Thread

Contiene lucidi tratti da: 2006-2007 Marco Faella, Clemente Galdi, Giovanni Schmid (Università di Napoli Federico II), 2005-2007

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”

Thread

- Ad un generico processo, sono associati i seguenti dati e le seguenti informazioni:
  - –codice del programma in esecuzione
  - –un’area di memoria contenente le strutture dati dichiarate nel programma in esecuzione
  - –fileaperti
  - –stack(per le chiamate di procedure e funzioni)
  - –contenuto dei registridella CPU

Thread

Consideriamo due processi che devono lavorare sugli stessi dati.

Come possono fare, se ogni processo ha la propria area dati (ossia, gli spazi di indirizzamento sono separati)?

– i dati possono essere scambiati mediante messaggi (pipe, FIFO) o tenuti in memoria condivisa

– i dati possono essere tenuti in un file che viene acceduto a turno dai due processi.

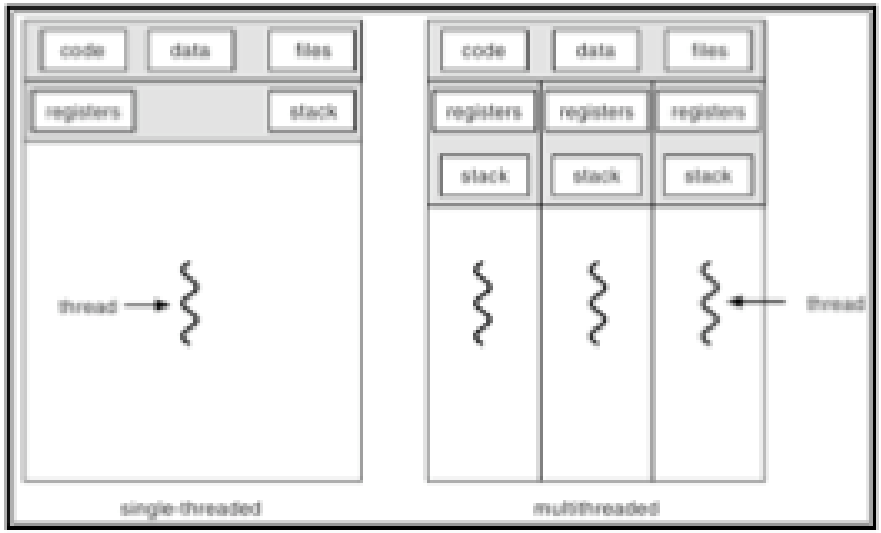
Non sarebbe comodo poter avere processi che possano automaticamente lavorare sugli stessi dati, senza usare meccanismi espliciti di condivisione/comunicazione?

Thread

- Il context switch tra processi richiede molto lavoro al SO: oltre a cambiare il valore dei vari registri, deve spostarsi dalle aree dati e di codice del processo uscente, a quelle del processo entrante.
- Se dati e codice di quest'ultimo erano stati swappati in memoria secondaria, occorre prima riportarli in memoria primaria.
- Se due (o più) processi potessero condividere dati e codice, il context switch fra di loro sarebbe molto più veloce
- Per soddisfare questo tipo di esigenza è nato il concetto di thread

Thread

Un processo **Multi-Thread** è fatto di più **thread**, detti peer thread.



Thread

- Ad ogni thread è associato in modo esclusivo il suo stato della computazione, fatto da:
  - valore del program counter e degli altri registri della CPU
  - uno stack

- Ma un thread condivide con i suoi peer thread:
  - il codice in esecuzione
  - i dati
  - i file aperti

Thread

- I context switch avviene anche tra ognuno dei peer thread che formano task, in modo che tutti possano portare avanti la computazione
- Ma il context switch fra peer thread richiede il salvataggio e il ripristino solo dei registri della CPU e dello stack (che sono diversi per ogni thread)
- Codice, dati e file aperti sono gli stessi per tutti, e non devono essere cambiati: IL CONTEXT SWITCH TRA PEER THREAD E' MOLTO PIU' VELOCE

Thread

I thread sono "unità esecutive" indipendenti all'interno di un processo, caratterizzate da:
- un thread ID;
- un insieme di registri, incluso un contatore di programma ed un puntatote di pila;
- una pila per le variabili locali e gli indirizzi di ritorno (di chiamate a funzioni);
- una maschera dei segnali;
- una priorità.

