

Threads

Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma

Luca Iezzi Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Reti di Calcolatori A.A. 2023/24

Prof.ssa Chiara Petrioli - Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Michele Mastrogiovanni - Dipartimento di Ingegneria Informatica

Ringraziamenti: - **Emanuele Giona** Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Differenze tra processi e thread

2

Processo

Fornisce le risorse per l'esecuzione di un dato programma

- Spazio virtuale degli indirizzi
- Codice eseguibile
- Tabella file descriptor
- Tabella pagine di memoria
- Contesto di sicurezza
- Variabili di ambiente
- Classe di priorità
- PID unico
- *Almeno un thread di esecuzione*

Thread

È l'entità di esecuzione **effettiva** in un processo, che può essere schedata dal sistema operativo

- Spazio virtuale degli indirizzi (globale)
- Codice eseguibile
- Tabella file descriptor
- Signal handlers
- ID user e group
- Thread ID
- Registri (tra cui program counter e stack pointer)
- Priorità
- Variabile `errno`

API POSIX: **Pthread**

Confronto tra processi e thread

3

Processo

- ✗ Più pesante da eseguire
- ~ Memoria non condivisa con altri processi
- ✗ IPC più lenta per via di memoria isolata
Appositi meccanismi sono necessari per l'implementazione
- ✗ Context switch: più costoso
Salvare e caricare process memory e stack comporta alcuni rallentamenti
- ✓ Multi-process può migliorare la gestione di poca memoria: swap su disco facilitato
Processi inattivi dell'applicazione possono essere impostati con bassa priorità

Thread

- ✓ ~ Più leggero da eseguire
Memoria condivisa con il processo genitore e altri thread
- ✓ Comunicazione inter-thread: più rapida
Memoria condivisa di default a livello del processo: è sufficiente la sincronizzazione
- ✓ Context switch: meno costoso
Thread di uno stesso processo già condividono parte della memoria
- ✗ Gestione di poca memoria: nessun meccanismo per migliorarla

Creazione di un nuovo thread

4

Librerie: `sys/types.h`, `pthread.h`

➤ `int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*func)(void*), void *arg)`

Crea un nuovo thread all'interno dello stesso processo, utilizzando gli attributi specificati via `attr`, ed invocando la funzione `func` con `arg` come suo unico argomento; `tid` contiene l'ID del thread creato. `attr` può assumere il valore `NULL`, ovvero degli attributi di default verranno usati. Un thread generalmente termina non appena `func` ritorna.

Ritorna 0 se la chiamata ha successo, altrimenti -1 ed un valore per `errno`; tra i più interessanti ci sono `EAGAIN` (risorse non disponibili) oppure `EPERM` (permessi inadeguati).

➤ `void pthread_exit(void *status)`

Termina l'esecuzione del thread che invoca questa funzione, rendendo disponibile un valore tramite il puntatore `status`. Questa funzione viene implicitamente invocata quando la funzione associata ad un thread invoca `return`.

➤ `int pthread_join(pthread_t tid, void **status)`

Analogamente a `wait` per i processi, attende la terminazione* del thread identificato da `tid`; se `status` è diverso da `NULL`, il valore che il thread restituisce tramite `pthread_exit` verrà salvato nella locazione puntata da `status`.

Ritorna 0 se la chiamata ha successo, altrimenti -1 ed un valore per `errno`.

Attributi dei thread

5

La struttura `pthread_attr_t` usata in `pthread_create` può essere implementata in diversi modi seppur mantenendo la compatibilità con lo standard POSIX.

Per questo motivo:

- Non è buona pratica utilizzare direttamente i membri della struct definita nel sistema in cui si sviluppa
- Le API POSIX offrono **funzioni specifiche** per operare sui determinati attributi dei thread

