Processi e thread: i servizi

Pietro Braione

Reti e Sistemi Operativi – Anno accademico 2025-2026

Argomenti

- Concetto di processo
- Operazioni sui processi
- Le API POSIX per le operazioni sui processi
- Comunicazione interprocesso
- Le API POSIX per la comunicazione interprocesso
- Multithreading
- Le API POSIX per il multithreading

Concetto di processo

Concetto di processo (1)

- Un sistema operativo esegue un certo numero di programmi sullo stesso sistema di elaborazione
- Il numero di programmi da eseguire può essere arbitrariamente elevato, di solito molto maggiore del numero di CPU del sistema
- A tale scopo il sistema operativo realizza e mette a disposizione un'astrazione detta **processo**

Concetto di processo (2)

- Un processo è un'entità attiva astratta definita dal sistema operativo allo scopo di eseguire un programma
- Un processo può essere considerato un computer virtuale che esegue un determinato programma:
  - una macchina di Von Neumann...
  - ...con una sua CPU dedicata...
  - ...ed una sua area di memoria dedicata
  - ma con le altre risorse astratte (files, dispositivi di I/O...) condivise con gli altri processi
- Supporremo che l'esecuzione di un processo sia sequenziale (ma presto rilasseremo questa assunzione)

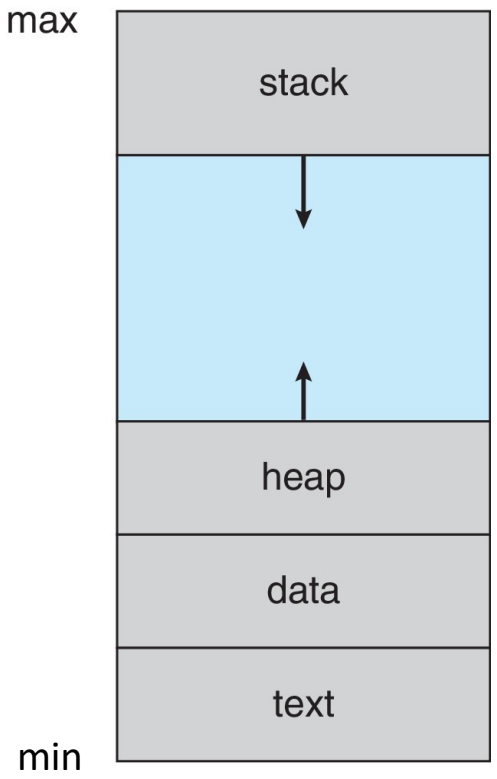
Programmi e processi

- Notare la differenza tra *programma* e *processo*!
  - Un programma è un'entità passiva (un insieme di istruzioni, tipicamente contenuto in un file sorgente o eseguibile)
  - Un processo è un'entità attiva (è un *esecutore di un programma*, o un *programma in esecuzione*)
- Uno stesso programma può dare origine a diversi processi:
  - Diversi utenti eseguono lo stesso programma
  - Uno stesso programma viene eseguito più volte, anche contemporaneamente, dallo stesso utente

Struttura di un processo

- Un processo è composto da diverse parti:
  - Lo stato dei registri del processore che esegue il programma, incluso il program counter
  - Lo stato della **immagine** del processo, ossia della regione di memoria centrale usata dal programma
  - Le risorse del sistema operativo in uso al programma (files, locks…)
  - Più diverse informazioni sullo stato del processo per il sistema operativo
- Notare che processi distinti hanno immagini *distinte*! Due processi operano su zone di memoria centrale separate!
- Le risorse del sistema operativo invece possono essere condivise tra processi (a seconda del tipo di risorsa)

