

Esercitazione [01]

Processi e Thread

Riccardo Lazzeretti lazzeretti@diag.uniroma1.it

Alessio Izzillo izzillo@diag.uniroma1.it

Sistemi di Calcolo 2

Programmazione dei Sistemi di Calcolo Multi-Nodo

Corso di Laurea in Ingegneria Informatica e Automatica

Sommario

- Thread
 - Riepilogo primitive C
 - Esempi
- Processi vs thread
 - Tempi per lancio e terminazione
 - Interazione con la memoria virtuale
- Accesso concorrente a variabili condivise

Obiettivi

1. Capire le differenze tra processi e thread in termini di reattività, e da cosa dipendono
2. Imparare le basi della programmazione multi-threading
 - a. Impostare un'applicazione multi-threading
 - b. Comprendere le problematiche legate all'accesso concorrente

Thread in C – Primitive (1/2)

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

- *thread*: puntatore a variabile di tipo **pthread_t**, su cui verrà memorizzato l'ID del thread creato
- *attr*: attributi di creazione ← sempre *NULL* in questo Corso
 - https://man7.org/linux/man-pages/man3/pthread_attr_init.3.html
- *start_routine*: funzione da eseguire (prende **sempre** come argomento un `void*` e restituisce un `void*`)
- *arg*: puntatore da passare come argomento alla funzione `start_routine`
- ✓ Return value: 0 in caso di successo, altrimenti la causa dell'errore

Thread in C – Primitive (2/2)

```
void pthread_exit (void* value_ptr);
```

- Termina il thread corrente, rendendo disponibile il valore puntato da *value_ptr* ad una eventuale operazione di join
- All'interno di *start_routine*, può essere sostituita da *return*

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

- Attende esplicitamente la terminazione del thread con ID *thread*
- Se *value_ptr* \neq *NULL*, vi memorizza il valore eventualmente restituito dal thread (un *void**, tramite *pthread_exit*)
- ✓ Return value: 0 in caso di successo, altrimenti la causa dell'errore

```
int pthread_detach(pthread_t thread);
```

- Notifica al sistema che non ci saranno operazioni di join su *thread*
- ✓ Return value: 0 in caso di successo, altrimenti la causa dell'errore

Thread in C – Un primo esempio

```
void* thread_stuff(void *arg) {  
    // codice thread che non usa argomenti  
    return NULL;  
}
```

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

```
int ret;  
pthread_t thread;  
ret = pthread_create(&thread, NULL, thread_stuff, NULL);  
if (ret != 0) {  
    fprintf(stderr, "ERROR with pthread_create!\n");  
    exit(EXIT_FAILURE);  
}  
// codice main indipendente dal thread  
ret = pthread_join(thread, NULL);  
if (ret != 0) [...]
```

Example: Multiple Threads

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) {
    printf("Hello Thread\n");
}

main() {
    pthread_t tid[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], NULL, hello, NULL);

    for (int i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}
```

Thread in C – Passaggio argomenti

Scenario: lancio di N thread con argomenti necessariamente distinti

```
void* foo(void *arg) {  
    type_t* obj_ptr = (type_t*) arg; // cast ptr argomenti  
    /* <corpo del thread> */  
    free(obj_ptr);  
    return NULL;  
}  
  
pthread_t* threads = malloc(N * sizeof(pthread_t));  
type_t* objs = malloc(N * sizeof(type_t));  
for (i=0; i<N; i++) {  
    objs[i] = [...] // imposto argomenti thread i-esimo  
    ret = pthread_create(&threads[i], NULL, foo, &objs[i]);  
    if (ret != 0) [...]  
}
```

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


Processi vs Thread

- Performance

Lanciare/terminare un thread è più veloce rispetto a lanciare/terminare un processo

- Memoria

La creazione di un processo via `fork()` comporta la «copia» dell'intera memoria del padre, mentre i thread la condividono

- Comunicazione

I thread di uno stesso processo possono usare la sua memoria per comunicare tra loro, mentre la comunicazione tra processi richiede meccanismi aggiuntivi (*Inter-Process Communication*)

Misurazione performance

- Tempo di esecuzione di una porzione di codice
- In fase di compilazione
 - Includere il modulo `performance.c`
 - Linkare le librerie run time (`-lrt`) e math (`-lm`)

```
#include "performance.h"
```

```
...
```

```
timer t; begin(&t);
```

```
// porzione di codice di cui cronometrare l'esecuzione
```

```
end(&t); unsigned long int time;
```

```
time=get_seconds(&t); time=get_milliseconds(&t);
```

```
time=get_microseconds(&t); time=get_nanoseconds(&t);
```

Tempi di lancio/terminazione

- Misurazione della reattività (tempo richiesto) di processi e thread nelle fasi di lancio e terminazione
- Sorgenti: `reactivity-processes.c`, `reactivity-threads.c`

Compilazione:

- `gcc -o reactivity-processes reactivity-processes.c performance.c -lrt -lm`
- `gcc -o reactivity-threads reactivity-threads.c performance.c -lpthread -lrt -lm`
- In ogni programma vengono eseguiti N test per calcolare le reattività medie
 - N passato come argomento dalla linea di comando
- Lo speedup deve essere calcolato come:
$$\frac{\text{tempo reactivity process}}{\text{tempo thread process}}$$

Esercizio 1

- Nella slide 9 vengono elencate alcune differenze tra processi e thread rispetto alla condivisione di memoria
- È possibile sfruttarne una per modificare `reactivity-process.c` `reactivity-thread.c` e rendere ancor meno reattiva la sezione multi-process, a vantaggio della sezione multi-thread!
- Esercizio
 - Individuare la differenza da sfruttare
 - Capire come sfruttarla nel corpo dei processi figli e dei thread
 - Modificare allo stesso modo la funzione `do_work()` in `reactivity-processes.c` e `reactivity-thread.c` di conseguenza
 - *Caveat: copy-on-write per la memoria virtuale in Linux!*
 - Verificare l'aumento dello speedup

Accesso concorrente a variabili condivise

- Cosa succede quando più thread accedono in scrittura ad una variabile condivisa in concorrenza?
 - **Sorgente:** `concurrent_threads.c`
 - **Compilazione:** `gcc -o concurrent_threads concurrent_threads.c -lpthread`
- N thread in parallelo che aggiungono M volte un valore V ad una variabile condivisa (inizializzata a 0)
- La variabile condivisa alla fine dovrebbe valere $N * M * V$: succede sempre?

Esercizio 2

- Nelle prossime lezioni presenteremo meccanismi di sincronizzazione pensati per risolvere questi problemi
- Tuttavia, è possibile implementare una soluzione che non usa meccanismi di sincronizzazione, pur mantenendo la semantica originale:
 - N thread effettuano in parallelo M incrementi di valore V
 - Al termine, il main thread verifica che tali incrementi equivalgano complessivamente a $N * M * V$
- **Modificare `concurrent_threads.c` di conseguenza**
 - Suggerimento: lavorare sulle strutture dati per evitare accessi concorrenti in scrittura