Attributi dei thread

6

Librerie: `sys/types.h`, `pthread.h`

- `int pthread_attr_init(pthread_attr_t *attr)`
Inizializza gli attributi puntati da `attr` ai valori di default; non è specificato il comportamento in caso di invocazione su `attr` già inizializzato. Uno **stesso oggetto** `attr` può essere usato nella creazione di più thread.
Ritorna 0 se la chiamata ha successo, altrimenti -1
- `int pthread_attr_destroy(pthread_attr_t *attr)`
Distrugge gli attributi puntati da `attr`; POSIX non specifica ulteriormente il comportamento. Ritorna 0 se la chiamata ha successo, altrimenti -1

```
1 pthread_attr_t attrs;  
2 pthread_attr_init(&attrs);  
3 pthread_t tid;  
4 pthread_create(&tid, &attrs, ...);
```



```
1 pthread_t tid;  
2 pthread_create(&tid, NULL, ...);
```

Attributi dei thread

7

Librerie: `sys/types.h`, `pthread.h`

Detach state

I thread possono essere creati in due stati: **detached** o **joinable**. Un thread detached **rilascia immediatamente** tutte le risorse una volta giunto **al termine** dell'esecuzione, senza dover attendere l'invocazione di `pthread_join` su di esso. Eventuali risultati dell'esecuzione dovranno quindi essere **opportunamente salvati** in modo che il resto dell'applicazione ne possa far uso.

Al contrario, un thread joinable tratterrà le risorse fino all'esecuzione di `pthread_join` su di esso o fino al termine del processo che lo ha creato.

- `int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate)`
- `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)`

Valori per detach state: `PTHREAD_CREATE_DETACHED` oppure `PTHREAD_CREATE_JOINABLE`

Attributi dei thread

8

Librerie: `sys/types.h`, `pthread.h`

Guard size

Per default, lo stack di ogni thread viene allocato con un determinato numero di byte usati come **guard size**. Tale spazio non viene effettivamente reso disponibile al thread, bensì permette di implementare una **protezione** da eventuali **stack overflow**.

L'effettiva dimensione dell'area può essere maggiore di quanto specificato; l'utilità principale di gestirla direttamente è quella di poter **disabilitare tale protezione** in caso l'applicazione sia certa che non avvengano stack overflow.

- `int pthread_attr_getguardsize(const pthread_attr_t *attr, size_t *guardsize)`
- `int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize)`

Il valore 0 per `guardsize` permette di disabilitare la protezione da thread stack overflow

Attributi dei thread

9

Librerie: `sys/types.h`, `pthread.h`

Scheduling

Le condizioni di scheduling di un thread possono essere personalizzate tramite le funzioni:

- `int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched)`
Assegnando `PTHREAD_EXPLICIT_SCHED` come valore ad `inheritsched` si userà la configurazione in `attr`
- `int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)`
`policy` può assumere i valori `SCHED_FIFO`, `SCHED_RR`, `SCHED_SPORADIC`, e `SCHED_OTHER` (definiti in `sched.h`)
- `int pthread_attr_setschedparam(pthread_attr_t *attr,`
`const struct sched_param *param)`

La struct `sched_param` (`sched.h`) serve per **configurare** nel dettaglio la **politica di scheduling**: nei casi di `SCHED_FIFO` e

`SCHED_RR`, il solo membro `sched_priority` è richiesto

- `int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope)`
Determina l'ambito di scheduling di un thread: `PTHREAD_SCOPE_SYSTEM` (globale) o `PTHREAD_SCOPE_PROCESS` (locale al processo)

Attributi dei thread

10

Librerie: `sys/types.h`, `pthread.h`

Allocazione dello stack

È possibile controllare direttamente la dimensione dello stack allocato per un nuovo thread, così come l'esatta locazione di memoria in cui questo sarà allocato.

- `int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize)` Imposta la dimensione **minima** dello stack di un nuovo thread al valore fornito tramite `stacksize`; se il valore è **minore di** `PTHREAD_STACK_MIN` o **supera la dimensione massima** supportata dal S.O. viene segnalato un errore `EINVAL`
- `int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr, size_t stacksize)`

Unitamente specifica dimensione minima dello stack e **locazione base di memoria** (`stackaddr`); tutte le pagine di memoria a partire da `stackaddr` ed entro `stacksize` bytes da quell'indirizzo saranno disponibili al thread **sia in lettura che scrittura**. Oltre gli errori relativi a `stacksize`, `EINVAL` rappresentano anche **errori di allineamento** per quanto riguarda `stackaddr`

Altre funzioni di utilità

11

Come la struct `pthread_attr_t`, anche il tipo `pthread_t` è in realtà dipendente dall'**implementazione**, e non può essere utilizzato come visto nel caso di `pid_t` dei processi.

