2. I THREAD

2.1 Introduzione

Il tipo di parallelismo che è opportuno avere a disposizione nelle applicazioni varia in base al grado di cooperazione necessaria tra le diverse attività svolte in parallelo: da un lato abbiamo situazioni in cui le diverse attività svolte in parallelo sono totalmente indipendenti tra loro, ad esempio l'esecuzione di diversi programmi in parallelo, dall'altro abbiamo attività che devono cooperare intensamente e condividere molte strutture dati.

Il modello dei processi, che sono macchine virtuali totalmente indipendenti tra loro (anche se possono scambiarsi alcune informazioni in maniera limitata), è chiaramente più adatto alla realizzazione di attività indipendenti che alla realizzazione di funzioni fortemente cooperanti; per questo motivo la nozione di processo viene affiancata con una nozione simile, ma più adatta a risolvere il secondo tipo di problemi, la nozione di thread.

Un thread è un flusso di controllo che può essere attivato in parallelo ad altri thread nell'ambito di uno stesso processo e quindi nell'esecuzione dello stesso programma. Con flusso di controllo si intende una esecuzione sequenziale di istruzioni della macchina.

Ogni thread può eseguire un insieme di istruzioni (tipicamente una funzione) indipendentemente e in parallelo ad altri processi o thread. Tuttavia, essendo i diversi thread attivi nell'ambito dello stesso processo, essi condividono lo spazio di indirizzamento e quindi le strutture dati.

Un thread è chiamato a volte "processo leggero" (lightweight process), perchè condivide molte caratteristiche di un processo, in particolare la caratteristiche di essere un flusso di controllo sequenziale che viene eseguito in parallelo con altri flussi di controllo sequenziali; il termine "leggero" vuole indicare che l'implementazione di un thread è meno onerosa di quella di un vero processo. Tuttavia, a differenza dei processi, diversi thread possono condividere molte risorse, in particolare lo spazio di indirizzamento e quindi le strutture dati.

Riassumendo:

- Nell'ambito di un processo è possibile attivare diversi flussi di controllo, detti thread, che procedono in parallelo tra loro
- I thread di uno stesso processo condividono lo spazio di indirizzamento e altre risorse.
- Normalmente la creazione e gestione di un thread da parte del Sistema Operativo è meno costosa, in termini di risorse quali il tempo macchina, della creazione e gestione di un processo.

Esistono diversi modelli di thread, che differiscono tra loro nei dettagli, pur condividendo l'idea generale. Noi seguiremo il modello definito dallo standard POSIX (Portable Operating System Interface for Computing Environments). POSIX è un insieme di standard per le interfacce applicative (API) dei sistemi operativi. L'obiettivo principale di tale standard è la portabilità dei programmi applicativi in ambienti diversi: se un programma applicativo utilizza solamente i servizi previsti dalle API di POSIX può essere portato su tutti i sistemi operativi che implementano tali API.

I thread di POSIX sono chiamati Pthread. Nel seguito verranno illustrate solamente le caratteristiche fondamentali dei Pthread, necessarie per capirne i principi di funzionamento e per svolgere, nel capitolo successivo, alcuni esempi di programmazione concorrente; non è invece intenzione di queste note spiegare i dettagli di programmazione dei Pthread, per i quali si rimanda ai numerosi testi esistenti sull'argomento e al manuale.

2.2 Concorrenza, parallelismo e parallelismo reale

Prima di procedere è utile fare una precisazione terminologica relativamente ai termini sequenziale, concorrente e parallelo.

– diremo che 2 attività A e B sono **sequenziali** se, conoscendo il codice del programma, possiamo sapere con certezza l'ordine in cui esse vengono svolte, cioè che A è svolta prima di B (indicato sinteticamente con A<B) oppure B è svolta prima di A (B<A);

– diremo che invece le due attività sono **concorrenti** se non possiamo determinare tale ordine in base al codice del programma.

Il termine **parallelismo** in questo testo verrà usato con un significato uguale a quello di concorrenza: diremo cioè che due attività A e B sono eseguite in parallelo (o concorrentemente) se non è possibile stabilire a priori se un'istruzione di A è eseguita prima o dopo un'istruzione di B; si noti che questo può accadere in un sistema monoprocessore per il fatto che il processore esegue istruzioni di A o di B in base a proprie scelte, non determinabili a priori.

