T5: Multithreading

SISTEMES OPERATIUS
GRAU EN ENGINYERIA INFORMÀTICA

Autors: Professors de Sistemes Operatius del
Departament d'Arquitectura de Computadors
(UPC- BarcelonaTech)

Llicència

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

Índex

- Threads vs processos
- Llibreries de gestió de threads
- Comunicació entre threads: memòria compartida
- Pthreads sobre Linux

THREADS VS PROCESSOS

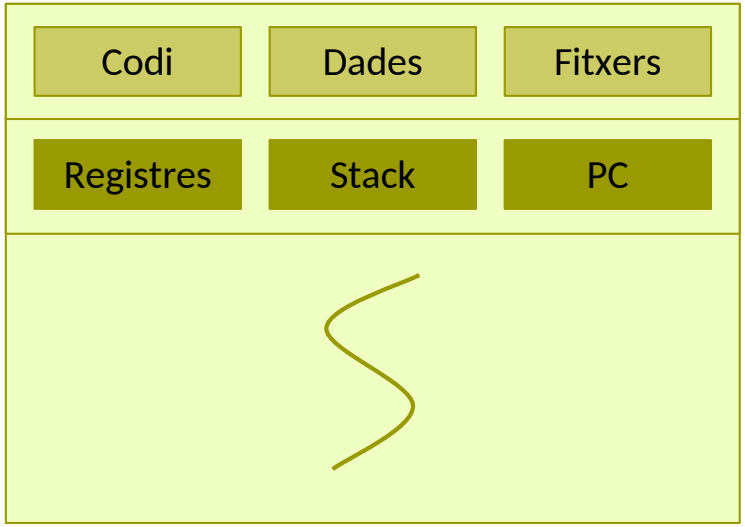
Fils d'execució - Threads

- Recordem: un **procés** és la representació del SO **d'un programa en execució**.
- Fins ara, hem vist que cada procés només té un "*fil d'execució*", és a dir, que només pot estar executant una sola cosa alhora, però:

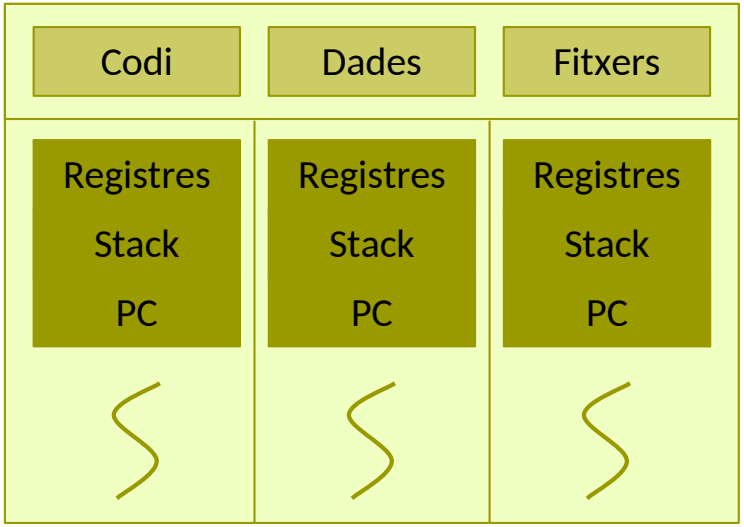
- Entre els recursos que pot gestionar un procés, estan els **fils d'execució (threads).**
  - Un thread és una instància o flux d'execució d'un procés, i és la unitat mínima d'execució i planificació del SO (la mínima unitat a la que se li pot assignar temps de CPU)
    - Cada part del codi que es pot executar de forma independent es podria assignar a un thread
  - Un thread té assignat el context necessari per representar un **flux d'execució d'instruccions:**
    - Identificador (Thread ID: TID)
    - Punter a la pila (Stack Pointer)
    - Punter a la següent instrucció (Program Counter)
    - Registres (Register File)
    - Variables locals del thread (per exemple, errno).