- `pthread_t pthread_self(void)`
Un thread può ottenere il proprio thread ID invocando questa funzione
- `int pthread_equal(pthread_t t1, pthread_t t2)`
Confronta due thread ID `t1` e `t2`: in alcuni casi, `pthread_t` può essere **implementato come una struct** al posto di un semplice `int`. Ritorna 0 se `t1` e `t2` sono **uguali**, altrimenti un valore **diverso da 0**
- `int pthread_kill(pthread_t tid, int sig)`
Invia un segnale `sig` al thread identificato da `tid` (**stesso processo**); il segnale viene gestito nel contesto del thread che lo riceve, ma eventuale **terminazione** o **interruzione** ha effetto sull'**intero processo**

Un thread può essere **creato** con detach state pari a **joinable**, ma questo comportamento può essere **cambiato anche in seguito** alla creazione:

- `int pthread_detach(pthread_t tid)`
Il thread **joinable** identificato da `tid` rilascerà immediatamente le sue risorse al termine dell'esecuzione; non è specificato il comportamento su `tid` già relativi a detached threads. Se il thread è ancora in esecuzione, non ne causa il termine.
Utilità: annullamento di `pthread_join` oppure rendere detached il **thread iniziale** dell'applicazione

2. Thread-specific data

Dati locali ad ogni thread

13

In uno stesso processo, i thread condividono memoria accessibile al processo che li ha generati così come dagli altri thread.

L'uso di funzioni che sfruttano variabili globali o statiche all'interno di un'applicazione multi-thread potrebbe risultare in problemi imprevisti.

```
1 int main(){
2   int counter_n = 5;
3
4   pthread_t tid;
5   pthread_create(&tid, NULL, thread_fn, &counter_n);
6
7   pthread_t tid2;
8   pthread_create(&tid2, NULL, thread_fn, &counter_n);
9
10  pthread_join(tid, NULL);
11  pthread_join(tid2, NULL);
12  return 0;
13 }
```

```
1 void counter_fn(){
2   static int counter = 0;
3   counter++;
4   pthread_t tid = pthread_self();
5   printf("Thread %lu: counter = %d\n", tid, counter);
6 }
7
8 void thread_fn(void *arg){
9   int n = *(int*)arg;
10
11   for(int i=0; i<n; i++){
12     counter_fn();
13     sleep(1);
14   }
15   pthread_exit(0);
16 }
```

Dati locali ad ogni thread

14

```
1 int main(){
2     int counter_n = 5;
3
4     pthread_t tid;
5     pthread_create(&tid, NULL, thread_fn, &counter_n);
6
7     pthread_t tid2;
8     pthread_create(&tid2, NULL, thread_fn, &counter_n);
9
10    pthread_join(tid, NULL);
11    pthread_join(tid2, NULL);
12    return 0;
13 }
```

```
1 void counter_fn(){
2     static int counter = 0;
3     counter++;
4     pthread_t tid = pthread_self();
5     printf("Thread %lu: counter = %d\n", tid, counter);
6 }
7
8 void thread_fn(void *arg){
9     int n = *(int*)arg;
10
11     for(int i=0; i<n; i++){
12         counter_fn();
13         sleep(1);
14     }
15     pthread_exit(0);
16 }
```

Output:

```
Thread 140534716462848: counter = 1
Thread 140534708070144: counter = 2
Thread 140534716462848: counter = 3
Thread 140534708070144: counter = 4
Thread 140534716462848: counter = 5
Thread 140534708070144: counter = 6
Thread 140534716462848: counter = 7
Thread 140534708070144: counter = 8
Thread 140534716462848: counter = 9
Thread 140534708070144: counter = 10
```

La variabile statica `counter` nella funzione `counter_fn` viene **condivisa tra i due thread**: viene quindi aggiornata da entrambi, risultando in un valore del contatore errato.

