

This work is licensed under a Creative Commons license



Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0)

You are free to:

Share copy and redistribute the material in any medium or format.

Under the following terms:

Attribution You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial You may not use the material for commercial purposes.

NoDerivatives If you remix, transform, or build upon the material, you may not distribute the modified material.

Multi-threading

... e problemi legati alla concorrenza

Giovanni Lagorio

`giovanni.lagorio@unige.it`

`https://csec.it/people/giovanni_lagorio`

Twitter & GitHub: `zxgio`

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy



`www.zenhack.it`

- 1 Introduzione
- 2 Primitive di sincronizzazione
 - Implementazione
 - Spin-lock
 - Inversione delle priorità
- 3 Funzioni “rientranti”
- 4 Bug tipici
 - Deadlock

Introduzione

- I **thread** permettono di avere **più flussi di esecuzione in un processo**
 - Per Windows i processi sono contenitori di risorse, fra cui i thread
 - In generale, creare un thread significa creare una “CPU virtuale”
 - A livello di kernel: TCB vs PCB
 - La differenza principale fra thread e processi è la condivisione (o meno) dello spazio di indirizzamento
 - Il kernel di Linux non fa differenza: sono “così” schedulabili
 - Il context-switch fra thread costa meno del context-switch fra processi
- *Dal punto di vista logico* ogni thread ha il proprio
 - **stack, ma condivide codice e dati** con altri thread dello stesso processo
 - attenzione: ha il proprio ma tecnicamente può accedere anche agli altri, siccome sono nello stesso spazio di indirizzamento
 - **errno** (nel proprio TLS)

La prima parte di slide è basata su

<http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf> e

<http://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf>

TLS

Thread Local Storage, spazio (logicamente) locale a ogni thread, tipicamente accessibile con un'interfaccia tipo dizionario chiave/valore

In pthread:

- `pthread_key_create` crea una chiave, di tipo `pthread_key_t`
- può essere usata per memorizzare dei `(void *)`
 - `pthread_setspecific`
 - `pthread_getspecific`

ma possiamo **affidarci al compilatore**; in GCC: `__thread`

In Windows, interfaccia analoga (`TlsAlloc`, `TlsSetValue` e `TlsGetValue`) e `__declspec(thread)`

Con i moderni C/C++, **`thread_local`**

- keyword in C++
- macro, definita in `threads.h`, corrispondente a `_Thread_local` in C

Perché usare i thread?

- Sfruttare il **parallelismo**
- Non bloccarsi per fare I/O
 - Non è l'unico modo, ma facilita le cose

Creazione di thread

Consideriamo le API POSIX, `pthread`:

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

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

```
// Compile and link with -pthread.
```

Attenzione, in caso di:

- successo restituisce 0, *altrimenti*
- restituisce direttamente il codice di errore (NON lo scrive in `errno`)

Pagine di manuale

Su Ubuntu dovete installare il package `manpages-posix-dev` per rendere disponibili tutte le pagine di manuale corrispondenti ai `pthread`

Terminazione e attesa

Potete uscire, con un certo valore, da un thread

- facendo un return dalla funzione iniziale, oppure
- chiamando:

```
void pthread_exit(void *retval);
```

E attenderne la terminazione con:

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

- se non volete attenderne la terminazione usate pthread_detach, altrimenti rimane un thread *zombie*

Argomento e valore di ritorno sono dei (void *).

Interi e puntatori non possono essere mischiati arbitrariamente, *ma* usando qualche typedef è possibile fare qualche “schifezza comoda” 😊

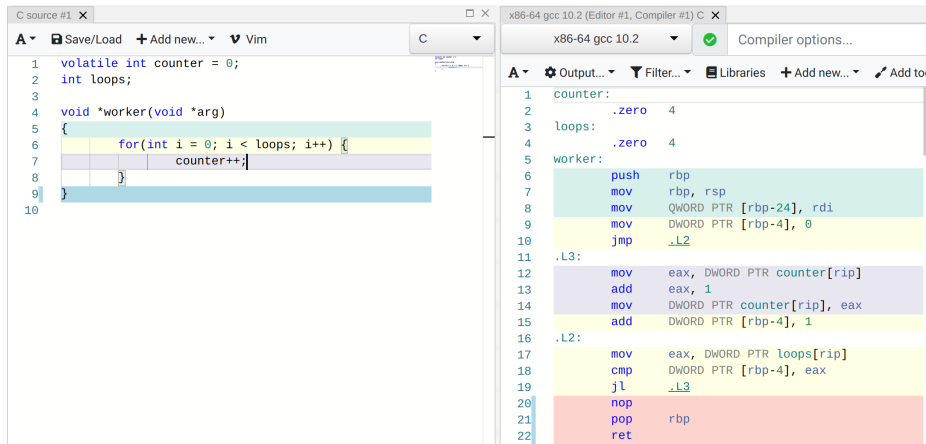
<https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/inttypes.h.html>

Vediamo qualche esempio:

- `→threads_101.c`
- `→threads.v0.c`

Perché?