Fils d'execució - Threads

- Tots els threads d'un mateix procés **comparteixen els recursos d'aquest**:
  - Mateix PCB, i per tant…
  - Mateixa memòria
  - Mateixos dispositius d'entrada/sortida, fitxers oberts, signal handlers, etc.



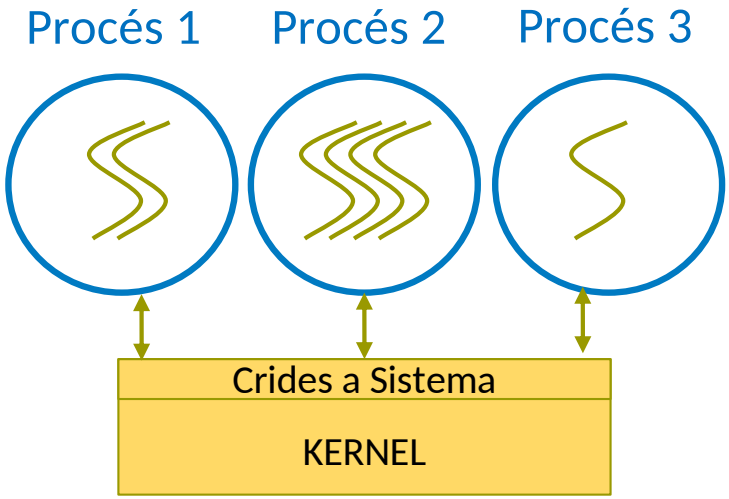
Procés single thread



Procés multi-thread

Fils d'execució - Threads

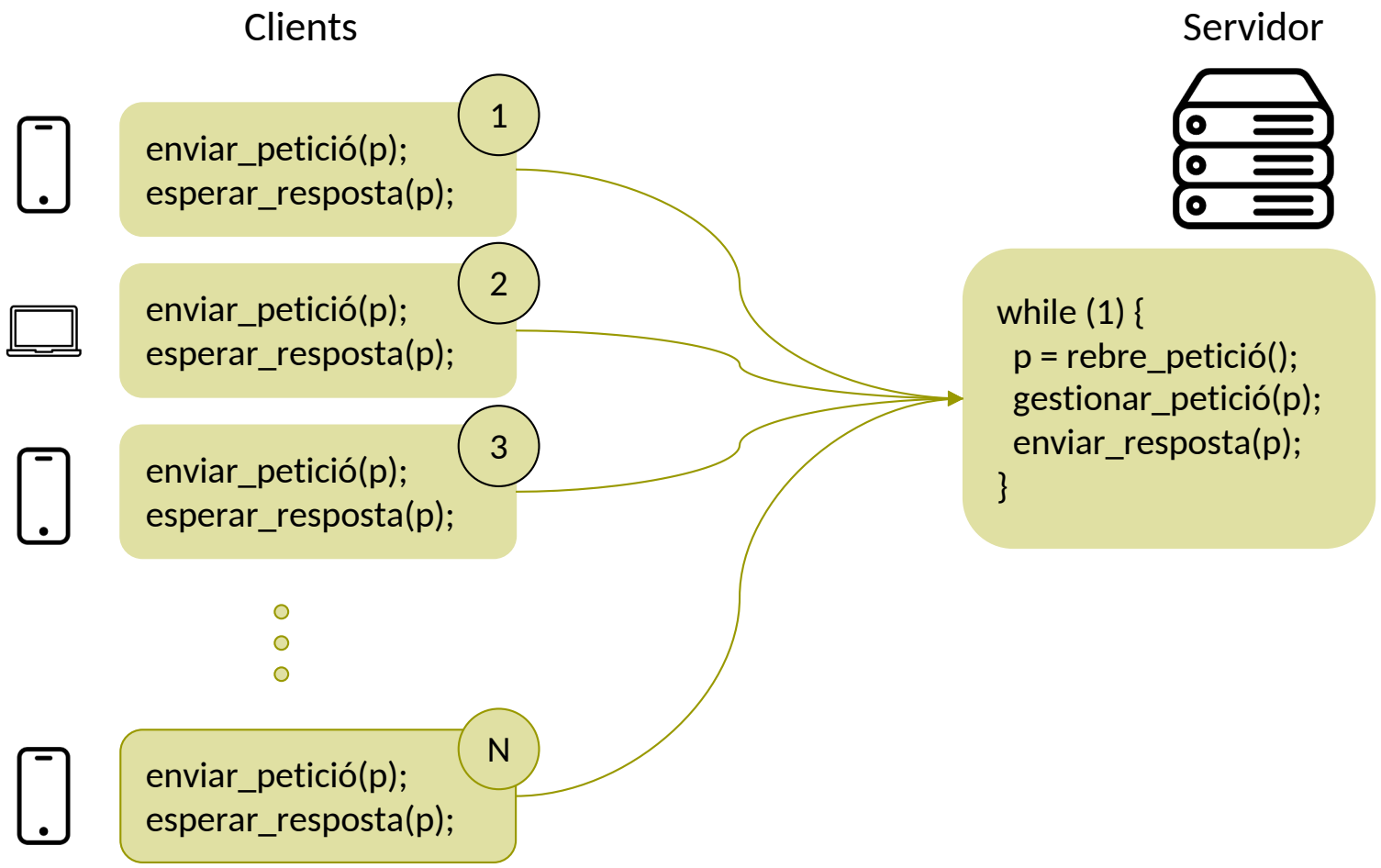
- Al principi de l'execució, un procés té **un sol thread**.
- Un procés pot tenir diversos threads:
  - Un videojoc actual pot tenir > 50 threads. Chrome / Firefox pot tenir > 80 threads.
- La gestió de processos amb diversos threads dependrà del suport del SO
  - **User Level Threads** vs **Kernel Level Threads**
- A la figura següent tenim tres processos diferents: P1 amb 2 threads, P2 amb 4 threads i P3 amb un sol thread