Dati locali ad ogni thread

15

Per prevenire questa situazione, è possibile utilizzare il thread local storage (TLS), che in POSIX viene denominato **thread-specific data**. Questo meccanismo permette di definire uno **storage chiave-valore**, in cui la **chiave è la stessa** per tutti i thread, mentre il **valore è locale** al thread.

- `int pthread_key_create(pthread_key_t *key, void (*destructor)(void*))`
Crea una **chiave visibile a tutti i thread** in uno stesso processo, possibilmente associando una funzione **destructor** da eseguire in caso al **termine del thread** il valore locale sia **diverso da NULL**
- `int pthread_key_delete(pthread_key_t key)`
Elimina una chiave legata a thread-specific data precedentemente creata da `pthread_key_create`; nessun thread dovrebbe accedere ad una chiave eliminata da questa funzione. **Non viene invocata** l'eventuale funzione **destructor** associata a `key`
- `void* pthread_getspecific(pthread_key_t key)`
Ottiene il valore associato a `key` nel thread che invoca questa funzione
- `int pthread_setspecific(pthread_key_t key, const void *value)`
Imposta il valore associato a `key` nel thread che invoca questa funzione

Dati locali ad ogni thread

16

```
1 static pthread_key_t counterKey; Chiave condivisa  
2                                dai thread  
3 int main(){  
4     int counter_n = 5;  
5     if(-1 == pthread_key_create(&counterKey, free)){  
6         perror("Key create error");  
7         pthread_exit(1);  
8     }  
9  
10    pthread_t tid;  
11    pthread_create(&tid, NULL, thread_fn, &counter_n);  
12  
13    pthread_t tid2;  
14    pthread_create(&tid2, NULL, thread_fn, &counter_n);  
15  
16    pthread_join(tid, NULL);  
17    pthread_join(tid2, NULL);  
18  
19    return 0;  
20 }
```

Creazione chiave condivisa con funzione destructor
`free`

Resto del codice è
invariato

Dati locali ad ogni thread

17

```
1 void counter_fn(){
2   pthread_t tid = pthread_self();
3   int *counterPtr = (int*)pthread_getspecific(counterKey);
4   if(NULL == counterPtr){
5     printf("Thread %lu read NULL\n", tid);
6     counterPtr = calloc(1, sizeof(int));
7     if(NULL == counterPtr){
8       perror("Thread failed on calloc");
9       pthread_exit(1);
10    }
11  }
12
13  *counterPtr += 1;
14  if(-1 == pthread_setspecific(counterKey, counterPtr)){
15    perror("Thread failed on setspecific");
16    pthread_exit(1);
17  }
18
19  printf("Thread %lu: counter = %d\n", tid, *counterPtr);
20 }
```

Allocazione ed
inizializzazione
in caso non sia
presente un
valore

Un thread legge il valore thread-specific tramite la chiave condivisa

Aggiornamento del valore thread-specific associato alla chiave condivisa

Dati locali ad ogni thread

18

```
1 void thread_fn(void *arg){
2     int n = *(int*)arg;
3
4     for(int i=0; i<n; i++){
5         counter_fn();
6         sleep(1);
7     }
8
9     pthread_exit(0);
10 }
```

La funzione del thread rimane invariata

Output:

```
Thread 140369850771200 read NULL
Thread 140369850771200: counter = 1
Thread 140369842378496 read NULL
Thread 140369842378496: counter = 1
Thread 140369850771200: counter = 2
Thread 140369842378496: counter = 2
Thread 140369850771200: counter = 3
Thread 140369842378496: counter = 3
Thread 140369850771200: counter = 4
Thread 140369842378496: counter = 4
Thread 140369850771200: counter = 5
Thread 140369842378496: counter = 5
```

Ogni thread accede ed aggiorna una copia **thread-local**: il contatore mostra correttamente il numero di invocazioni della funzione `counter_fn` per thread

Operazioni di inizializzazione

19

Spesso è necessario assicurarsi che alcune istruzioni, tipicamente di inizializzazione, vengano eseguite esattamente una volta ed alla prima occorrenza durante l'esecuzione di un thread.