Parleremo invece di **parallelismo reale** quando vorremo indicare che le istruzioni di A e B sono eseguite effettivamente in parallelo da diversi processori. E' evidente che il parallelismo reale implica la concorrenza o parallelismo semplice, ma non viceversa.

2.3 Pthread e processi: similitudini e differenze

Come già detto, i thread sono per molti aspetti simili ai processi; possiamo considerare un thread come un "sottoprocesso" che può esistere solamente nell'ambito del processo che lo contiene. Questo fatto ha alcune importanti conseguenze:

- se un processo termina, tutti i suoi thread terminano anch'essi; per questo motivo è necessario garantire che un processo non termini prima che tutti i suoi thread abbiano ultimato lo svolgimento del loro compito
- dobbiamo distinguere tra l'identificatore del thread (ogni Pthread possiede infatti un proprio identificatore di tipo *pthread_t*) e l'identificatore del processo che lo contiene

Ad esempio, se diversi thread appartenenti allo stesso processo eseguono le istruzioni

```
pid_t miopid;
...
miopid=getpid( );
printf("%i", miopid);
```

essi stampano tutti lo stesso valore, cioè il valore del pid del processo al quale appartengono.

Se prescindiamo dal fatto che i thread sono sempre contenuti in un processo, con le conseguenze appena indicate, i thread possono essere considerati per altri aspetti come dei veri e propri sottoprocessi, perchè:

- sono eseguiti in parallelo tra loro
- possono andare in attesa di un evento di ingresso/uscita in maniera indipendente uno dall'altro (cioè se un thread di un processo si pone in attesa di un'operazione di ingresso/uscita gli altri thread dello stesso processo proseguono – non viene posto in attesa l'intero processo)

Esistono pertanto anche da un punto di vista programmativo numerose analogie tra i thread e i processi:

1. esiste una funzione di creazione dei thread, *pthread_create(...)* (corrispondente alla *fork( )*, ma con la seguente differenza fondamentale: un thread viene creato passandogli il nome di una funzione che deve eseguire, quindi la sua esecuzione inizia da tale funzione, indipendentemente dal punto di esecuzione in cui è il thread che lo ha creato);
2. esiste una funzione di attesa, *pthread_join(...)*, tramite la quale un thread può attendere la terminazione di un altro thread (funzione simile a *waitpid( )*);
3. un thread termina quando termina la funzione eseguita dal thread, per l'esecuzione di un *return( )* oppure perchè raggiunge il termine del codice eseguibile (simile a un processo che termina per esecuzione di una *exit( )* oppure perchè raggiunge il termine o la *return( )* del main( ));
4. esiste la possibilità di passare un codice di terminazione tra un thread che termina e quello che lo ha creato, se quest'ultimo ha eseguito una *pthread_join* per attenderlo (simile al valore passato dalla *exit* di un processo alla *wait* del processo padre)

Si osservi che ogni processo ha un suo flusso di controllo che possiamo considerare il **thread principale** (o **thread di default**) del processo, associato alla funzione main( ); pertanto quando viene lanciato in esecuzione un programma eseguibile viene creato nel processo un unico thread principale, che inizia l'esecuzione dalla funzione main( ), il quale a sua volta può creare altri thread tramite

la funzione *pthread_create( )*. Dopo la prima creazione di un thread da parte del thread principale esisteranno quindi due flussi di controllo concorrenti: quello principale e quello relativo al nuovo thread.

Nonostante le similitudini elencate sopra, sono da sottolineare le differenze fondamentali tra i processi e i thread; in particolare:

1) Un thread viene creato passandogli il nome di una funzione che deve eseguire, quindi la sua esecuzione inizia da tale funzione, indipendentemente dal punto in cui è il thread che lo ha creato (invece un nuovo processo inizia sempre dal punto del codice in cui si trova la *fork* che lo ha creato).

2) Un thread viene eseguito nello spazio di indirizzamento del processo in cui viene creato, *condivide quindi la memoria con gli altri thread del processo*, con una fondamentale eccezione: *per ogni thread di un processo viene gestita una diversa pila*. Da un punto di vista programmativo questo significa che:

- le variabili locali della funzione eseguita dal thread appartengono solo a quel thread, perchè sono allocate sulla sua pila;
- *le variabili non allocate sulla pila*, ad esempio le variabili statiche e globali definite in un programma in linguaggio C, *sono condivise tra i diversi thread del processo*.