L’immagine di un processo



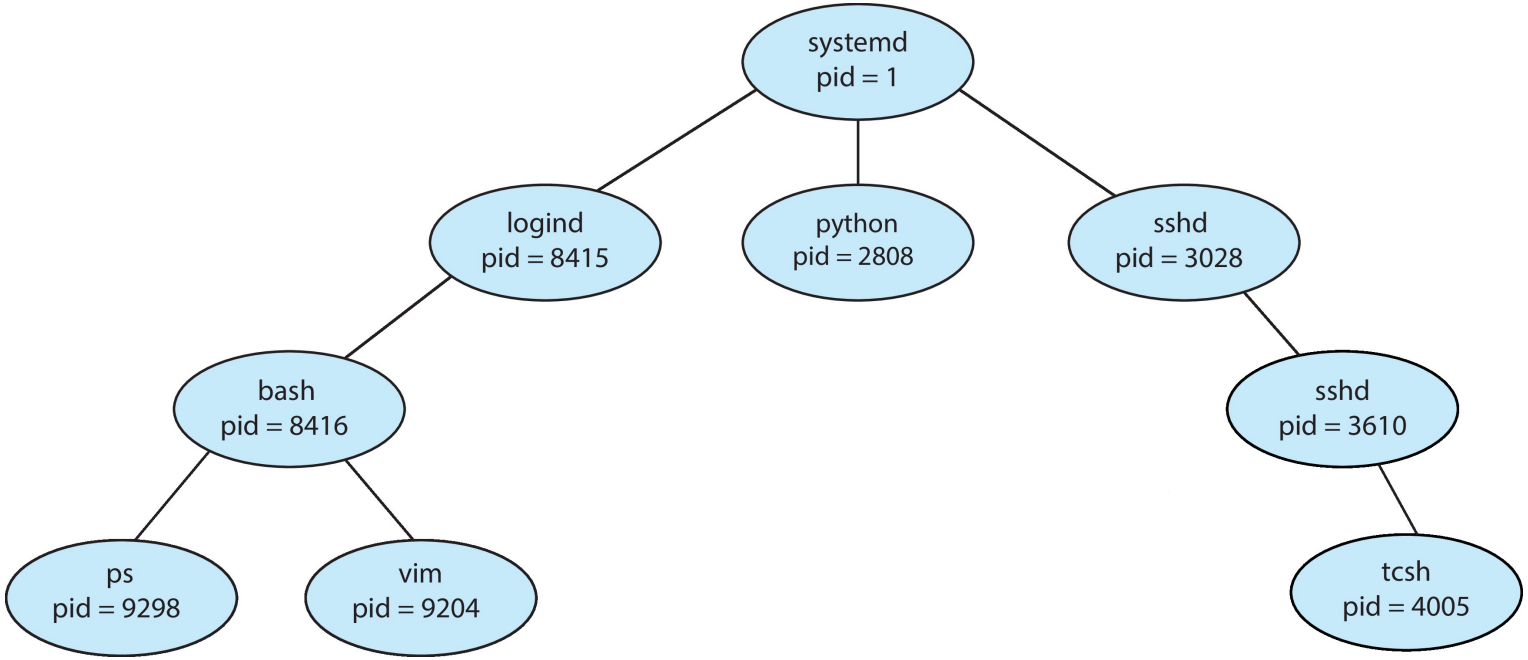
- L’intervallo di indirizzi di memoria min…max di cui è costituita l’immagine di un processo è anche detto **spazio di indirizzamento (address space)** del processo
- L’immagine di un processo di norma contiene:
  - La **text section,** contenente il codice macchina del programma
  - La **data section**, contenente le variabili globali
  - Lo **heap**, contenente la memoria allocata dinamicamente durante l’esecuzione
  - Lo **stack** delle chiamate, contenente parametri, variabili locali e indirizzo di ritorno delle varie procedure che vengono invocate durante l’esecuzione del programma
- Text e data section hanno dimensioni costanti per tutta la vita del processo
- Stack e heap invece crescono / decrescono durante la vita del processo

Operazioni sui processi

Operazioni sui processi

- I sistemi operativi di solito forniscono delle chiamate di sistema con le quali un processo può creare / terminare / manipolare altri processi
  - Un processo (**padre**) può creare altri processi (**figli**)
  - Questi a loro volta possono essere padri di altri processi figli
- Dal momento che solo un processo può creare un altro processo, all'avvio il sistema operativo deve creare dei processi «primordiali» dai quali tutti gli altri processi vengono progressivamente creati
- Per tale motivo i processi sono organizzati in una gerarchia (**albero di processi**) determinata dal rapporto di creazione

Un albero di processi in Linux



Creazione di processi

- La relazione padre/figlio è di norma importante per le politiche di condivisione risorse e di coordinazione tra processi
- Possibili politiche di condivisione di risorse:
  - Padre e figlio condividono tutte le risorse…
  - …o un opportuno sottoinsieme…
  - …o nessuna
- Possibili politiche di creazione spazio di indirizzi:
  - Il figlio è un duplicato del padre (stessa memoria e programma)…
  - …oppure no, e bisogna specificare quale programma deve eseguire il figlio
- Possibili politiche di coordinazione padre/figli:
  - Il padre è sospeso finché i figli non terminano…
  - …oppure eseguono in maniera concorrente