Questo tipo di inizializzazioni spesso sfruttano funzioni globali → stesso problema di accesso multiplo

- `int pthread_once(pthread_once_t *once_control, void (*init_routine)(void))`
A **parità** di oggetto `once_control`, la **prima invocazione** di questa funzione da parte di un qualsiasi thread di uno stesso processo **eseguirà** `init_routine`; ogni successiva chiamata a `pthread_once` ignorerà l'esecuzione di `init_routine`.
`once_control` determina se l'inizializzazione è avvenuta o meno, e **deve** essere inizializzato al valore `PTHREAD_ONCE_INIT`.

In realtà questa funzione viene internamente invocata ogni volta che si usano dati thread-specific, ma lato utente è possibile sfruttarla in queste situazioni relative all'inizializzazione.

Esempio di `pthread_once`

20

```
1 static pthread_once_t counterInit = PTHREAD_ONCE_INIT;
2 static pthread_key_t counterKey;
3
4 void init_key(){
5     if(-1 == pthread_key_create(&counterKey, free)){
6         perror("Key create error");
7         pthread_exit(1);
8     }
9 }
```

`counterInit` deve essere dichiarata **globale** e **static**, oltre che inizializzata a `PTHREAD_ONCE_INIT`

Una nuova funzione `init_key` viene definita affinché il **primo thread** che entri in esecuzione garantisca la creazione della chiave condivisa per i dati thread-specific.

La creazione di una chiave da parte di un thread la rende **comunque disponibile agli altri thread** nello stesso processo: in questo modo il codice del **processo** può essere completamente **ignorare dell'esistenza di thread-specific data**, mantenendo però la **garanzia** di un'unica esecuzione della funzione `init_key`.

```
1 void counter_fn(){
2     pthread_once(&counterInit, init_key);
3
4     pthread_t tid = pthread_self();
5     int *counterPtr = (int*)pthread_getspecific(counterKey);
6     if(NULL == counterPtr){
7         printf("Thread %lu read NULL\n", tid);
8         counterPtr = calloc(1, sizeof(int));
9         if(NULL == counterPtr){
10             perror("Thread failed on calloc");
11             pthread_exit(1);
12         }
13     }
14
15     *counterPtr += 1;
16     if(-1 == pthread_setspecific(counterKey, counterPtr)){
17         perror("Thread failed on setspecific");
18         pthread_exit(1);
19     }
20
21     printf("Thread %lu: counter = %d\n", tid, *counterPtr);
22 }
```

3. Sincronizzazione tra thread

Sincronizzazione tra thread

22

Diversamente da quanto visto nei casi di thread-specific data e garanzie sull'unica esecuzione di una funzione, lo sfruttamento di **memoria condivisa** da parte di più thread può essere **esattamente il motivo** per cui si stia sviluppando un'applicazione multi-thread.

Come visto nel contesto di processi multipli infatti, implementare il **controllo dell'accesso a risorse condivise** è fondamentale affinché si possano ottenere i vantaggi dati dall'uso del multi-threading.

Sincronizzazione tra thread

23

In un contesto multi-threading ci sono molteplici meccanismi per implementare la sincronizzazione:

- Semafori Unix
Già visti nel contesto dei processi: il loro funzionamento è del tutto **analogo** nel caso dei thread

Meccanismi specifici per sincronizzazione tra thread:

- *Mutex*
- *Condition variables*
- *Barriera*

Riferimenti

24

- [pthread_create](#), [pthread_exit](#), [pthread_join](#)
- [pthread_attr_destroy/init](#)
- [pthread_attr_get/setdetachstate](#)
- [pthread_attr_get/setguardsize](#)
- [pthread_attr_get/setinheritsched](#), [pthread_attr_get/setschedparam](#),
[pthread_attr_get/setschedpolicy](#), [pthread_attr_get/setscope](#)
- [pthread_attr_get/setstacksize](#), [pthread_attr_get/set_stack](#)
- [pthread_self](#), [pthread_equal](#), [pthread_kill](#), [pthread_detach](#)
- https://www.masterraghu.com/subjects/np/introduction/unix_network_programming_v1.3/ch26lev1sec5.html
- [pthread_key_create](#), [pthread_key_delete](#), [pthread_get/setspecific](#)
- [pthread_once](#)