Thread in Linux

La libreria pthread, I Kernel threads

Roberto Marino, PhD

Università di Messina
CdL in Informatica — Reti e Sistemi Distribuiti Mod.B

4 maggio 2026



UniMe
1548

Perché i thread?

Con `fork()`

- Copia dell'intero spazio di indirizzamento
- IPC esplicita: pipe, socket, memoria condivisa
- Context switch pesante (TLB flush)
- Adatto per isolamento forte

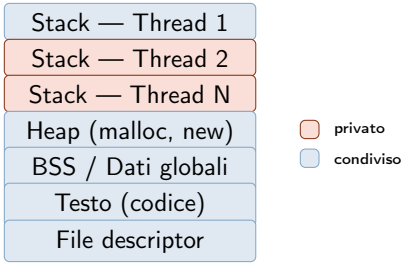
Con i thread

- Condivisione automatica di heap e dati
- Comunicazione tramite variabili globali
- Context switch leggero (stesso spazio VM)
- Adatto per concorrenza fine-grained

Casi d'uso tipici:

- Server concorrenti: un thread per connessione in entrata
- Pipeline di elaborazione: produttore $\rightarrow$ stadi $\rightarrow$ consumatore
- GUI responsive: thread UI separato dal thread di calcolo

Cosa condividono i thread?



Privato per ogni thread:

- Stack (variabili locali, frame di chiamata)
- Registri CPU e program counter
- Maschera dei segnali (`sigmask`)
- `errno` (TLS implicita)

Condiviso tra tutti i thread:

- Heap e variabili globali
- File descriptor aperti
- Handlers dei segnali
- Mappature di memoria (`mmap`)

Conseguenza

Accessi concorrenti a dati condivisi richiedono sincronizzazione → *lezione prossima*.

Creare e terminare un thread

```c
#include <pthread.h>

void *worker(void *arg) {
    int n = *(int *)arg;
    return (void *)(long)n; /* valore di ritorno */
}

int main(void) {
    pthread_t tid;
    int val = 42;
    /* Crea il thread: esegue worker(&val) */
    pthread_create(&tid, NULL, worker, &val);
    void *retval;
    /* Attende terminazione, recupera valore */
    pthread_join(tid, &retval);
    return 0; /* retval == (void*)42 */
}
```

`pthread_create()` — **4 argomenti:**

1. `pthread_t *tid` — identificatore del nuovo thread
2. `pthread_attr_t *` — attributi (NULL = default)
3. `void *(*fn)(void*)` — funzione da eseguire
4. `void *arg` — argomento generico

`pthread_join()` — attende il thread e raccoglie il valore restituito da `return` o `pthread_exit()`.

`pthread_detach()` — il thread è autonomo: le risorse vengono liberate automaticamente, senza bisogno di `join`.

pthread_exit e modalità di uscita

Un thread può terminare in tre modi:

1. **Return dalla funzione thread** — equivalente a chiamare `pthread_exit()` con lo stesso valore.
2. `pthread_exit(retval)` — uscita esplicita; esegue i cleanup handler registrati.
3. **Cancellazione** — un altro thread chiama `pthread_cancel()` (vedi sezione successiva).

Attenzione

Chiamare `exit()` (la exit del processo) da un thread termina *l'intero processo*, non solo il thread corrente.

pthread_detach — rilascio automatico

Un thread deve essere o **joined** o **detached**: altrimenti le sue risorse restano occupate fino alla fine del processo.

Modo 1 — dal thread creante:

```
1 pthread_t tid;
2 pthread_create(&tid, NULL, worker, arg);
3 /* non mi interessa il risultato */
4 pthread_detach(tid);
```

Modo 2 — dal thread stesso:

```
1 void *worker(void *arg) {
2     pthread_detach(pthread_self());
3     /* ... lavoro ... */
4     return NULL; /* risorse liberate qui */
5 }
```

Modo 3 — tramite attributi:

```
1 pthread_attr_t attr;
2 pthread_attr_init(&attr);
3 pthread_attr_setdetachstate(&attr,
4     PTHREAD_CREATE_DETACHED);
5 pthread_create(&tid, &attr, worker, arg);
6 pthread_attr_destroy(&attr);
```