Fils d'execució - Threads

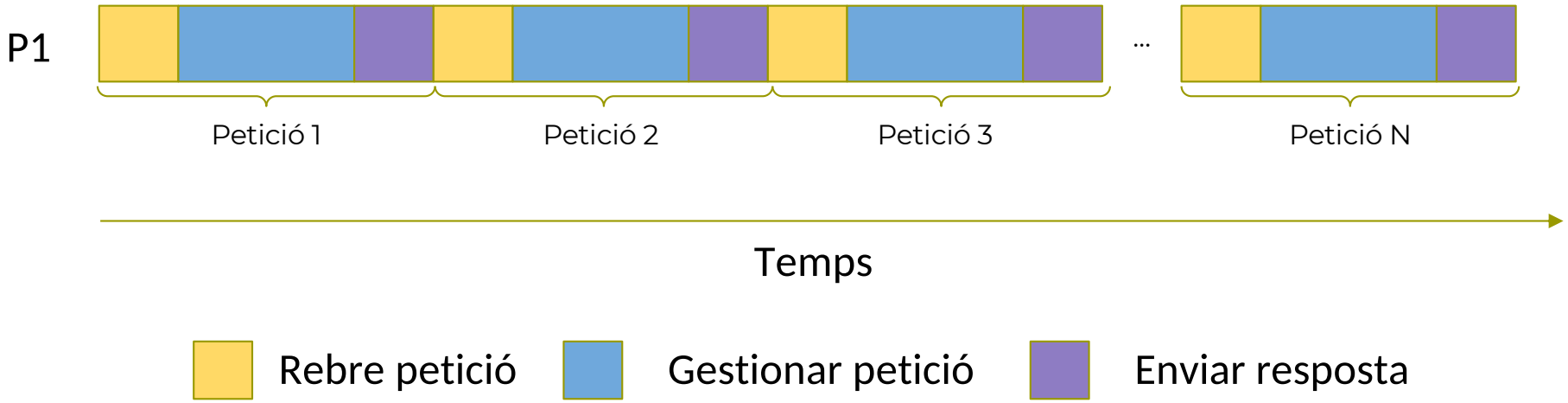
- Per a què serveixen?
  - Permeten explotar **paral·lelisme** en un mateix procés (de codi i de recursos)
  - Encapsulen tasques (programació modular)
  - Eficiència en entrada/sortida (es poden delegar operacions)
  - Pipelining de serveis en servidors (per mantenir temps de resposta)
- Avantatges
  - Els threads són més barats de crear, finalitzar i canviar de context (dintre d'un procés) que els processos
  - Al compartir memòria entre threads es poden comunicar sense usar crides a sistema
- Inconvenients
  - Complexos de programar i debugar per la memòria compartida
    - S'han de solventar problemes de sincronització i exclusió mútua: execucions incoherents, resultats erronis, deadlocks, etc.

