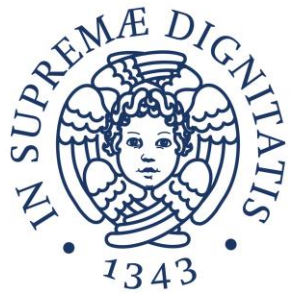

LABORATORIO DI SISTEMI OPERATIVI

Corso di Laurea in Ingegneria Informatica

A.A. 2023/2024

Ing. Maurizio Palmieri


maurizio.palmieri@unipi.it

ESERCITAZIONE 8

Thread POSIX nel sistema Linux (parte I)

I thread (processi leggeri)

- Il thread è un flusso di esecuzione indipendente all'interno di un processo
  - Ad un singolo processo possono essere associati più thread
  - I thread condividono le risorse e lo spazio di indirizzi (o parte di esso) con gli altri thread del processo
  - I thread sono anche detti "processi leggeri", in quanto
    - La creazione/distruzione di thread è meno onerosa rispetto alla creazione/distruzione di un processo
    - Il cambio di contesto fra thread dello stesso processo è meno oneroso rispetto al cambio di contesto fra processi

I thread (processi leggeri)

- Vantaggi dell'approccio multithreaded
  - Interazioni più semplici ed efficienti basate su risorse comuni
  - Passaggio di contesto fra thread meno oneroso

- Svantaggi
  - Va gestita la concorrenza fra thread: il codice utilizzato deve essere *thread safe*
    - Il codice è scritto in modo da garantire il corretto comportamento del programma e l'assenza di interazioni non volute fra i thread
    - Le risorse condivise devono essere accedute in mutua esclusione

I thread in Linux

- Linux supporta nativamente, a livello di kernel, il concetto di thread
  - Il thread è l'unità di scheduling e può essere eseguito in parallelo con altri thread
  - Il "processo tradizionale" dei sistemi Unix può essere visto come un thread che non condivide risorse con altri thread

Libreria pthreads

- Lo standard POSIX definisce la libreria pthreads per la programmazione di applicazioni multithreaded portabili

- Utilizzo
  - Includere la libreria `#include <pthread.h>`
  - Compilare specificando l'uso della libreria

    ```
    gcc <opzioni> file.c -lpthread
    ```
  - **Su DEBIAN**

    ```
    gcc <opzioni> file.c -lpthread -std=c99
    ```

  - Pagine del manuale sulla libreria

    ```
    man pthreads
    man nomefunzione
    ```

Identificatori del thread

- Un thread è identificato da un ID, di tipo `pthread_t`
- Funzione per conoscere ID del thread corrente:

```
pthread_t pthread_self(void)
```

- E' un "tipo opaco", che può essere utilizzato solo mediante apposite funzioni. Ad esempio:
  - Non ha senso stamparlo a video
  - Per fare un confronto fra due ID thread è necessario usare la funzione `pthread_equals(tid1, tid2)`

- Su Linux c'è anche la funzione `gettid()`, che ritorna un thread ID (TID) analogo del process ID (PID)
  - Se il thread è l'unico thread del processo, il suo TID è uguale al PID
  - `gettid()` è Linux-specific, quindi non deve essere usata se si vuole ottenere del codice "portabile" su sistemi Unix tradizionali

Creazione di un thread

- In Linux l'esecuzione di un programma determina la creazione di un primo thread che esegue il codice del main
- Il thread iniziale può generare una gerarchia di thread utilizzando:

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

Creazione di un thread

- `pthread_t* thread`
  Puntatore ad identificatore di thread, dove verrà scritto l'ID del thread creato
- `const pthread_attr_t* attr`
  Attributi del thread, NULL per utilizzare valori di default
- `void* (*start_routine)(void *)`
  Puntatore alla funzione che contiene il codice del nuovo thread
- `void* arg`
  Puntatore che viene passato come argomento a `start_routine`
- Il valore di ritorno è zero in assenza di errore, diverso da zero altrimenti

Terminazione e join

- Un thread può terminare la sua esecuzione con:

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

- `pthread_exit` ha i seguenti effetti:
  - L'esecuzione del thread termina e il sistema libera le risorse allocate
  - Quando un thread "padre" (es. il main) termina prima dei thread figli:
    - Se non chiama la pthread_exit → i figli vengono terminati
    - Se chiama la pthread_exit → i figli continuano la loro esecuzione
- `void* retval`
  Valore di ritorno del thread (exit status) consultabile da altri thread che utilizzano la pthread_join

Terminazione e join

- Un thread può bloccarsi in attesa della terminazione di un thread specifico:

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

- `pthread_t thread`
  ID del thread di cui attendere la terminazione
- `void** retval`
  Puntatore al puntatore dove verrà salvato l'indirizzo restituito dal thread con la pthread_exit. Può essere impostato a NULL (in questo caso viene ignorato)

- Ritorna zero in caso di successo, altrimenti un codice di errore (ad esempio se un altro thread ha già fatto join sullo stesso thread, o se c'è un rischio di deadlock)

Esempio creazione thread (1/3)

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


/* Corpo del thread */
void* tr_code(void* arg)
{
    printf("Hello World! My arg is %d\n", *(int*)arg);
    free(arg);
    pthread_exit(NULL);
}
```

Esempio creazione thread (2/3)