Grazie a <https://gcc.gnu.org/> possiamo capire facilmente la ragione del comportamento bizzarro:



The image shows a side-by-side comparison of C source code and its assembly output. The left pane, titled 'C source #1', contains the following code:

```
1 volatile int counter = 0;
2 int loops;
3
4 void *worker(void *arg)
5 {
6     for(int i = 0; i < loops; i++) {
7         counter++;
8     }
9 }
10
```

The right pane, titled 'x86-64 gcc 10.2 (Editor #1, Compiler #1) C', shows the assembly output for the same code:

```
1 counter:
2     .zero    4
3 loops:
4     .zero    4
5 worker:
6     push    rbp
7     mov     rbp, rsp
8     mov     QWORD PTR [rbp-24], rdi
9     mov     DWORD PTR [rbp-4], 0
10    jmp     .L2
11 .L3:
12     mov     eax, DWORD PTR counter[rip]
13     add     eax, 1
14     mov     DWORD PTR counter[rip], eax
15     add     DWORD PTR [rbp-4], 1
16 .L2:
17     mov     eax, DWORD PTR loops[rip]
18     cmp     DWORD PTR [rbp-4], eax
19     jl      .L3
20     nop
21     pop     rbp
22     ret
```

Sezioni critiche e *race-condition*

- **Sezione critica** = frammento di codice che accede a **risorsa condivisa**
- Quando si hanno più flussi di esecuzione, si parla di ***race condition*** quando **il risultato finale dipende dalla temporizzazione** o dall'ordine con cui vengono schedulati
 - La computazione è non-deterministica
 - Per esempio, si ha una *race-condition* quando più thread eseguono (più o meno allo stesso tempo) una sezione critica
- Per evitare questi problemi, serve **sincronizzazione** fra i thread
 - alle volte, anche di processi diversi
- Esistono tante *primitive di sincronizzazione*, che vedrete in PCAD (Programmazione Concorrente e Algoritmi Distribuiti) al terzo anno
- Qui presentiamo solo le primitive per la **mutua esclusione**

Outline

- 1 Introduzione
- 2 Primitive di sincronizzazione
 - Implementazione
 - Spin-lock
 - Inversione delle priorità
- 3 Funzioni “rientranti”
- 4 Bug tipici
 - Deadlock

Lock

- Il problema dell'esempio era la mancanza di **atomicità** nell'incrementare il contatore
- Una soluzione è introdurre dei **lock**, implementati dai **mutex** in **pthread**; un lock:
 - si dichiara con **pthread_mutex_t** e si può inizializzare tramite:
 - assegnazione della costante **PTHREAD_MUTEX_INITIALIZER**
 - **pthread_mutex_init**
 - può essere acquisito (preso/tenuto), da un thread alla volta, tramite **pthread_mutex_lock**
 - va rilasciato, il prima possibile, tramite **pthread_mutex_unlock**

Questa parte di slide è basata su

<http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf>

→ **threads.v1.c** (anche **time** e **meld**)

→ **threads.v2.c**

Quanti lock?

Se un programma usa più strutture dati (condivise) diverse, probabilmente meglio avere più lock

- Aumenta efficienza (ma può complicare la gestione)
- Senza esagerare; per esempio, un lock per ogni nodo di una lista sarebbe troppo: considerate anche l'overhead

Per esempio, in Xv6

In kernel/proc.c:

```
struct {  
    struct spinlock lock;  
    struct proc proc[NPROC];  
} ptable;
```

In kernel/file.c:

```
struct {  
    struct spinlock lock;  
    struct file file[NFILE];  
} ftable;
```

...etc. etc.

Implementazione di lock

Come possiamo costruire dei lock? Come possiamo confrontare implementazioni diverse?

Vanno considerate:

- ① mutua esclusione (AKA “funziona?”)
- ② fairness/starvation
- ③ performance

Consideriamo varie possibilità. . .

Disabilitare le interruzioni

```
void lock() {  
    DisableInterrupts();  
}
```

```
void unlock() {  
    EnableInterrupts();  
}
```

Ha senso? Vari problemi:

- istruzioni privilegiate
- vogliamo davvero permettere a un processo di disabilitare le interruzioni e monopolizzare la CPU?
- nei sistemi multi-processore/core non funziona: due core possono eseguire la stessa sezione critica
- ...

Proviamo un'altra strada...