Tutti i thread relativi ad uno stesso processo ne condividono:
- lo spazio di indirizzamento;
- i descrittori dei file aperti;
- la disposizione ed i gestori dei segnali;
- le credenziali.

Identificare i Thread

- processo ha `process id (pid)` di tipo `pid_t`
- thread ha `thread id (tid)` di tipo `pthread_t`

Nota: `pthread_t` struttura, funzione per il confronto:

int **pthread_equal**(pthread_t t1, pthread_t t2);

Ritorna non zero se uguali, 0 se diversi

pthread_t **pthread_self**(void);

Restiuisce il tid del thread corrente

Creazione Thread

• Analogamente ai processi, quando un thread viene creato esso è associato ad un pezzo di codice;

• Tuttavia solo se il processo ospite (e quindi il suo corrispondente programma) è single-threaded, il thread è associato all'intero programma;

• Se il processo ospite è multi-threading, ciascun thread di tale processo è associato ad una funzione da eseguire, detta funzione di avvio;

• Anche i thread, come i processi, possono assumere gli stati running, sleeping, blocked e terminated ;

• Quando un thread termina la situazione è analoga al caso dei processi: è il programmatore che deve preoccuparsi affinchè tutte le risorse impegnate dal thread siano rilasciate al sistema, facendo in modo che venga effettuata una wait a livello di thread.

Creazione Thread

- Quando un programma viene mandato in esecuzione tramite una chiamata exec, viene creato un singolo thread, detto thread principale o iniziale;

- Ulteriori thread vanno creati esplicitamente;

- Ogni thread ha numerosi attributi, da assegnare alla creazione: livello di priorità, dimensione iniziale della pila, etc;

- All'atto della creazione di un thread è necessario specificare la funzione di avvio: il thread inizierà la propria esecuzione richiamando la funzione di avvio;

- La terminazione di un thread può avvenire in tre diverse circostanze: (a) una chiamata esplicita di terminazione del thread, (b) la funzione di avvio ritorna, (c) il processo contenente il thread termina.

Funzione pthread_create

Per creare thread addizionali relativi ad uno stesso processo, Posix prevede la funzione:

```
#include <pthread.h>

int  pthread_create ( pthread_t *tid, const pthread_attr_t *attr,
                      void * ( *start_func) (void *), void *arg );
```

- se la chiamata ha successo, *tid* punta al thread ID;
- *attr* permette di specificare gli attributi del thread (se *attr* = NULL, gli attributi sono quelli di default);
- *start_func* è l'indirizzo della funzione di avvio;
- *arg* è l'indirizzo dell'argomento accettato dalla funzione di avvio;
- restituisce 0 in caso di successo, un intero positivo – secondo le convenzioni di <sys/errno.h> – in caso di errore.

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)
- start = indirizzo della funzione da cui partire
- argument = l'argomento passato alla funzione start

Esempio

```
#include <pthread.h>
pthread_t ntid;

void printids(const char *s){
  pid_t pid;
  pthread_t tid;
  pid = getpid();
  tid = pthread_self();
  printf("%s pid %u tid (0x%x)\n", s, (unsigned int)pid,
  (unsigned int)tid, (unsigned int)tid);
}

void * thr_fn(void * arg){
 printids("new thread; ");
 return ((void *) 0);
}

int main(void){
  int err;
  err = pthread_create(&ntid, NULL, thr_fn, NULL);
  if(err != 0){
    printf("can't create thread %s \n", strerror(err));
    exit(1);}
  printids("main thread:");
  sleep(1);
  exit(0);
}
```

Non c'e' modo portabile per stampare thread ID (dipende da piattaforma)

