

THREAD POSIX

TPSIT

Libreria <pthread.h>

- ▶ Lo standard ANSI C non prevede l'utilizzo dei Thread.
- ▶ Lo standard POSIX definisce una nuova libreria contenente delle nuove funzioni per la gestione dei thread.

Che cos'è un thread?

- ▶ Un thread non è altro che una funzione C che viene eseguita in maniera concorrente ad altre funzioni nell'ambito di uno stesso processo.
- ▶ Tutti i thread che fanno parte di uno stesso processo ne condividono lo spazio di indirizzamento.
- ▶ Le variabili che vengono dichiarate localmente all'interno della funzione che implementa il codice di ciascun thread sono private per quello specifico thread, e pertanto non accessibili da parte degli altri thread del processo.
- ▶ La memoria condivisa sarà pertanto rappresentata dalle variabili globali del processo.
- ▶ Gli identificativi dei thread sono gestiti mediante il tipo `pthread_t`, che è la ridefinizione di un `Unsigned long int`

Abilitazione dei thread

- ▶ **`int pthread_setconcurrency (int nThread);`**
- ▶ Comunica al sistema il numero di thread che si intende creare.
- ▶ Restituisce 0 in caso di successo, qualsiasi cosa diversa da 0 in caso di insuccesso.
- ▶ `nThread` rappresenta il numero di thread da creare

Creazione di un nuovo thread

- ▶ **int pthread_create (pthread_t * tID, const pthread_attr *attr, void *(*foo)(void*), void *arg);**
- ▶ Restituisce 0 in caso di successo, qualsiasi cosa diversa da 0 in caso di insuccesso.
- ▶ **tID** è l'indirizzo dell'oggetto pthread_t destinato a contenere l'id che il sistema assegna al thread
- ▶ **attr** sono gli attributi che l'utente vuole assegnare al thread. (se vale NULL vengono usati quelli di default)
- ▶ **foo** è il nome della funzione C che verrà eseguita all'interno del nuovo thread.
- ▶ **arg** è il puntatore ai parametri da passare alla funzione foo. Deve assolutamente essere un void *

Attendere un thread

- ▶ **`int pthread_join (pthread_t tID, void ** risPt);`**
- ▶ Sospende il thread principale (main) in attesa che il thread identificato da tID giunga al termine
- ▶ risPt rappresenta l'indirizzo del puntatore al valore restituito dalla funzione foo eseguita nel thread
- ▶ Ritorna 0 in caso di successo, diverso da 0 altrimenti.
- ▶ Non è prevista invece, nello standard POSIX, una funzione di attesa di un thread generico.

Funzioni utili

- ▶ **int pthread_kill (pthread_t tid, int signo);**
 - ▶ Invia un segnale di terminazione al thread specificato dal 1° par Ritorna 0 in caso di successo, diverso da 0 altrimenti.
 - ▶ **signo** è l'identificatore del segnale che si vuole inviare al thread (normalmente SIGKILL definito in "signal.h").
- ▶ **int pthread_self ();**
 - ▶ restituisce il tID del thread corrente.

Istruzioni per la compilazione

- ▶ Per compilare un programma multithread occorre specificare il flag **-l pthread** sulla linea di comando
- ▶ Es:
- ▶ `gcc main.c -o main -lpthread`

SEMAFORI DI MUTUA ESCLUSIONE: MUTEX

- ▶ Un mutex è una variabile che serve per la protezione delle sezioni critiche:
 - ▶ Variabili condivise e modificabili da più thread
 - ▶ Solo un thread alla volta può accedere ad una risorsa protetta da un mutex
- ▶ Il mutex è un semaforo **binario**, cioè un valore può essere 0 (occupato) oppure 1 (libero)

MUTEX

- ▶ I mutex possono essere paragonati a classiche serrature:
 - ▶ Il primo thread che ha accesso alla coda dei lavori lascia fuori gli altri thread fino a quando il suo compito non è stato portato a termine
- ▶ I threads, dunque, «posizionano» un mutex nelle sezioni di codice nelle quali vengono condivisi i dati