```
/* Main function */
int main ()
{
    pthread_t tr1, tr2;
    int* arg1 = (int*)malloc(sizeof(int));
    int* arg2 = (int*)malloc(sizeof(int));
    *arg1 = 1;
    *arg2 = 2;
    int ret;
    ret = pthread_create(&tr1, NULL, tr_code, arg1);
    if (ret){
       printf("Error: return code from pthread_create is %d\n", ret);
       exit(-1);
    }
```

Esempio creazione thread (3/3)

```
    ret = pthread_create(&tr2, NULL, tr_code, arg2);
    if (ret){
       printf("Error: return code from pthread_create is %d\n", ret);
       exit(-1);
    }

    pthread_exit(NULL);
}
```

Esempio creazione e passaggio di parametri con NTHREADS

```
… headers
#define NTHREADS 10
… codice thread
int main ()
{
     pthread_t tr[NTHREADS];
     int* args[NTHREADS];
     int ret;
     for (int i=0; i<NTHREADS; i++) {
        args[i] = (int*)malloc(sizeof(int));
        *args[i] = i;
        ret = pthread_create(&tr[i], NULL, tr_code, args[i]);
         … gestione ret value
     }
     pthread_exit(NULL);
}
```

Mutua esclusione

- Per risolvere problemi di mutua esclusione, la libreria pthread mette a disposizione l'astrazione della variabile di tipo **mutex**, analoga all'astrazione di semaforo binario
  - Una variabile mutex permette di proteggere l'accesso a variabili condivise su cui operano più thread

Mutex

- Nella libreria pthread è definito il tipo `pthread_mutex_t` che rappresenta implicitamente
  - Lo stato del mutex
  - La coda dove verranno sospesi i processi in attesa che il mutex sia libero

- E' un semaforo binario, quindi il suo stato può assumere due valori (libero o occupato)

Mutex – inizializzazione

- Definizione di una variabile mutex:

```
pthread_mutex_t M;
```

- Per inizializzare la variabile mutex, si utilizza la funzione:

```
int pthread_mutex_init(pthread_mutex_t* M,
        const pthread_mutexattr_t* mattr)
```

  - `pthread_mutex_t* M`
    Puntatore al mutex da inizializzare
  - `const pthread_mutexattr_t* mattr`
    Puntatore a una struttura con attributi di inizializzazione. Con NULL vengono utilizzati i valori di default (mutex libero).

Mutex – lock e unlock

- La wait sulla variabile mutex è realizzata con la primitiva:

```
int pthread_mutex_lock(pthread_mutex_t* M)
```

- La signal sulla variabile mutex è realizzata con la primitiva:

```
int pthread_mutex_unlock(pthread_mutex_t* M)
```

- Ritornano zero in caso di successo, altrimenti un codice di errore

Mutex – utilizzo

- Utilizzo della variabile mutex
  - Definizione e inizializzazione

```
pthread_mutex_t M;
pthread_mutex_init(&M, NULL);
```

  - Lock sulla variabile mutex prima di accedere alla risorsa condivisa

```
pthread_mutex_lock(&M);
```

  - Unlock sulla variabile mutex dopo aver utilizzato la risorsa condivisa

```
pthread_mutex_unlock(&M);
```

- Se più thread provano ad accedere alla risorsa (lock), solo uno di essi potrà accedere, mentre gli altri rimarranno bloccati
  - Dopo aver occupato e utilizzato la risorsa, il thread provvederà a "liberarla" con la primitiva unlock: in questo modo uno dei thread (eventualmente) bloccati sulla variabile mutex potrà accedere alla risorsa

ESERCIZI

Esercizio 1.1

- Scrivere un programma C in cui il main genera un numero NTHREADS=4 di thread.
  - A ciascun thread figlio viene passato un intero <arg> che parte da 1 e arriva a NTHREADS
- I thread eseguono tutti lo stesso codice (stessa funzione)
  - Ciclo for con 4 iterazioni in cui:
    - Viene stampato un messaggio del tipo "Sono il thread <arg>"
    - Il thread va in sleep per <arg> secondi.
- Il padre dopo aver creato i thread figli chiama la pthread_exit(NULL)
  - Cosa succede se commentiamo questa chiamata?

Esercizio 1.2

- Modificare il codice in modo che il thread padre faccia join in attesa del secondo figlio prima di terminare
  - Controllare la riuscita dell'operazione
- A questo punto, modificare il codice dei thread in modo che il secondo thread figlio creato faccia join in attesa del padre prima di terminare
  - Suggerimento: prima di creare i thread, salvare il thread ID del padre in una variabile globale
  - Cosa succede? Le join hanno successo?

Esercizio 1.3

- Rimuovere le pthread_join e tornare al punto 1.1
- Rimuovere la funzione sleep dal ciclo for dei thread
- Aumentare a 100 000 il numero di iterazioni, e a 12 il numero di thread (NTHREADS=12).
- Aggiungere una variabile globale int cont = 0;
- Dentro il ciclo for dei thread
  - Incrementare cont (cont++)
  - Aggiungere al messaggio stampato anche il valore di cont:
  - "Sono il thread <arg>, il valore di cont è <cont>"
  - Eseguire il programma più volte e controllare il valore finale di cont
- Il risultato è sempre quello atteso?

Esercizio 2

- Modificare il codice dell'esercizio 1.3 in modo che l'accesso alla risorsa condivisa (contatore cont) avvenga in modo corretto
  - Sfruttare una variabile globale mutex, inizializzata nel main e utilizzata dai thread per accedere alla "sezione critica"