

Thread

- Un **thread** o **lightweight process** (LWP) è un'unità di utilizzo della CPU che consiste di:
 - un program counter,
 - un insieme di registri,
 - uno stack space.Tale unità condivide con gli altri peer thread le seguenti risorse:
 - codice,
 - dati,
 - risorse del sistema operativo (e.g., file aperti, segnali).
- Quindi, siccome i thread hanno un address space comune, la loro creazione ed il cambio di contesto sono notevolmente meno costosi rispetto ai corrispondenti meccanismi che riguardano i processi.
- I vantaggi dovuti all'uso dei thread sono maggiormente visibili nei sistemi multiprocessore, ma anche nei sistemi con un'unica CPU si hanno dei benefici. Infatti, è possibile sfruttare i tempi di latenza delle operazioni di I/O di un thread per eseguirne nel frattempo un altro.

Creazione di thread

- In Linux si utilizzano le librerie pthread (POSIX thread) per lavorare con i thread:

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

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

- in `thread` viene memorizzato il thread ID,
- `attr` specifica gli attributi del nuovo thread (il valore NULL fa in modo che vengano utilizzati gli attributi di default),
- `start_routine` è il puntatore alla funzione che verrà eseguita dal nuovo thread,
- `arg` è un puntatore all'argomento passato alla funzione `start_routine`; per passare più argomenti è sufficiente passare il puntatore ad una struttura.

Attributi di un thread

- Gli attributi del nuovo thread sono specificati impostando l'argomento `attr` che è un puntatore alla struttura `pthread_attr_t` definita in `bits/pthreadtypes.h`.
- Gli attributi principali sono:
 - `detachstate`: può assumere i valori `PTHREAD_CREATE_JOINABLE` (default) o `PTHREAD_CREATE_DETACHED`,
 - `schedpolicy`: può assumere i valori `SCHED_OTHER` (default), `SCHED_RR` o `SCHED_FIFO`.
 - `schedparam`: indica la priorità della politica di scheduling (valore di default: 0),
 - `inheritsched`: può assumere i valori `PTHREAD_EXPLICIT_SCHED` (default) o `PTHREAD_INHERIT_SCHED`,
 - `scope`: `PTHREAD_SCOPE_SYSTEM` (default) o `PTHREAD_SCOPE_PROCESS`.
- Per utilizzare i valori di default è sufficiente passare il valore `NULL` come argomento.

Terminazione di un thread

Un thread può terminare la propria esecuzione in uno dei modi seguenti:

- eseguendo l'istruzione `return` della funzione `start_routine`,
- eseguendo la funzione `pthread_exit`:

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

dove il valore di ritorno puntato da `retval` può essere utilizzato da un altro thread che esegua la funzione `pthread_join`.

- eseguendo la funzione `exit` che termina il processo e tutti i relativi thread.

Esempio

(compilare con `gcc -lpthread -o thread thread.c`)

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

void print_msg(void *ptr);

main() {
    pthread_t thread1,thread2;
    char *msg1="Thread 1";
    char *msg2="Thread 2";

    if(pthread_create(&thread1,NULL,(void *)&print_msg,(void *)msg1)!=0) {
        perror("Errore nella creazione del primo thread.\n");
        exit(1);
    }
    if(pthread_create(&thread2,NULL,(void *)&print_msg,(void *)msg2)!=0) {
        perror("Errore nella creazione del secondo thread.\n");
        exit(1);
    }
    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
    exit(0);
}

void print_msg(void *ptr) {
    printf("%s\n",(char *)ptr);
}
```

Sincronizzazione di thread (Mutex)

- Un mutex (MUTual EXclusion) è un meccanismo utile per proteggere strutture di dati condivise da modifiche concorrenti (e.g., in modo da implementare regioni critiche).

- Un mutex è inizializzabile con la seguente sintassi:

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
```

- Successivamente il codice compreso fra le chiamate `pthread_mutex_lock` e `pthread_mutex_unlock` potrà essere eseguito soltanto da un thread alla volta:

```
pthread_mutex_lock(&mutex);  
...  
pthread_mutex_unlock(&mutex);
```

- Se un thread non riesce a bloccare un mutex (perché già bloccato da un altro thread), viene sospeso fintanto che il thread che lo sta bloccando non lo rilascia.