MUTEX: Logica d'uso

- ▶ Creare ed inizializzare una variabile mutex
- ▶ Più thread tentano di accedere alla risorsa invocando l'operazione di lock
- ▶ Un solo thread riesce ad acquisire il mutex mentre gli altri si bloccano
- ▶ Il thread che ha acquisito il mutex manipola la risorsa
- ▶ Lo stesso thread la rilascia invocando la unlock
- ▶ Un altro thread acquisisce il mutex e così via
- ▶ Distruzione della variabile mutex

Creazione Mutex

- ▶ Per creare un mutex è necessario usare una variabile di tipo **pthread_mutex_t** contenuta nella libreria pthread
- ▶ **pthread_mutex_t** è una struttura che contiene:
 - ▶ Nome del mutex
 - ▶ Proprietario
 - ▶ Contatore
 - ▶ Struttura associata al mutex
 - ▶ La coda dei processi sospesi in attesa che mutex sia libero
 - ▶
- ▶ Per il tipo di dato **pthread_mutex_t**, è definita la macro di inizializzazione **PTHREAD_MUTEX_INITIALIZER**
- ▶ Il mutex è un tipo definito "ad hoc" per gestire la mutua esclusione quindi il valore iniziale può essergli assegnato anche in modo statico mediante questa macro
- ▶ **pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;**

Mutex: Lock

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

- ▶ Su mutex sono possibili solo due operazioni: locking e unlocking (equivalenti a wait e signal sui semafori)
- ▶ Ogni thread, prima di accedere ai dati condivisi, deve effettuare la lock su una stessa variabile mutex
- ▶ Blocca l'accesso da parte di altri thread
- ▶ Se più thread eseguono l'operazione di lock su una stessa variabile mutex, solo uno dei thread termina la lock e prosegue l'esecuzione, gli altri rimangono bloccati nella lock. In tal modo, il processo che continua l'esecuzione può accedere ai dati (protetti mediante la mutex).

Mutex: trylock

```
int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

► Valore di ritorno:

- 0 in caso di successo e si ottenga la proprietà della mutex
- EBUSY se il mutex è occupato

Mutex: Unlock

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

- ▶ Un altro thread che ha precedentemente eseguito la lock della mutex potrà allora terminare la lock ed accedere a sua volta ai dati.
- ▶ Valore di ritorno: 0 in caso di successo

Mutex: destroy

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

- ▶ Elimina il mutex
- ▶ Valore di ritorno:
 - ▶ 0 in caso di successo
 - ▶ EBUSY se il mutex è occupato

Semafori POSIX

- ▶ Possono svolgere il ruolo di semafori binari (stile mutex) o generali (n-ari) a seconda di quale valore viene assegnato durante l'inizializzazione.
- ▶ Ciascun semaforo ha una propria mutex per garantire la mutua esclusione delle proprie variabili interne (sezione critica).

Verranno utilizzati gli UNNAMED SEMAPHORES:

- ▶ Permettono di sincronizzare thread
- ▶ Condivisi tra thread o processi
- ▶ Si inizializzano con **sem_init**

Semafori POSIX

- ▶ I semafori sono una variabile di tipo **sem_t**
- ▶ Per poterli gestire occorre includere la libreria `<semaphore.h>`
- ▶ Operazioni degli unnamed semaphores
 - ▶ `int sem_init(sem_t *sem, int pshared, unsigned value);`
 - ▶ `int sem_destroy(sem_t *sem);`
- ▶ Atomiche: operano in mutua esclusione, in particolare nell'accesso a loro contatore interno
 - ▶ `int sem_wait(sem_t *sem); /* WAIT (mutex lock) */`
 - ▶ `int sem_post(sem_t *sem); /* SIGNAL (mutex unlock) */`
 - ▶ `int sem_trywait(sem_t *sem); /* TRY WAIT (mutex trylock) */`

Semafori: inizializzazione

