---------- seti 05/11 ----------

un thread è un flusso di esecuzione all'interno di un processo

- creare un thread significa quindi:
  - creare una "CPU virtuale"
  - allocare uno stack per il nuovo flusso di esecuzione
- la differenza principale fra thread e processi è la condivisione (o meno) dello spazio di indirizzamento, mentre ogni processo è completamente isolato dagli altri processi
- dal punto di vista logico, ogni thread ha il suo
  - stack, regione di memoria dove vengono allocate le variabili locali, salvati gli indirizzi di ritorno di una funzione, metadati, ecc
    però tutti questi stack vivono nello stesso spazio di indirizzamento -> se un puntatore va dove non deve, un thread può leggere o scrivere all'interno dello stack di un altro thread
  - variabile errno (che sta nel proprio TLS, thread local storage -> memoria privata che ha ogni thread) che restituisce il perché del fallimento di una system call

il **thread local storage** è uno spazio locale a ogni thread, accessibile tramite interfaccia di 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

potremmo voler usare i thread per

- sfruttare il parallelismo, suddividere il carico di lavoro
- facilitare l'input/output

per creare un thread c'è una libreria api posix, **pthread**

```c
#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.
```

il primo argomento è un puntatore a pthread_t (equivalente di pid_t. identifica il thread); il secondo parametro permette di specificare attributi opzionali; il terzo parametro è un puntatore a funzione che prende un void * e restituisce un void *, e questo sarà il codice eseguito sul thread; il paramentro arg di tipo void * è l'argomento che è passato a start_routine

in caso di successo la funzione restituise 0, altrimenti restituiscono il codice di errore
ma non viene scritto su errno
per uscire, con un certo valore, da un thread posso:

- fare un return dalla funzione iniziale -> esco da tutti i thread
- chiamare **void pthread_exit(void * retval);** -> esco solo dal thread scelto

posso attendere la terminazione di un thread tramite

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

dove specifico quale thread voglio aspettare e in retval verrà scritto il valore di uscita di quel thread

- se non voglio attendere la terminazione di un thread, uso **pthread_detach**, altrimenti diventerà un thread zombie

sezione critica -> un pezzo di codice che va ad accedere ad una risorsa condivisa
race condition -> quando il risultato finale dipende dalla temporizzazione, dipende da come sono schedulati i thread
per evitare questi problemi serve sincronizzare i thread, a volte anche di processi diversi

possiamo rendere la sezione critica atomica utilizzando i lock, o mutex
prima di iniziare la sezione critica, un thread deve accedere al lock, accedere al mutex, quindi prendersi il lock sul mutex

- 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`

Quanti lock usare all'interno di un programma?
Non c'è una regola fissa, se uso un unico lock in tutto il programma, vuol dire che quando dei thread devono sincronizzarsi su qualcosa utilizzano quell'unico lock
sarebbe quindi più intelligente utilizzare un lock all'interno di ogni struttura dati all'interno di un programma

Come possiamo implementare i lock?
Quando implementiamo un lock dobbiamo prendere in considerazione:

- se la nostra implementazione offre mutua esclusione
- fairness / starvation -> un thread in attesa ha qualche garanzia di ottenere, prima o poi, il lock?
- performance

Senza un supporto hardware non si può implementare -> diversi processori forniscono delle primitive atomiche quali:

- Test-and-set -> singola operazione che testa il valore di un'allocazione di memoria e se trova 0 lo mette a 1
- Scambi atomici -> inverte il valore di 2 allocazioni di memoria

```
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;
}
```

- garantisce mutua-esclusione
- non è fair ed 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 deschedulato, 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)

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

Inversione delle priorità:

supponiamo di avere uno scheduler a priorità e due thread:

- con T1 bassa priorità e T2 alta priorità
- assumiamo che T2 sia in attesa di qualcosa, quindi va in esecuzione T1
- T1 acquisisce un certo lock che chiamiamo L
- T2 torna ready
- lo scheduler deschedula T1 e manda in esecuzione T2
- T2 va in spin-wait per il lock L
- game over -> T1, che è l'unico che può rilasciare il lock L, non viene messo in esecuzione, perché T2 ha priorità

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 piú thread contemporaneamente

definizioni separate -> una non implica l'altra

ma in posix useremo la definizione di "rientrante" quando parliamo di funzioni thread-safe

- non tutte le funzioni posix sono rientranti
- quelle che non lo sono, hanno tipicamente la variante _r