Sincronizzazione di thread (Join)

- Un join permette ad un thread di sospendere la propria esecuzione in attesa che un altro thread termini la propria esecuzione.
- Solitamente un processo, in fase di inizializzazione, crea diversi thread (ognuno con uno scopo ben preciso) e si pone in attesa del loro completamento.
- La chiamata di sistema da utilizzare per il join è:

```
int pthread_join(pthread_t th, void **thread_return);
```

Nel caso in cui come secondo argomento non si specifichi NULL, il valore di ritorno del thread `th` specificato tramite la funzione `pthread_exit`, viene memorizzato nella locazione puntata da `thread_return`.

Sincronizzazione di thread (condition variable)

- Una **condition variable** è una variabile di tipo `pthread_cond_t` che viene utilizzata per sospendere l'esecuzione di un thread in attesa che si verifichi un certo evento.
- Le condition variable vanno sempre utilizzate associandole ad un mutex per evitare che si verifichino problemi di deadlock.

- Una condition variable viene inizializzata tramite la seguente sintassi:

```
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
```

- Per mettersi in attesa un thread può utilizzare una delle seguenti system call:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);  
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,  
                           const struct timespec *abstime);
```

- La segnalazione del verificarsi di una condizione può essere fatta tramite le seguenti system call (`pthread_cond_broadcast` fa ripartire tutti i thread in attesa sulla condizione `cond`):

```
int pthread_cond_signal(pthread_cond_t *cond);  
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Sincronizzazione di thread (condition variable)

Inizializzazione:

```
pthread_mutex_t condition_mutex=PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t condition_cond=PTHREAD_COND_INITIALIZER;
```

Codice del thread che si mette in attesa:

```
pthread_mutex_lock(&condition_mutex);  
...  
pthread_cond_wait(&condition_cond,&condition_mutex );  
...  
pthread_mutex_unlock(&condition_mutex);
```

Codice del thread che segnala la condizione:

```
pthread_mutex_lock(&condition_mutex);  
...  
pthread_cond_signal(&condition_cond);  
...  
pthread_mutex_unlock(&condition_mutex);
```

Documentazione online

- Nel libro di testo consigliato per la parte del corso sulla programmazione di sistema non c'è una trattazione dei thread.
- É possibile reperire dei tutorial sul Web in modo da integrare le informazioni disponibili nelle man page.
- Un buon tutorial (in Inglese) è disponibile all'indirizzo seguente:

<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

Esercizio

Modificare il programma del primo esercizio della Lezione 21, utilizzando i thread, in modo da implementare l'accesso esclusivo al file delle prenotazioni tramite regioni critiche gestite da mutex.

Progetto III: programmazione concorrente

Si consideri la seguente situazione: vi sono N entità (processi oppure thread, a scelta) che vogliono usufruire del servizio di stampa (soltanto a video, per semplicità) messo a disposizione da un'ulteriore entità (processo oppure thread, a scelta).

Realizzare un programma C che simuli la situazione precedentemente descritta, garantendo quanto segue:

- ogni entità che vuole usufruire del sistema di stampa comunica separatamente due linee di testo (per semplicità `linea 1`, `linea 2`) da stampare;
- l'entità che gestisce il servizio di stampa riceve le due linee da stampare e le visualizza su schermo nel seguente formato:

```
# x: linea 1  
# x: linea 2
```

dove x è il numero progressivo (da 1 a N) che identifica l'entità mittente;

- dopo che le due linee inviate sono state visualizzate, l'entità che le ha spedite effettua una pausa con tempo random e ripete l'invio delle due linee;
- l'entità che gestisce il servizio di stampa deve garantire quanto segue:
 - le linee inviate da ogni singola entità devono essere visualizzate nell'ordine corretto (prima linea 1 e poi linea 2);
 - le linee inviate da entità distinte non devono accavallarsi, cioè, non devono avvenire cose del genere:

```
# x: linea 1
```

```
# y: linea 1
```

```
# x: linea 2
```

dove x è diverso da y ;

- deve essere garantito che ogni richiesta di accesso al servizio venga soddisfatta, cioè, non deve esserci un'attesa indefinita (*starvation*) della relativa entità.