Produce (su Mac OSX):
main thread: pid 984 tid (0xa000b2a4)
new thread; pid 984 tid (0x1800200)
Come FreeBSD usa strutture, quindi tid puntatore, su Solaris interi

Sleep: processo main puo' terminare prima

pthread_self: thread creato non usa ntid perche' potrebbe non essere inizializzato

Trattare gli errori

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

---

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

- restituisce un messaggio corrispondente al codice d'errore n

Risorse condivise

- I thread di uno stesso processo condividono:
  - la memoria
  - il pid e il ppid
  - i file descriptor
  - le reazioni ai segnali
    - cioe', le chiamate a signal influenzano tutti i thread
- I thread non condividono: lo stack

Terminare un thread

- invocare `exit()` (`_exit`, `_Exit`) fa terminare *l'intero processo!*
- Analogamente un segnale ad un thread uccide il processo
- per terminare solo il thread corrente, si puo':
  - invocare return dalla rutine di start, il valore di ritorno e' l'exit code
  - invocare `pthread_exit`
  - un altro thread del processo puo' chiamare `pthread_cancel`

Terminare un thread

Un thread può richiedere esplicitamente la propria terminazione grazie alla chiamata seguente, lasciando traccia del proprio stato di terminazione per quei thread che attendono per lui:

```
#include <pthread.h>

void  pthread_exit ( void *status);
```

•*status* punta all'oggetto che definisce lo stato di terminazione del thread. Quest'ultimo non deve essere una variabile locale al thread chiamante, pena la sua scomparsa alla terminazione del thread stesso.

Terminare un thread

void **pthread_exit**(void *status);

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

Aspettare la terminazione di un thread

Un thread può attendere per la terminazione di un altro thread relativo allo stesso processo:

```
#include <pthread.h>

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

- *tid* è l'ID del thread del quale si vuole attendere la terminazione;
- *status* punta al valore restituito dal thread per cui si è atteso, indicante il suo stato di terminazione (se *status* = NULL, tale stato non viene restituito);
- restituisce 0 in caso di successo, un intero positivo – secondo le convenzioni di <sys/errno.h> – in caso di errore.

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 e' gia' terminato, ritorna subito (come wait)
- restituisce 0 se OK, un codice d'errore altrimenti
- ret e' un parametro di ritorno usato per restituire il valore d'uscita della funzione di start del thread atteso (return), se il thread e’ cancellato contiene PTHREAD_CANCELED
- se il valore di uscita non ci interessa, passiamo NULL al posto di ret

Esempio: create, join

```
/* thread_create: stampa i TID del main thread e di due altri
thread */

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

void *start_func(void *arg) /* funzione di avvio */
{
  printf("%s", (char *)arg);
  printf(" and my TID is: %d\n",(int)pthread_self());
}

int main(void)
{
  int en;
  pthread_t tid1, tid2;
  char *msg1 = "Hello world, I am thread #1";
  char *msg2 = "Hello world, I am thread #2";

  printf("The launching process has PID:%d\n",(int)getpid());

  printf("The main thread has TID:%d\n",(int)pthread_self());
```

Esempio: create, join

```
  /* crea il 1mo thread */
  if ((en = pthread_create(&tid1, NULL, start_func, msg1)!=0))
     errno=en, perror("pthread_create"), exit(1);

  /* crea il 2ndo thread */
  if ((en = pthread_create(&tid2, NULL, start_func, msg2)!=0))
     errno=en, perror("pthread_create"), exit(2);

  /* attende per il 1mo */
  if ((en = pthread_join(tid1, NULL)!=0))
     errno=en, perror("pthread_join"), exit(1);