Cas d'ús: servidor web



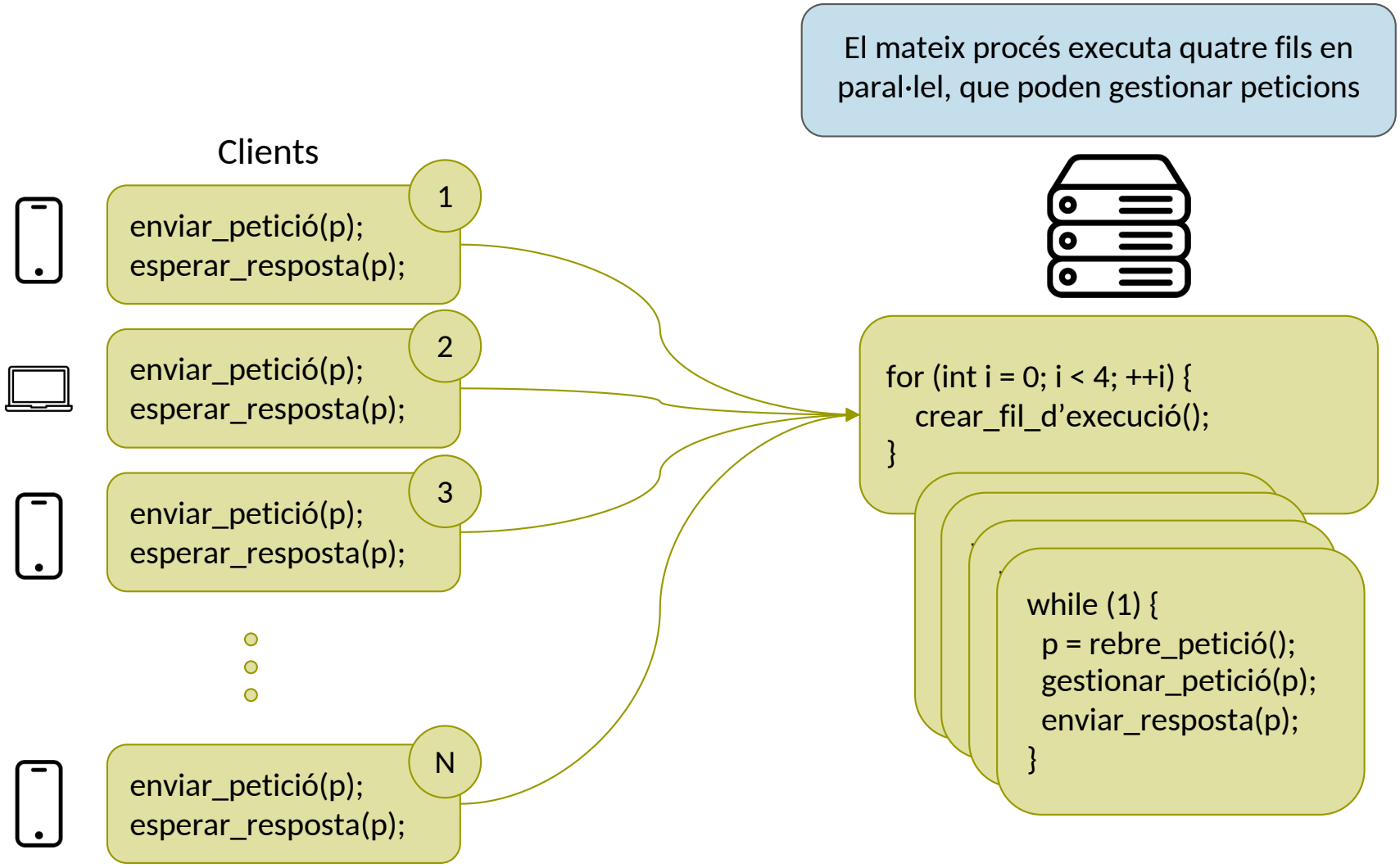
Cas d'ús: servidor web

Si tenim un sol procés, només podem gestionar una petició alhora:

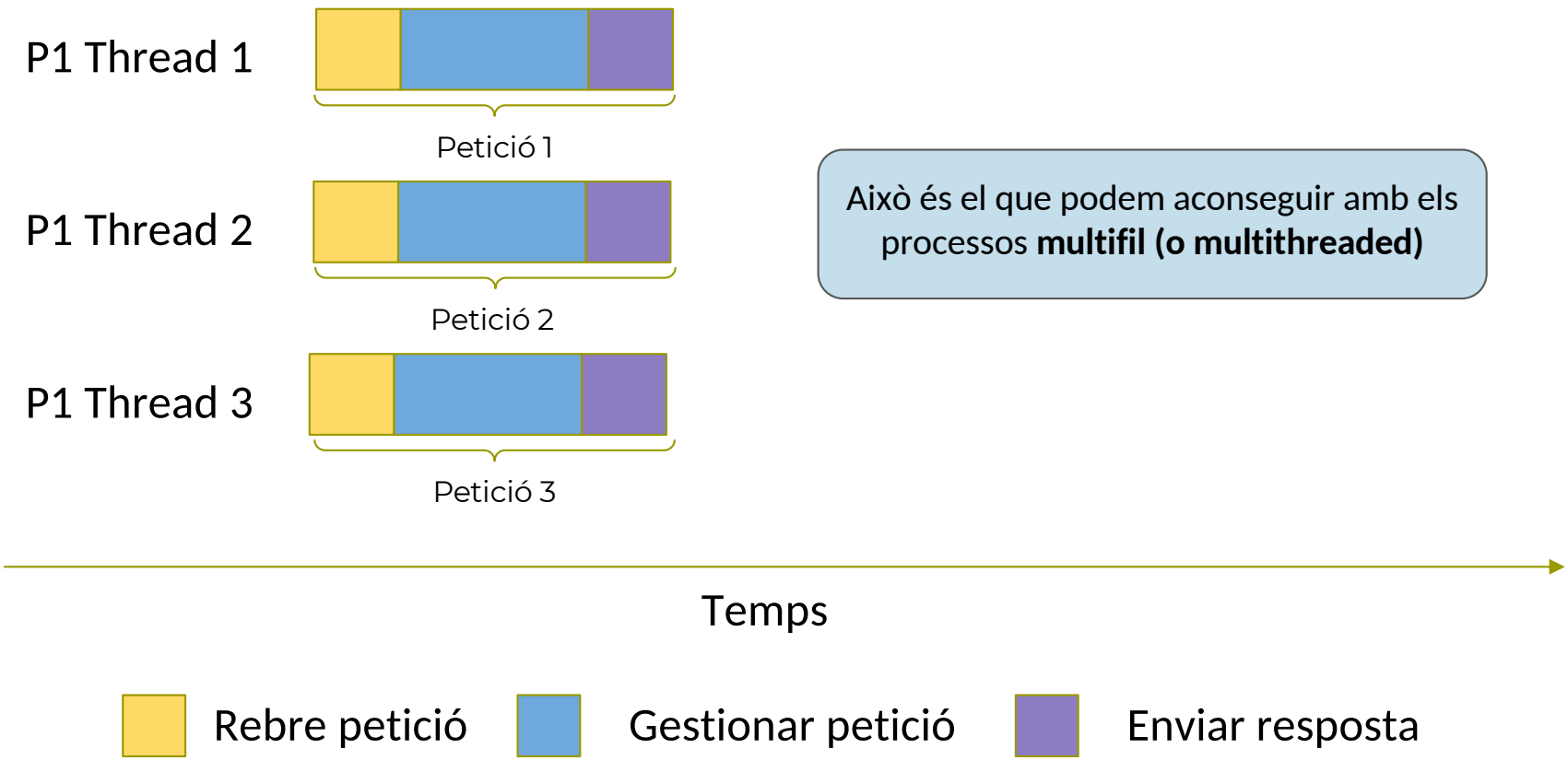


Però, per fer-ho amb múltiples processos, hauríem de replicar la memòria del servidor, establir mecanismes de comunicació, etc.

Cas d'ús: servidor web



Cas d'ús: servidor web



Això és el que podem aconseguir amb els processos **multifil (o multithreaded)**

Processos vs Threads

La diferència principal entre processos i threads és que els segons **comparteixen la memòria**:

- Entre processos, cadascun té una còpia privada de les dades globals i cap altre procés hi pot accedir
- Entre threads, tots els fils tenen accés a la **mateixa** memòria (tota la del procés)
  - El que si té cada thread és un stack/pila propi, així com els registres, però res impedeix que accedeixi a la memòria dels stacks dels altres threads

GESTIÓ DE THREADS

POSIX Threads

La llibreria més utilitzada per crear i gestionar fils d'execució és la llibreria de threads de POSIX, també coneguts com **Pthreads**.

Ens proporciona interfícies per:

- Crear i destruir fils d'execució
- Sincronitzar els fils entre si
- Crear regions d'exclusió mútua

* Val la pena mencionar que a partir de C++11 existeix una interfície fins a cert punt equivalent a C++, que és **std::thread**. És habitual que certs llenguatges de programació tinguin les seves pròpies interfícies de threads, tot i que quasi tots acaben usant POSIX threads per sota

