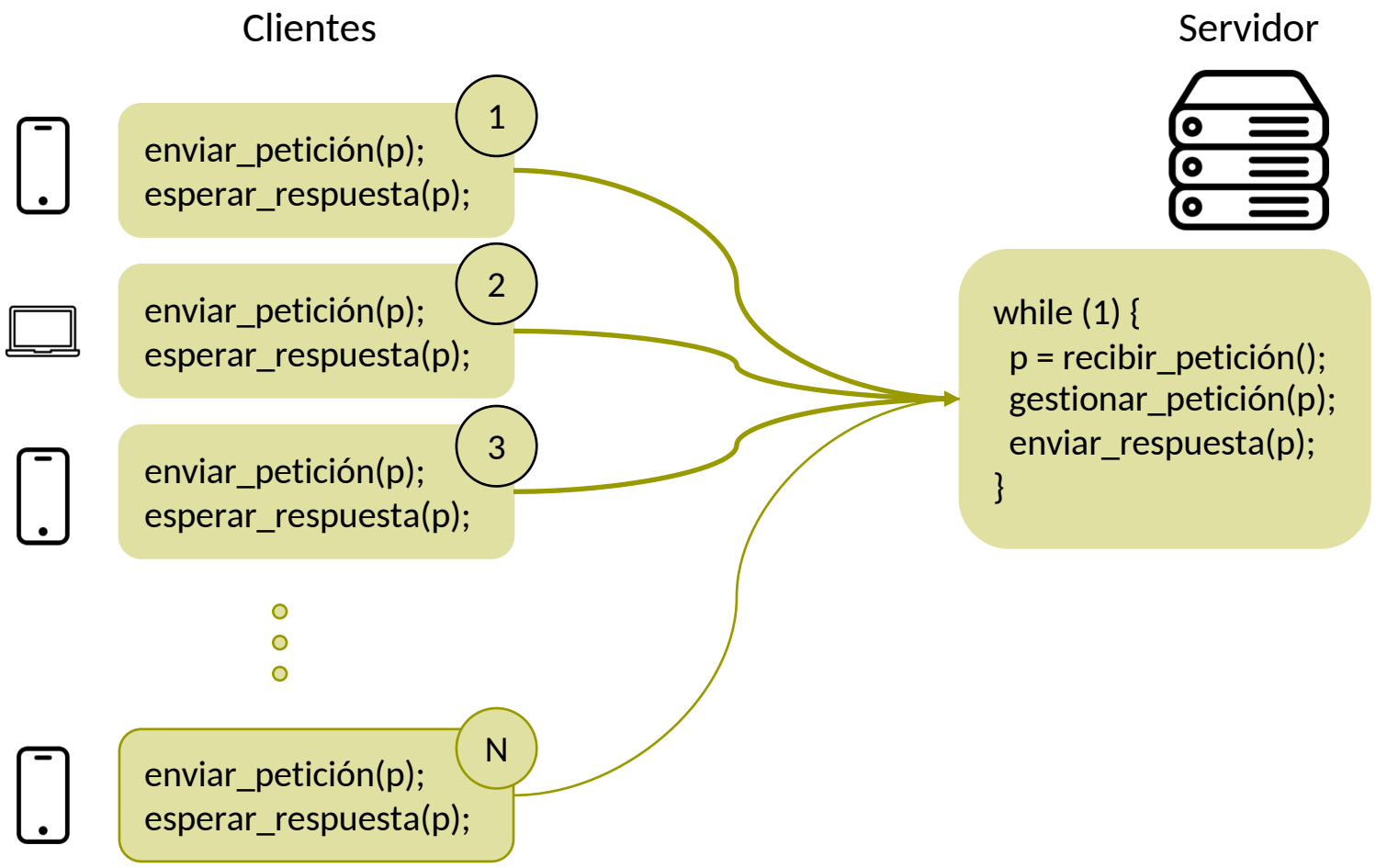
- Al inicio de la ejecución, un proceso tiene **un solo thread**.
- Un proceso puede llegar a tener múltiples threads:
  - Un videojuego actual podría tener > 50 threads. Chrome / Firefox pueden tener > 80 threads.
- La gestión de procesos con varios threads dependerá del soporte del SO
  - User Level Threads vs Kernel Level Threads
- En la figura siguiente tenemos: P1 con 2 threads, P2 con 4 threads y P3 con un solo thread