  /* attende per il 2ndo */
  if ((en = pthread_join(tid2, NULL)!=0))
     errno=en, perror("pthread_join"), exit(2);


  return 0;
}
```

Esempio: create, join

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


int count = 0;

void *f(void *x)
{
	sleep(rand() % 10);
	count++;
	printf("Ciao! %d\n", count);
	return NULL;
}
```

```
int main(int argc, char *argv[])
{
        int i, err, n;

        if (argc != 2) {
          printf("Uso: %s <numero thread>\n", argv[0]);
          exit(1);
        }

        n = atoi(argv[1]);
        /* Allocazione consentita */
        pthread_t tid[n];

        for (i=0; i<n ;i++) {
                if ((err=pthread_create(&tid[i], NULL, f, NULL)) != 0) {
                        printf("errore: %s\n", strerror(err));
                        exit(1);
                }
        }

        for (i=0; i<n ;i++)
           pthread_join(tid[i],NULL);
        printf("finito.\n");

        return 0;
}
```

Esempio: passaggio parametri

funzione che contiene il codice di un peer thread

```
#include <pthread.h>
#include <stdio.h>
void *tbody(void *arg)
 {
int j;
printf(" ciao sono un thread, mi hanno appena creato\n");
*(int *)arg = 10;
sleep(2)          /* faccio aspettare un pò il mio creatore, poi termino */
pthread_exit((int *)50); /* oppure return ((int *)50); */
}
```

Funzione di avvio

```
main(int argc, char **argv)
{
int i;
pthread_t mythread;
void *result;
printf("sono il primo thread, ora ne creo un altro \n");
pthread_create(&mythread, NULL, tbody, (void *) &i);
 printf("ora aspetto la terminazione del thread che ho creato \n");
pthread_join(mythread, &result);
printf("Il thread creato ha assegnato %d ad i\n",i);
printf("Il thread ha restituito %d \n",result);
}
```

Passa &i al thread, Nota: void * nella start permette di passare strutture di diverso tipo

Legge &result dal thread, Nota: void * restituito dalla start permette di leggere strutture di diverso tipo

Condivisione Memoria

- I due thread condividono lo stesso spazio di indirizzamento, e quindi vedono le stesse variabili: se uno dei due modifica una variabile, la modifica è vista anche dall'altro thread.
- Nel codice precedente *il* `main` *passa al thread* `tbody` *il puntatore alla variabile i dichiarata nel main*. il thread tbody modifica la variabile, e questa modifica è vista da main.
- Nel caso dei processi tradizionali, una cosa simile è ottenibile solo usando esplicitamente un segmento di memoria condivisa.

Variabili Globali

- Ma i thread di un task possono condividere variabili in maniera ancora più semplice, usando variabili globali.

```
#include <pthread.h>
#include <stdio.h>
int global_var = 5;
void *tbody(void *arg)
 {
printf(" ciao sono un thread, ora modifico una var globale\n");
global_var = 27;
*(int *)arg = 10;
pthread_exit((int *)50); /* oppure return ((int *)50); */
}
```

Esempio: variabili globali, locali, allocazione dinamica

```
typedef struct foo{
    int a;
    int b;
} myfoo;

myfoo test; // Variabile GLOBALE


void stampa(char *st, struct foo *test){
  printf("%s: tid=%d a=%d b=%d\n", st, pthread_self(),test->a, test->b);
}

void *fun1(void *arg){
    myfoo test2 = {1,2}; // Variabile LOCALE
    printf("%s %d\n", arg, pthread_self());
    stampa(arg, &test2);
    pthread_exit((void *)&test2);
}
```

Esempio

```
void *fun2(void *arg){
   test.a = 3;
   test.b = 4; // Variabile GLOBALE
   printf("%s %d\n", arg, pthread_self());
   stampa(arg, &test);
   pthread_exit((void *)&test);
}

void *fun3(void *arg){
   myfoo *test3;
   test3=malloc(sizeof(struct foo)); // Variabile allocata dinamicamente
   test3->a = 5;
   test3->b = 6;
   printf("%s %d\n", arg, pthread_self());
   stampa(arg, test3);
   pthread_exit((void *)test3); //c
}
```