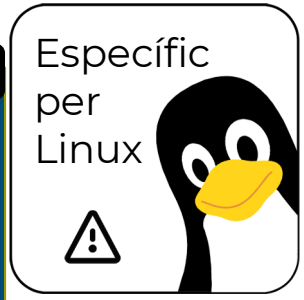
Gestió de fils

Operació	Processos	Threads
Creació	`fork()`	`pthread_create()`
Identificació	`getpid()`	`pthread_self()`
Finalització	`exit()`	`pthread_exit()`
Sincronització 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 nou thread que executarà la funció *start_routine* amb l'argument *arg*.

- **th:** paràmetre de sortida, contindrà l'identificador del nou thread
- **attr**: opcional, indica atributs addicionals del nou thread
- **start_routine**: funció que ha d'executar el nou thread
- **arg**: argument que se li passarà a la funció *start_routine* (com a void *)

La funció retorna 0 (si no hi ha error) o un error.

pthread_self

```
man 3 pthread_self
```

```
#include <pthread.h>

pthread_t pthread_self(void);
```

pthread_self() retorna l'identificador del thread actual.

Aquesta funció mai retorna error.

pthread_exit

```
man 3 pthread_exit
```

```
#include <pthread.h>

void pthread_exit(void *retval);
```

L'ha de cridar el thread que vol finalitzar, i té l'efecte de finalitzar l'execució del flux. El paràmetre *retval* serà el valor de retorn del pthread, i podrà ser consultat a través de la crida *pthread_join*.

Cridar pthread_exit() té exactament el mateix efecte que fer un *return* de la funció *start_routine* d'un thread.