2.4 Creazione e attesa della terminazione di un thread

Per creare un thread si usa la funzione *pthread_create( )* che ha 4 parametri:

1. Il primo parametro deve essere l'indirizzo di una variabile di tipo pthread_t; in tale variabile la funzione pthread_create( ) restituisce l'identificatore del thread appena creato.
2. Il secondo parametro è un puntatore agli attributi del thread e può essere NULL; questo argomento non verrà trattato (negli esempi useremo sempre NULL e il thread avrà gli attibuti standard di default).
3. Il terzo parametro è l'indirizzo della funzione che il thread deve eseguire (detta **thread_function**); questo parametro è fondamentale per il funzionamento del thread ed è obbligatorio.
4. Il quarto e ultimo parametro è l'indirizzo di un eventuale argomento che si vuole passare alla funzione; deve essere un unico argomento di tipo puntatore a void. Data questa limitazione, per passare molti

parametri a una thread_function è necessario creare una struttura dati e passare l'indirizzo di tale struttura.

E' possibile attendere la terminazione di un thread tramite la funzione *pthread_join( )*, che permette anche di recuperare uno stato di terminazione.

Esempio 1 - Il programma thread1 (figura 2.1) mostra le caratteristiche fondamentali di un programma che utilizza i pthread. Il programma consiste in un *main( )* che crea due thread tramite due invocazioni a *pthread_create( )*; ambedue i thread devono eseguire la stessa funzione *tf1( )*, alla quale viene passato come argomento il valore 1 nel primo thread e il valore 2 nel secondo thread. Il main attende poi la terminazione dei due thread creati tramite la funzione *pthread_join( )*, alla quale passa i valori degli identificatori dei due thread creati, e poi termina.

La funzione *tf1( )* ha una variabile locale *conta*, inizializzata a 0, ed esegue le due azioni seguenti:

1. incrementa il valore di conta;
2. stampa una frase nella quale identifica il thread che la sta eseguendo in base al parametro che le è stato passato e stampa il valore della variabile conta

Il programma è molto semplice, ma la sua esecuzione illustra alcuni aspetti fondamentali del meccanismo dei thread. Si consideri ad esempio la serie di esecuzioni mostrata in figura 2.2.

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

void * tf1(void *tID)  // questa è la thread_function
{
int conta =0;  //conta è una variabile locale
conta++;
printf("sono thread n: %d; conta = %d \n",(int) tID, conta);

return NULL;
}

int main(void)
{
pthread_t tID1;
pthread_t tID2;

pthread_create(&tID1, NULL, &tf1, (void *)1);
pthread_create(&tID2, NULL, &tf1, (void *)2);

pthread_join(tID1, NULL);
pthread_join(tID2, NULL);

