

I Thread Posix

I Thread

- processi “leggeri”
- un processo può avere diversi thread
- i thread di uno stesso processo condividono la memoria ed altre risorse
 - facile comunicare tra thread!
- pthread = “POSIX thread”
- per compilare: “gcc -pthread pippo.c”

Risorse condivise

- I thread di uno stesso processo condividono:
 - la memoria
 - il pid e il ppid
 - i file descriptor
 - le reazioni ai segnali
- I thread **non** condividono: lo stack e il program counter

Identificare i thread

- processo process id (pid) pid_t
 - thread thread id (tid) pthread_t
-

```
pthread_t pthread_self(void);
```

- restituisce il tid del thread corrente

Creare un thread

```
typedef void* (*thread_start)(void *);  
  
int pthread_create(pthread_t          *tid,  
                  const pthread_attr_t *attributes  
                  thread_start       start,  
                  void               *argument);
```

- restituisce 0 se OK, un codice d'errore altrimenti
- tid = argomento di ritorno, conterrà il tid del nuovo thread
- attributes = attributi del thread (vedere dopo), anche NULL
- start = indirizzo della funzione da cui partire
- argument = l'argomento passato alla funzione start

Trattare gli errori

- siccome i thread condividono la memoria, è meglio non usare una variabile globale (come `errno`) per i codici d'errore
 - quindi, le funzioni pthread restituiscono direttamente un codice d'errore
-

```
char *strerror(int n);
```

- restituisce un messaggio corrispondente al codice d'errore `n`

Terminare un thread

- invocare `exit()` fa terminare *l'intero processo!*
- per terminare solo il thread corrente, si può:
 - invocare `return` (a meno che non sia il thread principale)
 - invocare `pthread_exit` (vedere dopo)
 - far sì che un altro thread chiami `pthread_cancel` (vedere dopo)

Terminare un thread

```
void pthread_exit(void *ret);
```

- termina il thread corrente, con valore di uscita `ret`
- altri thread possono leggere il valore di uscita usando `pthread_join` (vedere dopo)
- fare attenzione che i dati puntati da `ret` sopravvivano alla terminazione del thread!
 - `ret` non deve puntare allo stack
 - no a variabili locali
 - si a variabili globali o allocate dinamicamente

Aspettare la terminazione di un thread

```
int pthread_join(pthread_t tid, void **ret);
```

- attende che il thread specificato da `tid` termini
 - se quel thread è già terminato, ritorna subito (come `wait`)
- restituisce 0 se OK, un codice d'errore altrimenti
- `ret` è un parametro di ritorno usato per restituire il valore d'uscita del thread che termina
 - occhio al doppio puntatore!
 - se il valore di uscita non ci interessa, passiamo NULL come valore di `ret`

Esercizio 1

- Scrivere un programma che accetta un intero n da riga di comando, crea n thread e poi aspetta la loro terminazione
- Ciascun thread aspetta un numero di secondi casuale tra 1 e 10, poi incrementa una variabile globale intera ed infine ne stampa il valore
- Domanda: ci sono race conditions in questo programma?

Esercizio 2

- Scrivere un programma che prende in input un intero n , il nome di un file di testo ed un carattere x
- Il programma ha il compito di contare le occorrenze del carattere x nel file di testo
- Il programma esegue tale compito creando n thread, ognuno dei quali esamina una porzione diversa del file di testo
 - ad esempio, se il file è lungo 1000 bytes ed $n=4$, il primo thread esaminerà i primi 250 bytes, il secondo thread esaminerà i 250 bytes successivi, e così via

Cancellare un thread

```
int pthread_cancel(pthread_t tid);
```

- chiede che il thread specificato da `tid` venga terminato
 - non *aspetta* la terminazione
- restituisce 0 se OK, un codice d'errore altrimenti
- il valore di uscita di un thread cancellato è dato dalla costante `PTHREAD_CANCELED`

Thread e fork/exec

- Se un thread chiama `fork`, nasce un nuovo processo con un solo thread
 - potenziali problemi con i mutex in possesso di altri thread
- Se un thread chiama `exec`, l'intero processo diventa il nuovo programma
 - cioè, gli altri thread spariscono

Thread e segnali

- Le chiamate a `signal` influenzano tutti i thread
- Se arriva un segnale a un processo, succede che:
 - se il processo ha impostato un handler, il segnale arriva ad *uno qualunque* dei thread (che esegue l'handler)
 - se invece la reazione al segnale consiste nel terminare il processo, *tutti i thread* vengono terminati

Inviare un segnale a un thread

```
int pthread_kill(pthread_t tid, int signo);
```

- manda il segnale `signo` al thread specificato da `tid`
 - se è impostato un handler, viene eseguito nel thread `tid`
 - se non è impostato un handler, e il comportamento di default è di terminare il processo, vengono comunque terminati *tutti i thread*
- restituisce 0 se OK, un codice d'errore altrimenti

Attributi di un thread

- un thread può essere creato in “detached state” (stato sconnesso)
- un thread può bloccare i tentativi di essere cancellato (cancellabilità)
- altri attributi
 - posizione e dimensione dello stack
 - attributi real-time

Detached state

- Se non ci interessa il valore di ritorno di un thread, conviene crearlo in *detached state*
 - però, poi non possiamo chiamare `pthread_join`

Gestire gli attributi di un thread

```
int pthread_attr_init    (pthread_attr_t *attr);  
int pthread_attr_destroy(pthread_attr_t *attr);
```

- inizializza e distrugge una struttura per gli attributi di un thread. Uso:
 - si alloca una struttura `pthread_attr_t`
 - si chiama `pthread_attr_init`
 - si modificano gli attributi contenuti nella struttura usando apposite funzioni (vedere dopo)
 - si passa la struttura a `pthread_create`
 - si distrugge la struttura con `pthread_attr_destroy`
- restituiscono 0 se OK, un codice d'errore altrimenti

Impostare il detached state

```
int pthread_attr_setdetachstate(pthread_attr_t *attr,  
                               int detachstate);
```

- imposta l'attributo detach-state della struttura puntata da `attr`
- l'argomento `detachstate` può essere:
 - `PTHREAD_CREATE_JOINABLE` (default)
 - `PTHREAD_CREATE_DETACHED`
- restituisce 0 se OK, un codice d'errore altrimenti

Cancellabilità

- In ogni istante, un thread può essere cancellabile o non cancellabile
- Quando partono tutti i thread sono cancellabili
- Quando un altro thread chiama `pthread_cancel`
 - se il thread è cancellabile, viene cancellato
 - se non è cancellabile, la richiesta di cancellazione viene memorizzata, in attesa che il thread diventi cancellabile

Impostare la cancellabilità

```
int pthread_setcancelstate(int state, int *oldstate);
```

- imposta la cancellabilità a `state` e restituisce la vecchia cancellabilità in `oldstate`
- `state` e `oldstate` possono assumere i valori:
 - `PTHREAD_CANCEL_ENABLE`
 - `PTHREAD_CANCEL_DISABLE`
- restituisce 0 se OK, un codice d'errore altrimenti

Sommario

- fork
- exit
- waitpid
- kill
- getpid
- processo zombie
- pthread_create
- pthread_exit
- pthread_join
- pthread_kill
- pthread_self
- thread terminato in attesa di pthread_join

Note su Linux

- Linux supporta lo standard dalla versione 2.6 del kernel
 - per sapere la versione del kernel, usare “uname -a”
- La versione 2.4 invece si discosta dallo standard
 - i thread hanno pid diversi!