Esempio

```
int main(void){
   char st[100];
   pthread_t tid1;
   pthread_t tid2;
   pthread_t tid3;

   myfoo *b; // PUNTATORE alla struttura (non allocata)

   pthread_create(&tid1, NULL, fun1, "Thread 1"); // Locale
   pthread_join(tid1, (void *)&b);
   stampa("Master  ", b);

   pthread_create(&tid2, NULL, fun2, "Thread 2"); // Globale
   pthread_join(tid2, (void *)&b);
   stampa("Master  ", b);

   pthread_create(&tid3, NULL, fun3, "Thread 3"); // Dinamica
   pthread_join(tid3, (void *)&b);
   stampa("Master  ", b);
}
```

Esempio

Thread 1: 1077283760
// Locale
Thread 1: a=1 b=2
Master : a=1075156600 b=1077281896

Thread 2: 1077283760
// Globale
Thread 2: a=3 b=4
Master : a=3 b=4

Thread 3: 1077283760
// Dinamica
Thread 3: a=5 b=6
Master : a=5 b=6

Esercizio

- 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?

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
- Come se pthread_exit() con il valore di uscita dato dalla costante PTHREAD_CANCELED

Similitudini Thread-Processi

- fork
- exit
- waitpit
- kill
- getpid
- processo zombie

- pthread_create
- pthread_exit
- pthread_join
- pthread_kill
- pthread_self
- thread terminato in attesa di pthread_join

pthread_detach

In taluni casi è opportuno far si che lo stato di terminazione di un thread ***T*** non venga memorizzato fintanto che un altro thread ***T'*** relativo allo stesso processo attenda per ***T***, ma sia invece cancellato subito dopo la terminazione di ***T***:

```
#include <pthread.h>

int  pthread_detach ( pthread_t *tid);
```

- *tid* è l'ID del thread che si vuole distaccare;
- restituisce 0 in caso di successo, un intero positivo – secondo le convenzioni di <sys/errno.h> – in caso di errore.

però, poi non possiamo chiamare pthread_join

thread e fork

- Se un thread chiama fork, nasce un nuovo processo con un solo thread
- Potenziali problemi con i mutex in possesso di altri thread (vedere dopo)

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 e' impostato un handler, viene eseguito nel thread tid
  - se non e' impostato un handler, e il comportamento di default e' di terminare il processo, vengono comunque terminati tutti i thread
- restituisce 0 se OK, un codice d'errore altrimenti

Esempio

```
signal(SIGUSR1, usr1);
pthread_create(&tid1, NULL, fun, "Thread 1");
pthread_create(&tid2, NULL, fun, "Thread 2");
pthread_create(&tid3, NULL, fun, "Thread 3");
sleep(1);
pthread_kill(tid1, SIGUSR1);
pthread_kill(tid2, SIGUSR1);
pthread_kill(tid3, SIGUSR1);
```

(USR1 = User defined signal)

```
sigemptyset(&set); // Configura la maschera SOLO nel master thread
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_SETMASK, &set, NULL);
sleep(1);
while (i++<10){
   sleep(1);
   kill(pid, SIGUSR1); // il segnale e' intercettato da un thread
}
```

Esempio

Thread id=1077283760 ricevuto segnale
Thread id=1079385008 ricevuto segnale
Thread id=1081486256 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale
Thread id=1077283760 ricevuto segnale

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

Gestione Attributi

include <pthread.h>

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 (struttura opaca)
  - 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

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

Impostare 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

Esempio

Creazione di un thread in stato dispached:

```
#include <pthread.h>

int makethread(void * (*fn)(void *), void arg *) {
  int err;
  pthread_t tid;
  pthread_attr_t attr;

  err = pthread_attr_init(&attr);
  iff(err!=0) return (err);
  err =
  pthread_att_setdetachstate(&attr,PTHREAD_CREATE_DETACHED)
  ;
  iff (err==0)
     err = pthread_create(&tid, &attr, fn, arg);
  pthread_att_destroy(&attr);
  return err;

}
```

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

Riferimenti

- Advanced Programming in the Unix Environment (Second Ed.)
  - Threads: 11.1, 11.2, 11.3, 11.4, 11.5