Una semplice struttura dati

```
typedef struct __lock_t { int is_locked; } lock_t;

void init(lock_t *mutex) { // 0 -> lock is available, 1 -> held
    mutex->is_locked = 0;
}

void lock(lock_t *mutex) {
    while (mutex->is_locked == 1) // TEST the is_locked
        ; // spin-wait (do nothing)
    mutex->is_locked = 1; // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->is_locked = 0;
}
```

Problema: non funziona

Supponiamo che `is_locked` (che il libro chiama `flag`) sia 0

Thread 1

`call lock()`

`while (flag == 1)`

interrupt: switch to Thread 2

`flag = 1; // set flag to 1 (too!)`

Thread 2

`call lock()`

`while (flag == 1)`

`flag = 1;`

interrupt: switch to Thread 1

Figure 28.2: Trace: No Mutual Exclusion

Serve supporto hardware

Diversi processori forniscono delle primitive atomiche quali

- *Test-and-Set*
- **scambi atomici** (il libro, chiama TestAndSet anche questi...)
- ...altri, che non ci interessano: l'idea è sempre la stessa

Su x86 abbiamo l'istruzione XCHG che corrisponde a:

```
/* pseudo-code (nel x86, istruzione XCHG) */  
int AtomicExchange(int *ptr, int new) {  
    int old = *ptr;  
    *ptr = new;  
    return old;  
}
```

Con queste istruzioni si possono implementare degli **spin-lock**...

Spin-lock: implementazione

```
typedef struct __lock_t {
    int is_locked;
} lock_t;

void init(lock_t *lock) {
    lock->is_locked = 0;
}

void lock(lock_t *lock) {
    while (AtomicExchange(&lock->is_locked, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->is_locked = 0;
}
```

Questo funziona?

Spin-lock: analisi

- garantisce mutua-esclusione? sì
- è *fair*? Cioè, un thread in attesa ha qualche garanzia di ottenere, prima o poi, il lock? no; è possibile *starvation*
- cosa succede se un thread che ha già acquisito il lock cerca di farne nuovamente il lock? lock *non ricorsivo* (ne esistono varianti ricorsive)
- performance? Dobbiamo distinguere rispetto al numero di core:
 - singolo core: se un thread che ha il lock viene descheduled, gli altri sprecano tempo e CPU
 - core multipli: funziona discretamente bene

Ha senso aspettare in un loop, consumando CPU?

- se si aspetta poco sì: i context-switch costano
- negli anni, varie proposte di approcci ibridi, ovvero “lock in due fasi” (un po' di spin, poi eventualmente sleep)

- 1 Introduzione
- 2 Primitive di sincronizzazione
 - Implementazione
 - Spin-lock
 - Inversione delle priorità
- 3 Funzioni “rientranti”
- 4 Bug tipici
 - Deadlock

Inversione delle priorità

Scheduling e (spin-)lock possono interagire in modi inaspettati

Supponete di avere un scheduler a priorità e due thread:

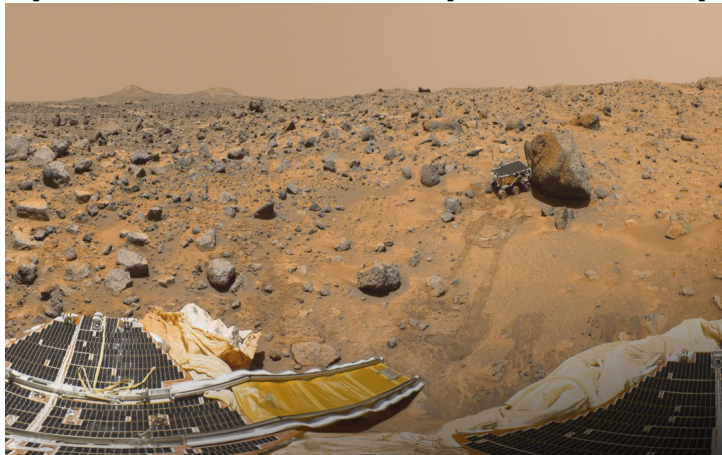
- T1 bassa priorità, T2 alta priorità
- supponiamo che T2 sia in attesa di qualcosa, va in esecuzione T1
- T1 acquisisce un certo lock L
- T2 torna ready
- lo scheduler deschedula T1 e manda in esecuzione T2
- T2 va in spin-wait per il lock L
- Game Over

Ci sono modi per risolvere il problema, per esempio “ereditare la priorità” di un thread in attesa (se maggiore di quella corrente); come è stato fatto per il NASA Pathfinder. . .