Terminazione di processi

- I processi di regola richiedono esplicitamente la propria terminazione al sistema operativo
- Un processo padre può attendere o meno la terminazione di un figlio
- Un processo padre può forzare la terminazione di un figlio. Possibili ragioni:
  - Il figlio sta usando risorse in eccesso (tempo, memoria…)
  - Le funzionalità del figlio non sono più richieste (ma è meglio terminarlo in maniera «ordinata» tramite IPC)
  - Il padre termina prima che il figlio termini (in alcuni sistemi operativi)
- Riguardo all'ultimo punto, alcuni sistemi operativi non permettono ai processi figli di esistere dopo la terminazione del padre
  - Terminazione in cascata: anche i nipoti, pronipoti… devono essere terminati
  - La terminazione viene iniziata dal sistema operativo

Le API POSIX per le operazioni sui processi

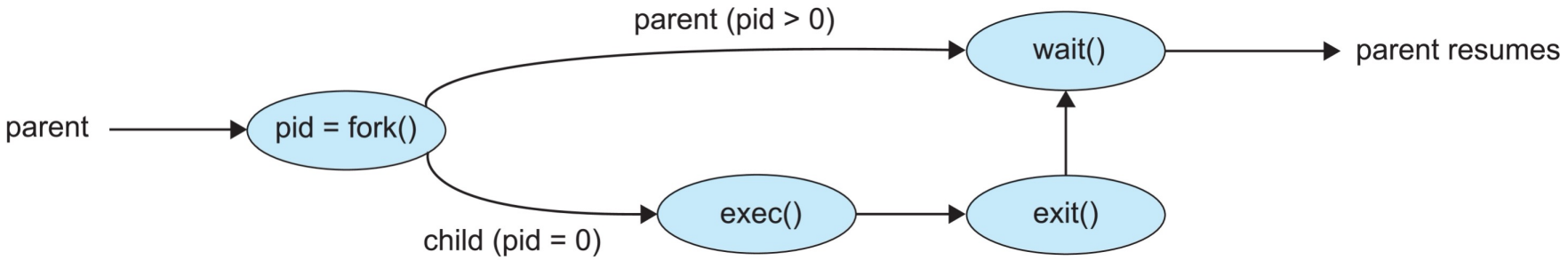
API POSIX per operazioni su processi (1)

- `fork()` crea un nuovo processo figlio; il figlio è un duplicato del padre ed esegue concorrentemente ad esso; ritorna al padre un numero identificatore (PID) del processo figlio, e al figlio il PID 0
- `exec()` sostituisce il programma in esecuzione da un processo con un altro programma, che viene eseguito dall'inizio; viene tipicamente usata dopo una `fork()` dal figlio per iniziare ad eseguire un programma diverso da quello del padre
- `wait()` viene chiamata dal padre per attendere la fine dell'esecuzione di un figlio; ritorna:
  - Il PID del figlio che è terminato
  - Il codice di ritorno del figlio (passato come parametro alla `exit()`)

API POSIX per operazioni su processi (2)

- `exit()` fa terminare il processo che la invoca:
  - Accetta come parametro un codice di ritorno numerico
  - Il sistema operativo elimina il processo e recupera le sue risorse
  - Quindi restituisce al processo padre il codice di ritorno (se ha invocato `wait()`, altrimenti lo memorizza per quando l'invocherà)
  - Viene implicitamente invocata se il processo esce dalla funzione main
- `abort()` fa terminare forzatamente un processo figlio