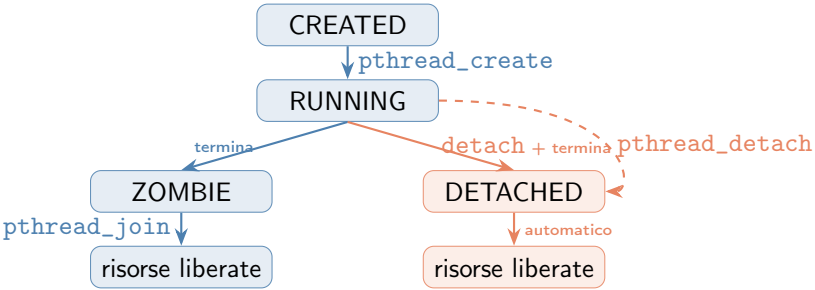
Quando usare detach

Quando il risultato del thread non interessa e non si vuole gestire una lista di TID da raccogliere — tipicamente nei server concorrenti con un thread per connessione.

Attenzione

Un thread detached **non può essere joined**: chiamare `pthread_join` su un thread detached è undefined behavior.

Ciclo di vita completo di un thread



Lo stato **ZOMBIE** non è uno stato di errore: è il periodo tra la terminazione del thread e la chiamata a `pthread_join()`. Senza join o detach, il thread rimane zombie indefinitamente.

Attributi del thread — pthread_attr_t

```c
pthread_attr_t attr;
pthread_attr_init(&attr);

/* Stack personalizzato: 1 MB */
pthread_attr_setstacksize(&attr,
    1024 * 1024);

/* Detached di default */
pthread_attr_setdetachstate(&attr,
    PTHREAD_CREATE_DETACHED);

pthread_create(&tid, &attr, fn, arg);
pthread_attr_destroy(&attr);
```

Attributi principali:

- `stacksize` — dimensione dello stack (default: 8 MB su Linux)
- `stackaddr` — indirizzo dello stack (per allocazione manuale)
- `detachstate` — `JOINABLE` (default) o `DETACHED`
- `schedpolicy` / `schedparam` — priorità di scheduling (richiede privilegi)

Attributi del thread — `pthread_attr_t`

Struttura opaca che raccoglie le opzioni di configurazione di un thread. Si passa come secondo argomento a `pthread_create()`; `NULL` usa i default.

Pattern d'uso:

1. `pthread_attr_init(&attr)` — inizializza con i default
2. impostare gli attributi desiderati
3. `pthread_create(&tid, &attr, fn, arg)`
4. `pthread_attr_destroy(&attr)` — *sempre*

`pthread_attr_destroy()` non distrugge il thread: libera solo le risorse interne della struttura attributi.

Attributi principali:

- `stacksize` — dimensione stack (default 8 MB su Linux)
- `stackaddr` — indirizzo base dello stack (raro, embedded)
- `detachstate` — `JOINABLE` (default) o `DETACHED`
- `schedpolicy` — `SCHED_OTHER`, `SCHED_FIFO`, `SCHED_RR`
- `schedparam` — priorità numerica
- `inheritsched` — eredita dal padre o usa i valori in `attr`
- `scope` — `SYSTEM` (default su Linux) o

Scheduling dei thread

Tre attributi da impostare insieme:

```
pthread_attr_setinheritsched(&attr,
  PTHREAD_EXPLICIT_SCHED);

pthread_attr_setschedpolicy(&attr,
  SCHED_FIFO);

struct sched_param sp =
  { .sched_priority = 10 };
pthread_attr_setschedparam(&attr, &sp);
```

Su thread già in esecuzione: `pthread_setschedparam(tid, SCHED_FIFO, &sp)`

Policy	Tipo	Prio
`SCHED_OTHER`	normale	nice
`SCHED_FIFO`	RT	1–99
`SCHED_RR`	RT	1–99

`SCHED_FIFO`: gira finché non si blocca o cede volontariamente.

`SCHED_RR`: come FIFO ma con timeslice — alla scadenza passa al prossimo thread della stessa priorità.

Privilegi

Richiedono `CAP_SYS_NICE` o un `RLIMIT_RTPRIO` adeguato. In pratica:

Scheduling soft real-time su Linux

Le policy RT si applicano al **singolo thread**, non al kernel. Il CFS continua a girare normalmente per tutti gli altri processi.