NASA Pathfinder

priority inversion which subsequently caused a deadline-miss of a critical task, which was identified by a watchdog timer, and finally, the action in such faulty scenario was to reset the spacecraft

http://www.cse.chalmers.se/~risat/Report_MarsPathFinder.pdf



Outline

- 1 Introduzione
- 2 Primitive di sincronizzazione
 - Implementazione
 - Spin-lock
 - Inversione delle priorità
- 3 Funzioni “rientranti”
- 4 Bug tipici
 - Deadlock

Funzioni rientranti e thread-safety: un po' di storia

Definizione nata in un'epoca single-threaded:

Funzione rientrante (single-thread)

Funzione che si comporta correttamente anche se interrotta a metà di un'esecuzione per essere nuovamente chiamata

Esempi: *interrupt-handler* o ricorsione

Funzione thread-safe

Funzione che si comporta correttamente anche se eseguita da più thread contemporaneamente

Due concetti collegati, ma distinti. Uno *non* implica l'altro...

Reentrancy vs thread-safety

Una funzione può essere

- rientrante perché
 - ① salva alcune variabili globali in variabili locali, in entrata
 - ② manipola tali variabili globali
 - ③ le ripristina in uscita
- thread-safe perché usa TLS o mutex non-ricorsivi
 - una ricorsione potrebbe produrre risultati errati (o “piantarsi”)

Vedere anche:

<https://stackoverflow.com/questions/856823/threadsafe-vs-re-entrant>

Funzioni rientranti “alla POSIX”

POSIX.1c (1995) introduce i thread e definisce *rientrante rispetto ai thread*, quindi sinonimo di *thread-safe*

- d'ora in poi useremo questa definizione di “rientrante”
- non tutte le funzioni POSIX sono rientranti
- quelle che non lo sono, hanno tipicamente la variante *_r*

Per evitare ambiguità, attualmente nel `man` troverete:

- *MT-Safe* o *Thread-Safe*
- tutte le funzioni non marcate esplicitamente come “safe” sono da considerare *MT-Unsafe*
- in alcuni casi, le funzioni possono essere *conditionally safe*; per es.
 - solo l'inizializzazione (*init*) potrebbe essere unsafe
 - potrebbero esserci race-condition (*race*)
 - ...

Per ulteriori dettagli: `pthreads(7)` e `attributes(7)`

Esempi:

- `sin, cos, ...`
- `putchar`
- `strerror`
- `strtok`

Outline

- 1 Introduzione
- 2 Primitive di sincronizzazione
 - Implementazione
 - Spin-lock
 - Inversione delle priorità
- 3 Funzioni “rientranti”
- 4 Bug tipici
 - Deadlock

Tipi di bug

Del codice *corretto* in un mondo *single-threaded*, può avere molti problemi quando si passa a in un mondo *multi-threaded*

Questa parte di slide è basata su

<http://pages.cs.wisc.edu/~remzi/OSTEP/threads-bugs.pdf>

Vediamo qualche esempio...

Violazione dell'atomicità (1/2)

```
struct foo *p;  
  
void f() {  
    ...  
    if (p!=NULL) {  
        p->bar = 3;  
    }  
}
```

Violazione dell'atomicità (2/2)

```
struct foo *p;  
  
void f() {  
    ...  
    if (p!=NULL) {  
        p->bar = 3; // boom; p==0  
    }  
}
```

Violazione dell'ordine (1/2)

```
/* some pointer type */ glob_thread = NULL;

void init()
{
    ...
    glob_thread = create_thread(thread_func, ...);
    ...
}

void thread_func(...)
{
    ...
    foo = glob_thread->bar;
    ...
}
```

Violazione dell'ordine (2/2)

```
/* some pointer type */ glob_thread = NULL;

void init()
{
    ...
    glob_thread = create_thread(thread_func, ...);
    ...
}

void thread_func(...)
{
    ...
    foo = glob_thread->bar; // boom: glob_thread is 0
    ...
}
```

Deadlock

Consideriamo due thread:

Thread-A `lock(m1); lock(m2);`

Thread-B `lock(m2); lock(m1);`

Cosa può succedere?

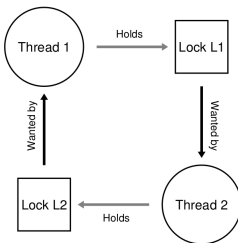


Figure 32.2: The Deadlock Dependency Graph

Condizioni necessarie per il deadlock

Condizioni per il deadlock:

- mutual exclusion: i thread hanno controllo esclusivo
- hold-and-wait: i thread mantengono le risorse acquisite mentre ne richiedono/aspettano altre
- no preemption: le risorse non possono essere tolte
- circular wait: ci deve essere un ciclo nelle attese

Come prevenirlo? I modi più semplici sono:

- Fare in modo che i lock vengano sempre acquisiti tutti assieme, atomicamente (no hold-and-wait)
- **Imporre un ordine sui lock e acquisirli in ordine** (no circular wait)