La tipica sequenza fork-exec



Processi zombie e orfani

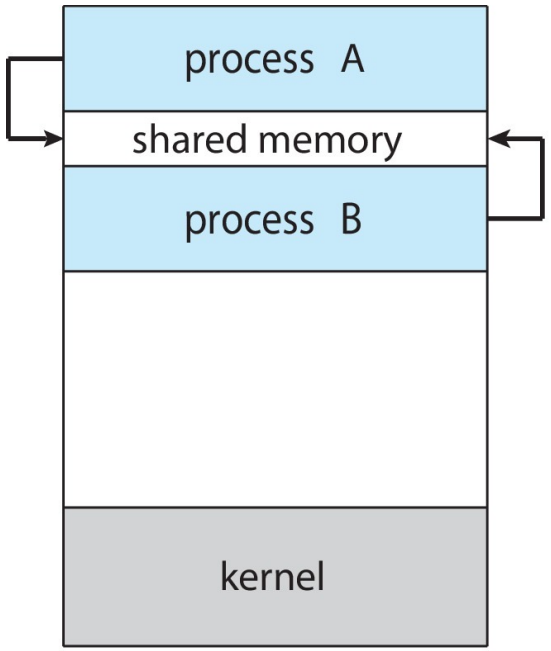
- Se un processo termina ma il suo padre non lo sta aspettando (non ha invocato `wait()`) il processo è detto essere **zombie**: le sue risorse non possono essere completamente deallocate (il padre potrebbe prima o poi invocare `wait()`)
- Se un processo padre termina prima di un suo figlio, non vi è terminazione a cascata: in tal caso, i figli ancora attivi di un processo padre che ha terminato sono detti essere **orfani**

Comunicazione interprocesso

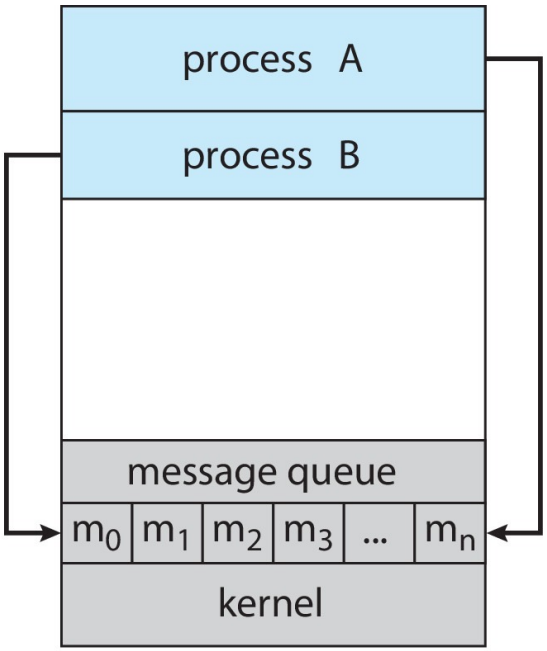
Comunicazione interprocesso

- Più processi possono essere indipendenti o cooperare
- Un processo coopera con uno o più altri processi se il suo comportamento «influenza» o «è influenzato da» il comportamento di questi ultimi
- Possibili motivi per avere più processi cooperanti:
  - Condivisione informazioni
  - Accelerazione computazioni
  - Modularità ed isolamento (come in Chrome)
- Per permettere ai processi di cooperare il sistema operativo deve mettere a disposizione primitive di **comunicazione interprocesso** (IPC)
- Due tipi di primitive:
  - Memoria condivisa
  - Message passing

Modelli di IPC



Memoria condivisa



Message passing

IPC tramite memoria condivisa

- Viene stabilita una zona di memoria condivisa tra i processi che intendono comunicare
- Vedremo come si può implementare quando parleremo di gestione della memoria
- La comunicazione è controllata dai processi che comunicano, non dal sistema operativo
- Un problema importante è permettere ai processi che comunicano tramite memoria condivisa di sincronizzarsi (un processo non deve leggere la memoria condivisa mentre l'altro la sta scrivendo)
- Allo scopo i sistemi operativi mettono a disposizione ulteriori primitive per la sincronizzazione

IPC tramite message passing

- Permettono ai processi *sia* di comunicare *che* di sincronizzarsi
- I processi comunicano tra di loro senza condividere memoria, attraverso la mediazione del sistema operativo
- Questo mette a disposizione:
  - Un'operazione `send(message)` con la quale un processo può inviare un messaggio ad un altro processo
  - Un'operazione `receive(message)` con la quale un processo può (mettersi in attesa fino a) ricevere un messaggio da un altro processo
- Per comunicare due processi devono:
  - Stabilire un **link di comunicazione** tra di loro
  - Scambiarsi messaggi usando `send` e `receive`

Pipe (1)

- Canali di comunicazione tra i processi (una forma di message passing)
- Varianti:
  - Unidirezionale o bidirezionale
  - (se bidirezionale) Half-duplex o full-duplex
  - Relazione tra i processi comunicanti (sono padre-figlio o no)
  - Usabili o meno in rete
- Pipe convenzionali:
  - Unidirezionali
  - Non accessibili al di fuori del processo creatore…
  - …quindi di solito condivise con un processo figlio attraverso una `fork()`
  - In Windows sono chiamate «pipe anonime»