```
int sem_init(sem_t *sem, int pshared, unsigned value);
```

- ▶ Inizializza in unnamed semaphore
- ▶ Restituisce
 - ▶ 0 in caso sia inizializzato con successo
 - ▶ -1 in caso di fallimento
- ▶ Parametri
 - ▶ Sem: identificativo del semaforo creato
 - ▶ Pshared:
 - ▶ 0: solo i threads del processo che lo crea può usare il semaforo
 - ▶ Non-0: altri processi possono usare il semaforo
 - ▶ Value: valore iniziale del semaforo

Semafori: inizializzazione

- ▶ L'inizializzazione di un semaforo può fallire
- ▶ In caso di insuccesso `sem_init` restituisce `-1` e setta **errno**:
 - ▶ `Errno = EINVAL` => `Value > sem_value_max`
 - ▶ `Errno = ENOSPC` => Resources exhausted
 - ▶ `Errno = EPERM` => Insufficient privileges
- ▶ Initial value deve essere ≥ 0

```
sem_t semA;
```

```
if(sem_init(&semA, 0, 1) == -1)
```

```
    printf(«Fallimento nell'inizializzazione del semaforo\n»);
```

Semafori: distruzione

```
int sem_destroy(sem_t *sem);
```

- ▶ Elimina il semaforo e le risorse da esso allocate
- ▶ Resistuisce
 - ▶ 0: in caso di successo
 - ▶ -1: in caso di fallimento, e setta **errno**
- ▶ Parametri
 - ▶ Sem: identificativo del semaforo
- ▶ Può distruggere un semaforo solo una volta
- ▶ Distruggere un semaforo su cui un thread è bloccato causa risultati inattesi.

Semafori: blocco

```
int sem_wait(sem_t *sem)
```

- ▶ Blocco il thread sul semaforo
 - ▶ Controlla il valore del semaforo
 - ▶ Se il valore è minore o uguale a zero:
 - ▶ Si blocca e riparte uscendo dalla wait quando diventa uguale a zero
 - ▶ Poi decrementa il valore del semaforo
 - ▶ Se il valore è maggiore di zero
 - ▶ Prosegue uscendo subito dalla wait
 - ▶ Poi decrementa il valore del semaforo
- ▶ Restituisce
 - ▶ 0: in caso di successo
 - ▶ -1: in caso di errore, setta **errno**
- ▶ Parametri
 - ▶ sem: identificativo semaforo
 - ▶ sem > 1: decrementa e basta
 - ▶ sem == 1: decrementa e blocca altri thread (lock)
 - ▶ sem <= 0: si blocca e quando viene sbloccato decrementa (thread blocks)

Semafori: sblocco

```
int sem_post(sem_t *sem);
```

- ▶ Sblocca semaforo, cioè
 - ▶ Incrementa di 1 il valore del semaforo
 - ▶ Se il valore diventa > 0 qualche altro thread bloccato sulla `sem_wait` può continuare l'esecuzione
- ▶ Restituisce
 - ▶ 0: in caso di successo
 - ▶ -1: in caso di errore, e setta **errno**
- ▶ Parametri:
 - ▶ `sem`: identificativo semaforo
 - ▶ `Sem > 0`: nessun thread è stato bloccato su questo semaforo, il valore si incrementa
 - ▶ `Sem == 0`: un thread viene bloccato

Semafori: controllo condizione

```
int sem_trywait(sem_t *sem);
```

- ▶ Valuta la condizione del semaforo
- ▶ Non blocca il thread
- ▶ Restituisce
 - ▶ 0 in caso di successo
 - ▶ -1 in caso di insuccesso, e setta **errno** (se **EAGAIN** il semaforo è già bloccato)
- ▶ Parametri
 - ▶ sem: identificativo semaforo
 - ▶ **Sem > 1: decrementa e basta**
 - ▶ **Sem == 1: decrementa e blocca altri thread (lock)**
 - ▶ **Sem <= 0: Non si blocca e non decrementa, ma restituisce -1 con errno == EAGAIN**

Semafori: Valore iniziale assegnato

- ▶ Il valore iniziale assegnato al semaforo coincide con il numero di risorse che, inizialmente, possono essere contemporaneamente elaborate da più thread.
- ▶ Semafori n-ari (Generali):
 - ▶ Valore iniziale $N > 0$
 - ▶ Significa che i primi N thread possono chiamare `sem_wait` e proseguire senza bloccarsi.
 - ▶ Lo $N+1$ esimo thread che chiama la `sem_wait` si blocca fino a che un thread non fa la `sem_post`