Come abilitarlo:

1. Eseguire con `sudo` — il modo più comodo per esperimenti
2. Oppure aggiungere a `/etc/security/limits.conf`:

   ```
   utente  -  rtprio  99
   ```

   e rifare il login

Attenzione su desktop

Un thread `SCHED_FIFO` in un loop senza blocchi può rendere il sistema non responsivo: nulla a priorità inferiore, incluso il compositor grafico, lo preempta.

Consiglio pratico

Usare `SCHED_RR` con priorità basse (1–5): il timeslice impedisce che un loop stretto blocchi tutto. Il 5% del tempo CPU è comunque riservato ai task non-RT da `sched_rt_runtime_us`.

Thread-Local Storage (TLS)

Problema: le variabili globali sono condivise — ma a volte si vuole una copia *per thread*.

Modo 1 — keyword `_Thread_local` (C11):

```c
/* Ogni thread ha la propria copia */
_Thread_local int request_id = 0;

void *worker(void *arg) {
    request_id = get_next_id();
    /* request_id e' locale a questo thread */
    ...
}
```

Modo 2 — chiavi dinamiche POSIX:

```c
pthread_key_t key;

/* Una volta sola (vedi pthread_once) */
pthread_key_create(&key, free);

/* In ogni thread */
pthread_setspecific(key, malloc(64));
void *p = pthread_getspecific(key);
```

Le chiavi dinamiche sono utili quando il tipo o la dimensione della TLS non sono noti a compile time (es. librerie).

`_Thread_local` — il problema

Variabile globale normale: una sola copia in memoria. Tutti i thread leggono e scrivono lo stesso indirizzo.

Se 3 thread incrementano `globale` 1000 volte ciascuno, il risultato atteso sarebbe 3000 — ma i thread si sovrascrivono a vicenda e il risultato reale è imprevedibile.

Perché non basta una variabile locale?
Una variabile locale vive solo durante l'esecuzione della funzione. Quando la funzione ritorna, sparisce. Non persiste tra una chiamata e la successiva.

Serve qualcosa che sia tutte e tre queste cose insieme:

- visibile ovunque nel codice, come una globale
- privata per ogni thread — gli altri non la vedono
- persistente per tutta la vita del thread

_Thread_local — esempio

```c
_Thread_local int contatore = 0;

void incrementa(void) {
    contatore++;
}

void *worker(void *arg) {
    int id = *(int *)arg;

    incrementa();
    incrementa();
    incrementa();

    printf("Thread %d: contatore = %d\n",
           id, contatore);
    return NULL;
}
int main(void) {
    pthread_t t1, t2, t3;
    int id1 = 1, id2 = 2, id3 = 3;

    pthread_create(&t1, NULL, worker, &id1);
    pthread_create(&t2, NULL, worker, &id2);
    pthread_create(&t3, NULL, worker, &id3);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    return 0;
}
```

Output:

```
Thread 1: contatore = 3
Thread 2: contatore = 3
Thread 3: contatore = 3
```

- Se `contatore` fosse una globale normale i tre thread si sovrascriverebbero a vicenda
- Se fosse una variabile locale dentro `worker`, la funzione `incrementa()` non potrebbe accederci
- `_Thread_local` è la via di mezzo: globale nella **visibilità**, privata nell'**accesso**

Cancellazione dei thread — come funziona

La funzione

`pthread_cancel(tid)` non termina il thread immediatamente. È una **richiesta** — il thread decide quando e se onorarla.

I cancellation point

Per default un thread viene cancellato solo quando raggiunge un **cancellation point** — una funzione di sistema bloccante come `read()`, `write()`, `sleep()`, `pthread_cond_wait()`. In quel momento il thread controlla se ha ricevuto una richiesta di cancellazione e se sì termina.

Questo è il comportamento `PTHREAD_CANCEL_DEFERRED` — il default. Garantisce che il thread non venga interrotto nel mezzo di un'operazione critica, ma solo in un punto prevedibile.

Esiste anche `PTHREAD_CANCEL_ASYNCHRONOUS`: il thread può essere interrotto in qualsiasi momento, anche nel mezzo di un calcolo. È pericoloso e raramente usato — non hai nessuna garanzia su dove il thread si trova quando viene fermato.