Aquesta rutina no retorna, ni té valor de retorn.

pthread_join

man 3 pthread_join

```
#include <pthread.h>

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

pthread_join() bloqueja l'execució del flux que la crida fins que *thread* crida a pthread_exit() o acaba d'executar la seva rutina. Després de la crida, **retval* contindrà el valor de retorn del thread que ha finalitzat la seva execució. Si el paràmetre *retval* és NULL, s'ignora.

La funció retorna 0 (si no hi ha error) o un error.

Exemple


```
#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

COMUNICACIÓ ENTRE THREADS: MEMÒRIA COMPARTIDA

Comunicació entre threads

Com que tots els threads d'un procés comparteixen la mateixa memòria, s'hi poden comunicar a través d'ella

- Per exemple, modificant i llegint de la mateixa variable

No obstant, hi ha un **problema comú** que resulta de l'ús simultani de la memòria:

- Les **condicions de carrera** o **race condition**, que es produeixen quan dos o més threads modifiquen la mateixa posició de memòria sense veure la modificació de l'altre, provocant un resultat final incorrecte.

Condició de carrera

Thread 1

```
1.if (primer) {
2.     primer = 0;
3.     tasca_1();
4.} else {
5.     tasca_2();
6.}
```

Thread 2

```
1.if (primer) {
2.     primer = 0;
3.     tasca_1();
4.} else {
5.     tasca_2();
6.}
```

Aquestes operacions poden passar en qualsevol ordre, si no fem res

Condició de carrera

Thread 1

1

```
1.if (primer) {
2.     primer = 0;
3.     tasca_1();
```

Thread 2

0

```
1.if (primer) {
2.} else {
3.     tasca_2();
4.}
```

Depèn de l'ordre, podem obtenir una execució correcta, com aquesta, en que cada thread fa la seva tasca

Condició de carrera

Thread 1

```
1.if (primer) {
2.     primer = 0;
3.     tasca_1();
```

1

Thread 2

```
1.if (primer) {
2.     primer = 0;
3.     tasca_1();
```

1

Però en altres casos no!

Condició de carrera

Thread 1	Thread 2
1.if (primer) {	1.if (primer) {
2. primer = 0;	2. primer = 0;
3. tasca_1();	3. tasca_1();
4.} else {	4.} else {
5. tasca_2();	5. tasca_2();
6.}	6.}

Idealment, voldriem poder especificar regions que **no es poden executar en dos threads alhora**

Exclusió mutua

Un dels mecanismes més comuns de sincronització entre threads és el d'**exclusió mutua (mutual exclusion)**. L'exclusió mutua serveix per definir regions de codi on no volem més d'un thread executant-se alhora (regions crítiques) i només permetre que entri un thread a la regió.

Implica dues operacions:

- **Lock** (bloqueig): marca l'inici d'una regió crítica. Si no hi ha cap thread executant-la, entra el primer que hi arriba. Si està ocupada, la resta s'esperen.
- **Unlock** (desbloqueig): marca el final d'una regió crítica. Si hi ha threads esperants per entrar-hi, es permet l'entrada a un d'ells.

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);
```

Els mutexes implementen l'algorisme d'exclusió mutua. S'han d'inicialitzar abans del primer ús amb pthread_mutex_init(). Si dos threads intenten fer una operació de pthread_mutex_lock() sobre el mateix mutex, només un d'ells podrà fer-ho, i l'altre romandrà bloquejat fins que es cridi a pthread_mutex_unlock().

Exclusió mutua

```
pthread_mutex_init(&l, NULL);

1.pthread_mutex_lock(&l);
2.if (primer) {
3.     primer = 0;
4.     pthread_mutex_unlock(&l);
5.     tasca_1();
6.} else {
7.     pthread_mutex_unlock(&l);
8.     tasca_2();
9.}
```

Exclusió mutua

Coses importants a tenir en compte quan usem l'exclusió mutua:

- Usar un mutex evita, fins a cert punt, l'execució paral·lela del nostre programa. Per tant, és interessant usar-la poc i en regions el més petites possible.
- La persona programadora és la responsable de trobar les regions crítiques del seu programa, i els errors que venen de condicions de carrera són difícils de trobar i a vegades inclús de reproduïr.
- Podem definir diferents mutexes per diferents regions crítiques (i és recomanable fer-ho).