Pipe (2)

- Named pipes:
  - Bidirezionali
  - Esistono anche dopo la terminazione del processo che le ha create
  - Non richiedono una relazione padre-figlio tra i processi che le usano
- In Unix:
  - Half-duplex
  - Solo sulla stessa macchina
  - Solo dati byte-oriented
- In Windows:
  - Full-duplex
  - Anche tra macchine diverse
  - Anche dati message-oriented

Notifiche con callback

- In alcuni sistemi operativi (es. API POSIX e Win32) un processo può notificare un altro processo in maniera da causare l'esecuzione di un blocco di codice («callback»), similmente ad un interrupt
- Nei sistemi Unix-like (POSIX, Linux) tale notifiche vengono dette **segnali**, ed interrompono in maniera asincrona la computazione del processo corrente causando un salto brusco alla callback di gestione, al termine della quale la computazione ritorna al punto di interruzione
- Nelle API Win32 esiste un meccanismo simile, detto **Asynchronous Procedure Call** (APC), che però richiede che il ricevente si metta esplicitamente in uno stato di attesa, e che esponga un servizio che il mittente possa invocare

Le API POSIX per la comunicazione interprocesso

Memoria condivisa in POSIX

- Un processo crea un segmento di memoria condivisa con la funzione `shm_open`:

```
int shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

- Anche usato per aprire un segmento precedentemente creato
- Quindi imposta la dimensione del segmento con la funzione `ftruncate`:

```
ftruncate(shm_fd, 4096);
```

- Infine la funzione `mmap` mappa la memoria condivisa nello spazio di memoria del processo:

```
void *shm_ptr = mmap(0, 4096, PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

- Da questo momento si può usare il puntatore `shm_ptr` ritornato da `mmap` per leggere/scrivere la memoria condivisa

Pipe anonime in POSIX

- Vengono create con la funzione pipe, che ritorna due descrittori, uno per il punto di lettura e uno per il punto di scrittura:

```
int p_fd[2];
int res = pipe(p_fd);
```

- Le funzioni read e write permettono di leggere e scrivere:

```
ssize_t n_wr = write(p_fd[1], "Hello, World!", 14);
char buffer[256];
ssize_t n_rd = read(p_fd[0], buffer, sizeof(buffer) - 1);
```

- È possibile utilizzare la funzione `fdopen` per fare il wrapping di un punto della pipe in un file, ed utilizzare le funzioni C stdio con esso

Named pipes in POSIX

- Le named pipes vengono anche chiamate FIFO nei sistemi POSIX
- Per creare una FIFO si utilizza l'API `mkfifo`:

```
int res = mkfifo("/home/pietro/myfifo", 0640);
```

- La FIFO si utilizza come un normale file:

```
int fd = open("/home/pietro/myfifo", O_RDONLY);
char buffer[256];
ssize_t n_rd = read(fd, buffer, sizeof(buffer) - 1);
```

- Al termine dell'utilizzo, ricordarsi di chiudere:

```
close(fd);
```

- Per eliminare la FIFO, usare l'API `unlink`:

```
unlink("/home/pietro/myfifo");
```

Segnali POSIX

- Per inviare un segnale ad un processo utilizzare l'API `kill`:

```
int ok = kill(1000, SIGTERM); /* terminazione al processo 1000 */
```

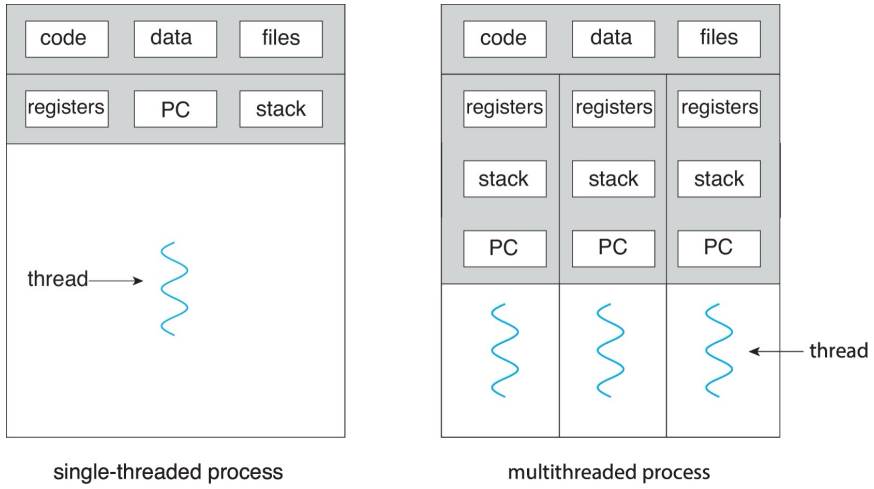
- Per registrare una callback per un determinato segnale esiste un'apposita API `sigaction`