Cancellazione dei thread — problema e soluzione

Il problema

```
1 void *worker(void *arg) {
2     char *buf = malloc(4096);
3     read(fd, buf, 4096);  /* cancellation point        */
4                           /* se cancellato qui, buf     */
5                           /* non viene mai liberato     */
6     free(buf);
7     return NULL;
8 }
```

La soluzione: cleanup handler

```
1 void *worker(void *arg) {
2     char *buf = malloc(4096);
3     pthread_cleanup_push(free, buf);
4     read(fd, buf, 4096);  /* se cancellato qui...       */
5     pthread_cleanup_pop(1); /* ...free(buf) viene       */
6     return NULL;            /*    chiamata comunque     */
7 }
```

Cancellazione dei thread

```c
void cleanup(void *arg) {
    free(arg); /* eseguito sempre */
}

void *worker(void *arg) {
    char *buf = malloc(256);
    /* Registra il cleanup handler */
    pthread_cleanup_push(cleanup, buf);

    /* Cancellation point: il thread
       puo' essere fermato qui */
    read(fd, buf, 256);

    pthread_cleanup_pop(1); /* 1 = esegui */
    return NULL;
}

/* Da un altro thread: */
pthread_cancel(tid);
```

Due modalità di cancellazione:

- `DEFERRED` (default) — il thread viene fermato solo ai *cancellation point*: `read`, `write`, `sleep`, `pthread_cond_wait`, ...
- `ASYNCHRONOUS` — interruzione immediata in qualsiasi punto (pericoloso, raramente usato), si setta con pthread$_s$*etcanceltype*()

Cleanup handler:

`pthread_cleanup_push/pop()` garantisce che le risorse vengano liberate anche in caso di cancellazione

Barriere — pthread_barrier_t

Una **barriera** è un punto di sincronizzazione: tutti i thread attendono finché *tutti* sono arrivati.

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

static pthread_barrier_t barrier;

void *worker(void *arg) {
    int id = *(int *)arg;

    printf("Thread %d: fase 1\n", id);

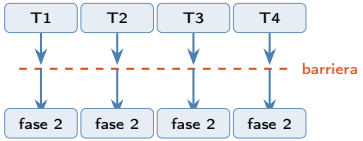
    pthread_barrier_wait(&barrier);  /* aspetto che
          tutti arrivino qui */

    printf("Thread %d: fase 2\n", id);
    return NULL;
}

int main(void) {
    pthread_barrier_init(&barrier, NULL, 3);  /*
         barriera per 3 thread */

    pthread_t t1, t2, t3;
    int id1 = 1, id2 = 2, id3 = 3;