Hilos de ejecución - Threads

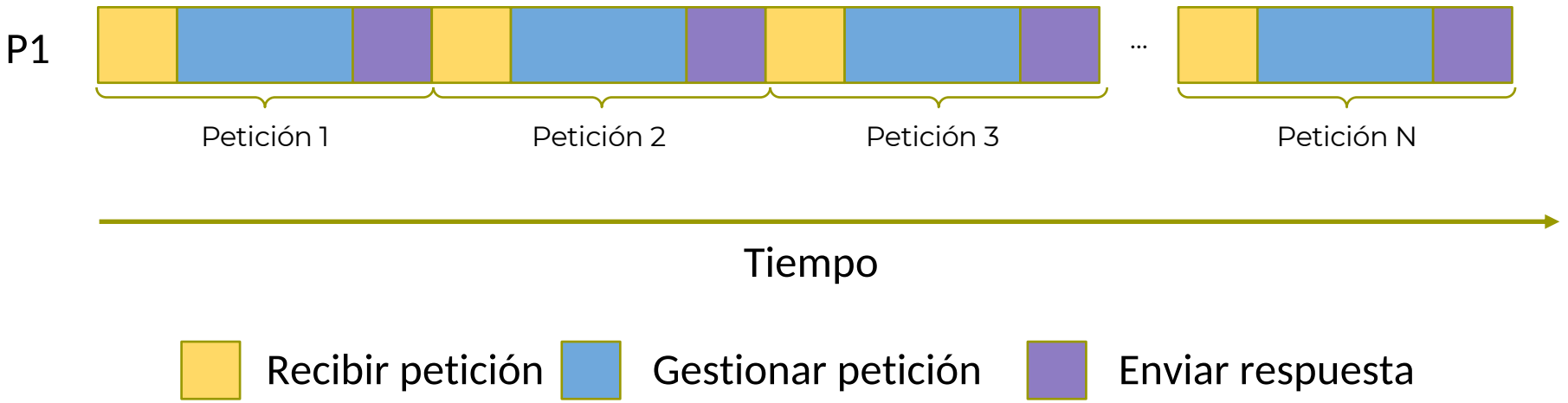
- ¿Para qué sirven?
  - Permiten explotar **paralelismo** en un solo proceso (de código y de recursos)
  - Encapsulan tareas (programación modular)
  - Mejoran la eficiencia de la entrada/salida (permiten delegar operaciones)
  - Pipelining de servicios en servidores (para mantener el tiempo de respuesta)
- Ventajas
  - Los threads son más baratos de crear, finalizar y cambiar de contexto (en el mismo proceso) que entre procesos diferentes
  - Al compartir memoria, los threads pueden comunicarse sin llamadas a sistema
- Inconvenientes
  - Complejos de programar y debugar por la memoria compartida
    - Se deben solventar problemas de sincronización y exclusión mútua: ejecuciones incoherentes, resultados erróneos, deadlocks, etc.