Multithreading

Multithreading

- Fino ad ora abbiamo assunto che un processo abbia un singolo flusso di esecuzione sequenziale (ossia, un singolo processore virtuale)
- Se supponiamo che un processo possa avere *molti* processori virtuali, più istruzioni possono eseguire concorrentemente, e quindi il processo può avere più percorsi (**thread**) di esecuzione concorrenti

Processi single- e multithreaded



- I thread di uno stesso processo condividono la memoria globale (data), la memoria contenente il codice (code) e le risorse ottenute dal sistema operativo (ad esempio i file aperti)
- Ogni thread di uno stesso processo però deve avere un proprio stack, altrimenti le chiamate a subroutine di un thread interferirebbero con quelle di un altro thread concorrente

Librerie di thread

- Le API fornite al programmatore per creare e gestire thread vengono anche chiamate **librerie di thread**
- Librerie più in uso:
  - POSIX pthreads
  - Windows threads

Le API POSIX per il multithreading

Creare un nuovo thread

- All'inizio un processo viene creato con un singolo thread
- Per creare un nuovo thread si utilizza l'API pthread_create, per attendere la fine dell'esecuzione di un thread si utilizza l'API pthread_join:

```
void *thread_code(void *name) { … }
…
pthread_id tid1, tid2;
int ok1 = pthread_create(&tid1, NULL, thread_code, "thread 1");
int ok2 = pthread_create(&tid2, NULL, thread_code, "thread 2");
…
void *ret1, *ret2;
ok1 = pthread_join(tid1, &ret1);
ok2 = pthread_join(tid2, &ret2);
…
```

Aspetti particolari nelle API per il multithreading

- Chiamate di sistema `fork()` ed `exec()`
- Gestione dei segnali
- Cancellazione dei thread
- Dati locali dei thread

Chiamate di sistema `fork()` ed `exec()`

- Una `fork()` dovrebbe duplicare solo il thread chiamante o tutti i thread? Alcuni sistemi operativi Unix-like hanno due diverse `fork()`
- `exec()` invocata da un thread che effetto ha sugli altri thread? Di solito termina tutti i thread del processo precedentemente in esecuzione

Gestione dei segnali

- Quando un processo è single-threaded, un segnale interrompe l'unico thread del processo
- Quando vi sono più thread, quale thread riceve il segnale?
- Possibili soluzioni:
  - Il thread a cui si applica il segnale (ad es. il segnale SIGSEGV viene inviato al thread che ha generato il segmentation fault)
  - Ogni thread del processo
  - Alcuni thread del processo
  - Un thread speciale del processo deputato esclusivamente alla ricezione dei segnali

Cancellazione dei thread

- L'operazione di cancellazione di un thread determina la terminazione prematura del thread
- Può essere invocata da un altro thread
- Due approcci:
  - Cancellazione asincrona: il thread che riceve la cancellazione viene terminato immediatamente
  - Cancellazione differita: un thread che supporta la cancellazione differita deve controllare periodicamente se esiste una richiesta di cancellazione pendente, e in tal caso terminare la propria esecuzione
- Vantaggi:
  - Cancellazione differita: dal momento che un thread controlla il momento della propria cancellazione, può effettuare una terminazione ordinata
  - Cancellazione asincrona: nessuna necessità di controllare periodicamente se ci sono richieste di cancellazione pendenti

Cancellazione nei POSIX pthreads

- Si può attivare/disattivare la cancellazione, ed avere sia cancellazione differita (default) che asincrona
- Se la cancellazione è inattiva, le richieste di cancellazione rimangono in attesa fino a quando (se) è attivata
- In caso di cancellazione differita, questa avviene solo quando l'esecuzione del thread raggiunge un punto di cancellazione (di solito una chiamata di sistema bloccante)
- Il thread può aggiungere un punto di cancellazione controllando l'esistenza di richieste di cancellazione con la funzione `pthread_testcancel()`