    pthread_create(&t1, NULL, worker, &id1);
    pthread_create(&t2, NULL, worker, &id2);
```



- Uno solo dei thread che raggiungono la barriera riceve `PTHREAD_BARRIER_SERIAL_THREAD` da $pthread_barrier_wait()$
- E' il meccanismo di sincronizzazione più rudimentale tra thread

clone — la primitiva comune

In Linux **processi e thread sono la stessa cosa**: entrambi sono *task* del kernel, creati da `clone()`. La differenza sta nei **flag**.

Flag	Cosa condivide il figlio	Usato da
`CLONE_VM`	Spazio di indirizzamento	thread
`CLONE_SIGHAND`	Handler dei segnali	thread
`CLONE_FILES`	Tabella file descriptor	thread
`CLONE_THREAD`	Thread group (stesso PID)	thread
`CLONE_NEWNET`	Nuovo network namespace	`ip netns`
(nessuno)	Copia completa (fork)	processo

Internamente `pthread_create()` chiama `clone()` con `CLONE_VM | CLONE_SIGHAND | CLONE_FILES | CLONE_THREAD | ...`

TID vs PID e visibilità in `/proc`

- Ogni task ha un **TID** (Thread ID) unico nel sistema: `gettid()`
- I thread dello stesso processo condividono il **PID** del thread principale
- `getpid()` restituisce il PID del thread group — uguale per tutti i thread del processo

Visibilità in `/proc`:

- `/proc/`*`PID`*`/` — il processo nel suo insieme
- `/proc/`*`PID`*`/task/` — una directory per ogni thread
- `/proc/`*`PID`*`/task/`*`TID`*`/status` — stato del singolo thread

Scheduling CFS

Il kernel Linux non distingue processi da thread ai fini dello scheduling: ogni task — sia esso un processo single-thread o un thread di un'applicazione multi-thread — è inserito nella stessa struttura dati dello scheduler.

CFS — Completely Fair Scheduler:

- Mantiene un *virtual runtime* per ogni task: tempo CPU consumato pesato per la priorità (`nice`)
- Seleziona sempre il task con virtual runtime minore (albero red-black)
- Nessuna distinzione tra thread e processi nella coda

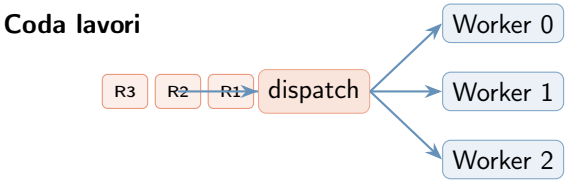
Implicazione pratica:

Un processo con 4 thread riceve — a parità di `nice` — **4 volte** più CPU di un processo single-thread.

Questo è intenzionale: più unità di esecuzione = più lavoro parallelizzabile.

Thread pool

Problema: creare un thread per ogni richiesta è costoso. Un **thread pool** mantiene N thread worker sempre attivi.



- I thread worker dormono in attesa sulla coda (con variabile condizione — prossima lezione)
- Il dispatcher inserisce il lavoro e sveglia un worker libero
- Vantaggio: costo di `pthread_create()` pagato una volta sola all'avvio

Cosa sono i kernel thread

I **kernel thread** sono task che vivono *esclusivamente* in kernel space: non hanno uno spazio di indirizzamento utente e non eseguono mai codice in modalità user.

Sono visibili in `ps aux` tra parentesi quadre:

- `[kworker/0:1]` — code di lavoro asincrone
- `[ksoftirqd/0]` — gestione interrupt software (softirq)
- `[migration/0]` — bilanciamento CPU nei sistemi SMP
- `[kswapd0]` — recupero pagine di memoria
- `[jbd2/sda1-8]` — journaling del filesystem ext4

Perché esistono?

Il kernel ha bisogno di eseguire lavoro in modo asincrono senza legarsi a un processo utente specifico:

- Writeback dei dirty buffer su disco
- Gestione dei softirq (rete, timer)
- RCU (Read-Copy-Update) callbacks
- Migration tra core in sistemi NUMA

Sono schedulati da **CFS** esattamente come i thread utente.

API dei kernel thread (per moduli)

Le API sono disponibili solo nel kernel space (driver o moduli):

```c
#include <linux/kthread.h>
#include <linux/delay.h>

static int my_fn(void *data) {
    while (!kthread_should_stop()) {
        /* lavoro periodico */
        msleep(1000);
    }
    return 0;
}
/* Nel modulo init: */
struct task_struct *kt;
kt = kthread_create(my_fn, NULL, "my_kthread");
if (!IS_ERR(kt))
    wake_up_process(kt);
/* Oppure in un passo solo: */
kt = kthread_run(my_fn, NULL, "my_kthread");
/* Per fermarlo: */
kthread_stop(kt);
```

- `kthread_create()` — crea il task ma non lo avvia
- `wake_up_process()` — mette il task in stato RUNNABLE
- `kthread_run()` — crea e avvia in un passo solo
- `kthread_should_stop()` — controlla se qualcuno ha chiamato `kthread_stop`; usato nel loop per uscire pulito
- `kthread_stop()` — segnala la terminazione e attende

Il kernel thread vive finché non viene fermato esplicitamente o finché il modulo non viene rimosso.

pthread vs kernel thread: confronto

	pthread (user thread)	**kernel thread**
Spazio di esecuzione	user space + kernel	solo kernel space
API di creazione	`pthread_create()`	`kthread_create()`
Accessibile da	qualsiasi programma C	moduli / driver kernel
Visibilità	`/proc/PID/task/TID`	`ps aux` tra `[ ]`
Spazio VM	condiviso con processo	`NULL` (nessuno)
Schedulato da	CFS	CFS
Primitiva comune	`clone()` — flag diversi	

Il denominatore comune

Sia `pthread_create()` che `kthread_create` creano internamente un *task_struct* nel kernel tramite `clone()`. La differenza è nei flag: i pthread condividono `CLONE_VM`, i kernel thread non hanno alcuno spazio utente da condividere.