Caso de uso: servidor web



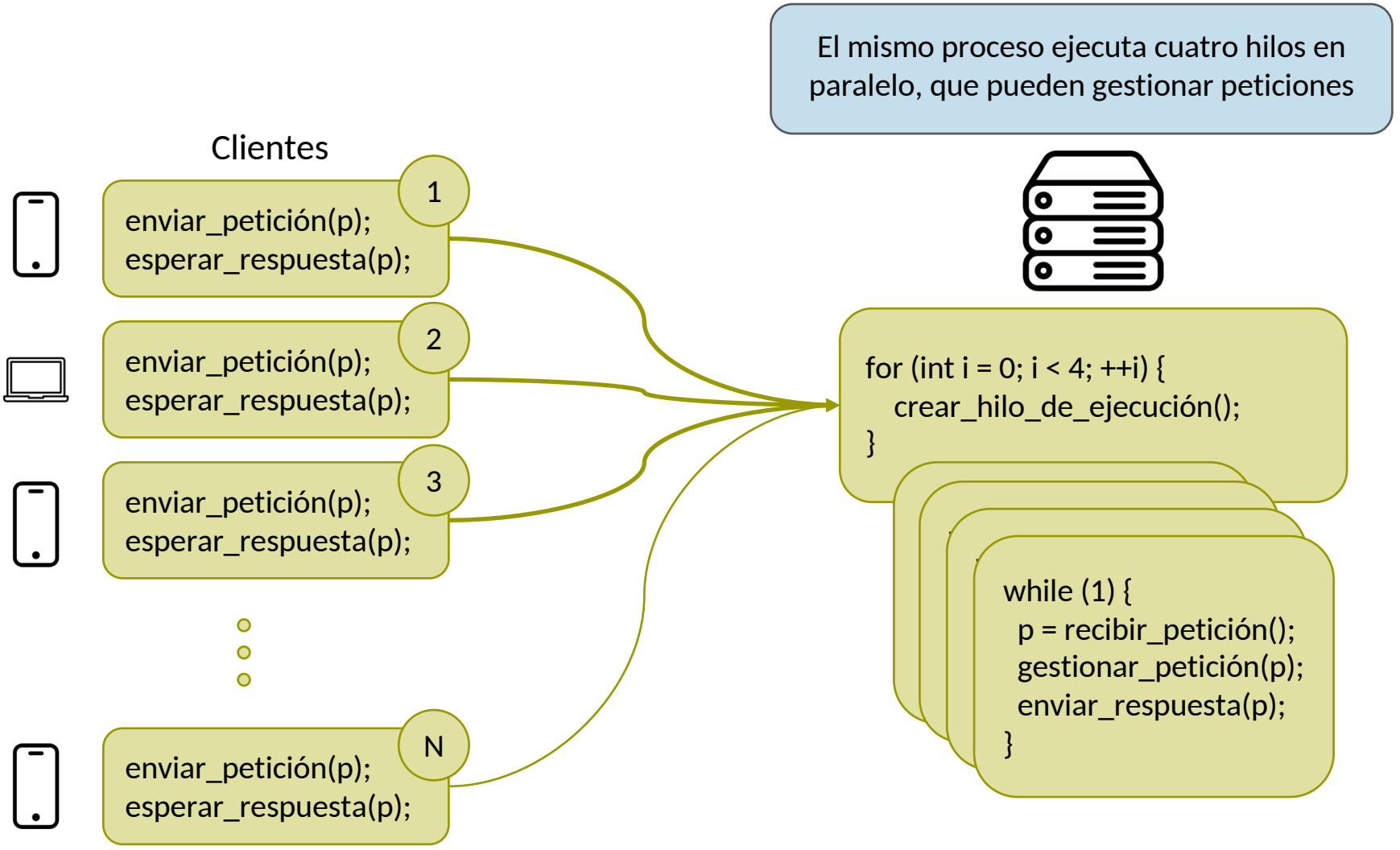
Caso de uso: servidor web

Con un solo proceso, únicamente podemos gestionar una petición a la vez:

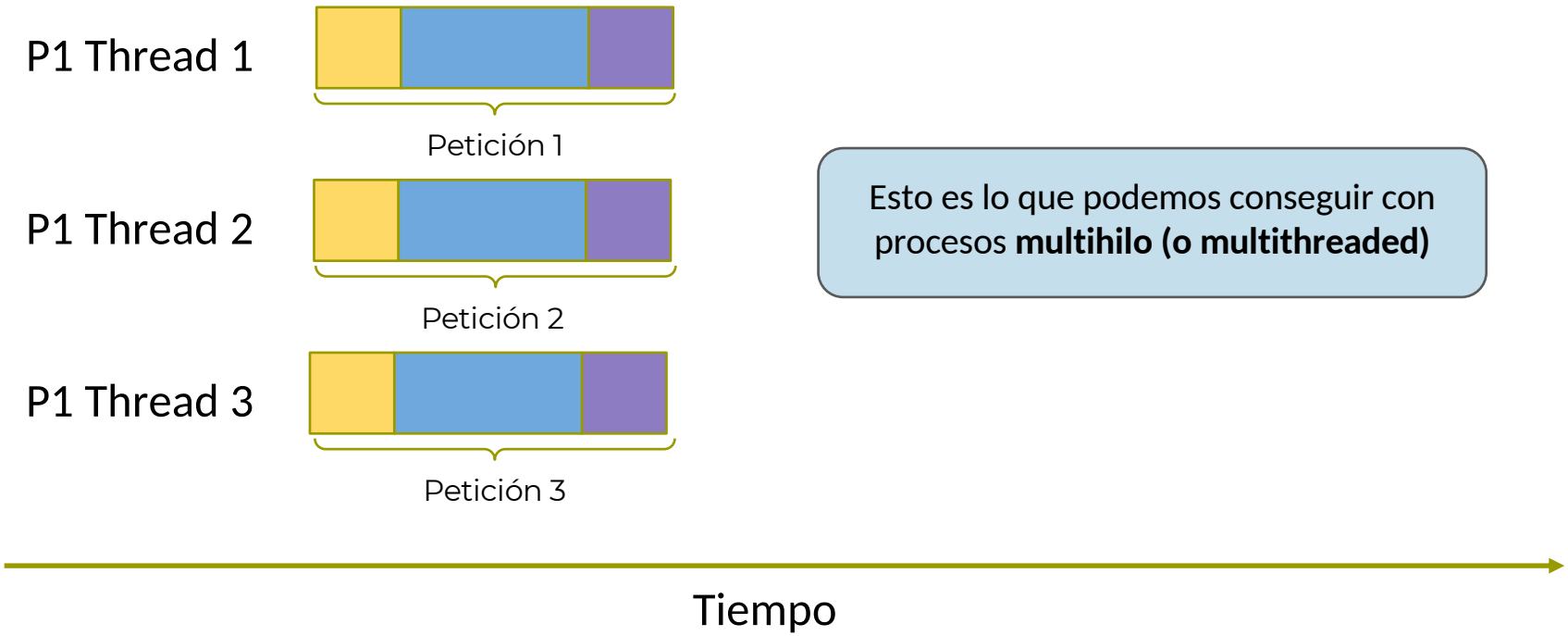


Pero, para hacerlo con múltiples procesos, deberíamos replicar la memoria del servidor, establecer mecanismos de comunicación, etc.

Caso de uso: servidor web



Caso de uso: servidor web



Esto es lo que podemos conseguir con procesos **multihilo (o multithreaded)**

Procesos vs Threads

- La principal diferencia entre procesos y threads es que los segundos **comparten la memoria**:
  - Entre procesos, cada uno tiene una copia privada de los datos globales, y ningún otro proceso puede acceder
  - Entre threads, todos los hilos tienen acceso a la misma memoria (toda la del proceso)
    - Lo que sí tiene cada thread es una pila/stack propio, así como los registros, aunque nada impide que un thread acceda al stack de los otros

GESTIÓN DE THREADS

POSIX Threads

- La librería más utilizada para crear y gestionar hilos de ejecución es la librería de threads de POSIX, también conocidos como **Pthreads**. Nos proporciona interfaces para:
  - Crear y destruir hilos de ejecución
  - Sincronizar los hilos entre ellos
  - Crear regiones de exclusión mutua

* Vale la pena mencionar que a partir de C++11 existe una interfaz hasta cierto punto equivalente en C++, que es **std::thread**. Es habitual que algunos lenguajes de programación tengan sus propias interfaces de threads, aunque casi todos acaben usando POSIX threads por debajo.

Gestión de hilos

Operació	Processos	Threads
Creación	`fork()`	`pthread_create()`
Identificación	`getpid()`	`pthread_self()`
Finalización	`exit()`	`pthread_exit()`
Sincronización final	`waitpid()`	`pthread_join()`

pthread_create

```
man 3 pthread_create
#include <pthread.h>

int pthread_create(pthread_t *th, pthread_attr_t
*attr, void *(*start_routine)(void *), void *arg);
```

- pthread_create() inicia un nuevo thread que ejecutará la función *start_routine* con el argumento *arg*.
  - **th:** parámetro de salida, contiene el identificador del nuevo thread
  - **attr**: opcional, indica atributos adicionales del nuevo thread
  - **start_routine**: función que debe ejecutar el nuevo thread
  - **arg**: argumento que se pasará a la función *start_routine* (como void *)

La función devuelve 0 (si no hay error) o un error.

pthread_self

```
man 3 pthread_self
```

```
#include <pthread.h>

pthread_t pthread_self(void);
```

- pthread_self() devuelve el identificador del thread actual.

Esta función nunca devuelve error.

pthread_exit

man 3 pthread_exit

```
#include <pthread.h>

void pthread_exit(void *retval);
```

- Debe llamarla el thread que quiere finalizar, y tiene el efecto de terminar la ejecución del flujo. El parámetro *retval* será el valor de retorno del pthread, que se puede consultar a través de la llamada *pthread_join*.
- Llamar a pthread_exit() tiene exactamente el mismo efecto que hacer un *return* desde la función *start_routine* de un thread.

