

# Esercitazione [01]

## Processi e Thread

Leonardo Aniello <aniello@dis.uniroma1.it>

Daniele Cono D'Elia <delia@dis.uniroma1.it>

Giuseppe Laurenza <laurenza@dis.uniroma1.it>

Federico Lombardi <lombardi@dis.uniroma1.it>

Sistemi di Calcolo - Secondo modulo (SC2)

Programmazione dei Sistemi di Calcolo Multi-Nodo

Corso di Laurea in Ingegneria Informatica e Automatica

A.A. 2016-2017

# 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
- *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;
}

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);
}
ret = pthread_join(thread, NULL);
if (ret != 0) [...]
```

# 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) [...]
}
```

# 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 (*shared memories*)

# 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
- Sorgente: `reactivity.c`
- Compilazione:  

```
gcc -o reactivity reactivity.c performance.c  
-lpthread -lrt -lm
```
- Vengono eseguiti N test per calcolare le reattività medie
  - N passato come argomento
  - Viene calcolato lo speedup come:  $\frac{\text{reattività processi}}{\text{reattività thread}}$

# Esercizio 1

- Nella slide 8 vengono elencate alcune differenze tra processi e thread rispetto alla condivisione di memoria
- È possibile sfruttarne una per modificare `reactivity.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 `reactivity.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**
  - Suggestione: lavorare sulle strutture dati per evitare accessi concorrenti in scrittura