Esta rutina nunca retorna, ni devuelve ningún valor.

pthread_join

man 3 pthread_join

```
#include <pthread.h>

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

- pthread_join() bloquea la ejecución del flujo que la llama hasta que *thread* llama a pthread_exit() o finaliza la ejecución de su rutina. Después de la llamada, **retval* contendrá el valor de retorno del thread que haya finalizado su ejecución. Si el parámetro *retval* es NULL, se ignora.

La función devuelve 0 (si no hay error) o un error.

Ejemplo


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

void *rutina(void *arg)
{
    printf("Hola del thread %ld (id: %p)\n", (long) arg, pthread_self());
    return arg;
}

int main()
{
    pthread_t threads[2];
    long ret;
    // Creem el primer thread
    pthread_create(&threads[0], NULL, rutina, (void *) 1);
    printf("Espero...\n");
    pthread_join(threads[0], (void **) &ret);
    printf("Thread finalizado, retorna %ld\n", ret);
    // Creem el segon thread
    pthread_create(&threads[1], NULL, rutina, (void *) 2);
    printf("Espero...\n");
    pthread_join(threads[1], (void **) &ret);
    printf("Thread finalizado, retorna %ld\n", ret);
}
```

https://godbolt.org/z/55YKTbc4n

COMUNICACIÓN ENTRE THREADS: MEMORIA COMPARTIDA

Comunicación entre threads

- Como todos los threads de un proceso comparten memoria, se pueden comunicar a través de ella de una manera muy sencilla
  - Por ejemplo, modificando y leyendo la misma variable
- Sin embargo, hay un **problema habitual** que resulta del uso simultáneo de la memoria:
  - Las **condiciones de carrera** o **race condition**, que se producen cuando dos o más threads modifican la misma posición de memoria sin ver la modificación del otro, provocando un resultado final incorrecto.

Condición de carrera

Thread 1

```
1.if (primero) {
2.     primero = 0;
3.     tarea_1();
4.} else {
5.     tarea_2();
6.}
```

Thread 2

```
1.if (primero) {
2.     primero = 0;
3.     tarea_1();
4.} else {
5.     tarea_2();
6.}
```

Sin más modificaciones, estas operaciones pueden suceder **en cualquier orden**

Condición de carrera

Thread 1

1

```
1.if (primero) {
2.     primero = 0;
3.     tarea_1();
```

Thread 2

0

```
1.if (primero) {
2.} else {
3.     tarea_2();
4.}
```

Dependiendo del orden, podemos obtener una ejecución correcta, como la que se muestra, en que cada thread hace su tarea

Condición de carrera

Thread 1

1

```
1.if (primero) {
2.     primero = 0;
3.     tarea_1();
```

Thread 2

1

```
1.if (primero) {
2.     primero = 0;
3.     tarea_1();
```

Pero en otros casos **NO**

Condición de carrera

Thread 1	Thread 2
1.if (primero) {	1.if (primero) {
2. primero = 0;	2. primero = 0;
3. tarea_1();	3. tarea_1();
4.} else {	4.} else {
5. tarea_2();	5. tarea_2();
6.}	6.}

Para solucionarlo, nos gustaría poder especificar regiones **que no se pueden ejecutar en más de un thread a la vez**

Exclusión mutua

- Uno de los mecanismos más comunes de sincronización entre threads es la exclusión mutua (mutual exclusion). La exclusión mutua sirve para definir regiones de código donde solo un thread puede acceder simultáneamente (regiones críticas).
- Implica dos operaciones:
  - **Lock** (bloqueo): marca el inicio de una región crítica. Si no hay ningún thread ejecutándola, entra el primero que llega. Si está ocupada, el resto espera.
  - **Unlock** (desbloquei): marca el final de una región crítica. Si hay threads esperando, se permite la entrada de uno de ellos.

pthread_mutex

man 3 pthread_mutex_init

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Los mutex implementan el algoritmo de exclusión mutua. Deben inicializarse antes del primer uso con pthread_mutex_init(). Si dos hilos intentan hacer una operación de pthread_mutex_lock() sobre el mismo mutex, solo uno de ellos podrá hacerlo, y el otro permanecerá bloqueado hasta que se llame a pthread_mutex_unlock().

Exclusión mutua

```
pthread_mutex_init(&l, NULL);

1.pthread_mutex_lock(&l);
2.if (primero) {
3.     primero = 0;
4.     pthread_mutex_unlock(&l);
5.     tarea_1();
6.} else {
7.     pthread_mutex_unlock(&l);
8.     tarea_2();
9.}
```

Exclusión mutua

- Cosas importantes a tener en cuenta cuando usamos la exclusión mutua:
  - Usar un mutex evita, hasta cierto punto, la ejecución paralela de nuestro programa. Por tanto, es interesante usarlo poco y en regiones lo mas pequeñas posible.
  - La persona programadora es la responsable de encontrar las regiones críticas de su programa, y los errores que vienen de condiciones de carrera son difíciles de encontrar y a veces incluso de reproducir.
  - Podemos definir diferentes mutex para diferentes regiones críticas (y es recomendable hacerlo).