return 0;
}
```

Figura 2.1 Codice del programma thread1

Si osservi anzitutto che nella programmazione sequenziale usuale un programma come questo, privo di dati di input, dovrebbe dare sempre lo stesso risultato. Invece, la figura 2.2 mostra che nelle diverse esecuzioni la stampa è stata eseguita talvolta prima dal thread numero 1 e poi dal thread numero 2, talvolta nell'ordine opposto. Questo fatto è dovuto all'esecuzione concorrente dei thread; noi non possiamo sapere in base al codice del programma quale sarà l'esatto ordine di esecuzione delle azioni del programma.

Per quanto riguarda i valori stampati della variabile *conta* possiamo osservare che tale variabile vale sempre 1; questo risultato è spiegato dal fatto che ogni thread possiede una sua pila, quindi durante l'esecuzione dei thread esistono due copie distinte della variabile locale *conta*, ognuna delle due viene allocata, inizializzata a 0 e quindi incrementata a 1. *In altri termini, la variabile locale conta NON è condivisa trai due thread.*

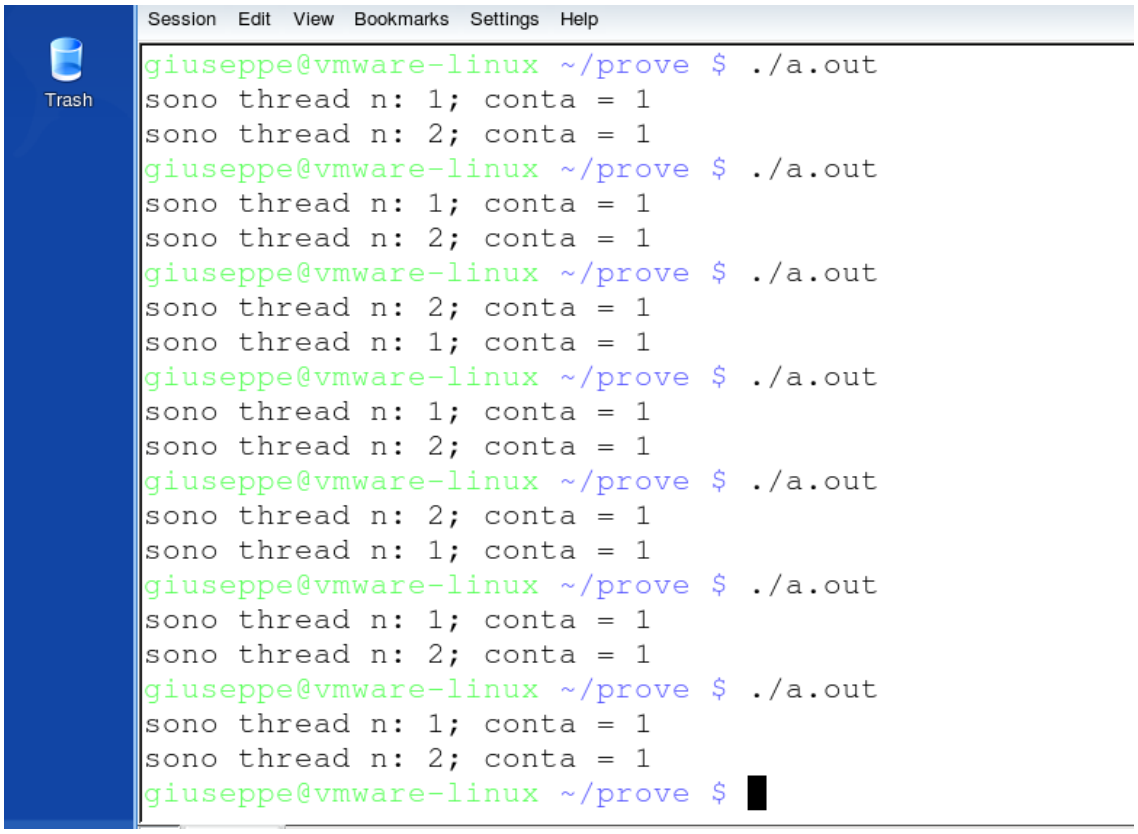


Figura 2.2 – risultato di 7 esecuzioni del programma thread1

Passiamo ora ad analizzare come cambia il comportamento dei thread utilizzando variabili statiche o globali al posto delle variabili locali.

In figura 2.3 è mostrato il programma thread2, che differisce da thread1 solamente per un piccolo particolare: la variabile *conta* è dichiarata *static*, cioè viene allocata e inizializzata una volta sola, alla prima invocazione della funzione. In successive invocazioni della stessa funzione viene utilizzata la stessa copia in memoria già creata nella prima esecuzione.

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

void * tf1(void *tID)
{
static int conta =0;  // conta è una variabile statica
conta++;
printf("sono thread n: %d; conta = %d \n",(int) tID, conta);

return NULL;
}

int main(void)
{
pthread_t tID1;
pthread_t tID2;

pthread_create(&tID1, NULL, &tf1, (void *)1);
pthread_create(&tID2, NULL, &tf1, (void *)2);

pthread_join(tID1, NULL);
pthread_join(tID2, NULL);

return 0;
}
```

Figura 2.3 Codice del programma thread2 – utilizzo di variabili statiche

Diverse esecuzioni di questo programma sono mostrate in figura 2.4.

Anche in questo caso ovviamente l'ordine di esecuzione dei thread è casuale e varia da un'esecuzione all'altra; a differenza però dal caso precedente la variabile conta viene incrementata due volte, perchè si tratta della stessa variabile.

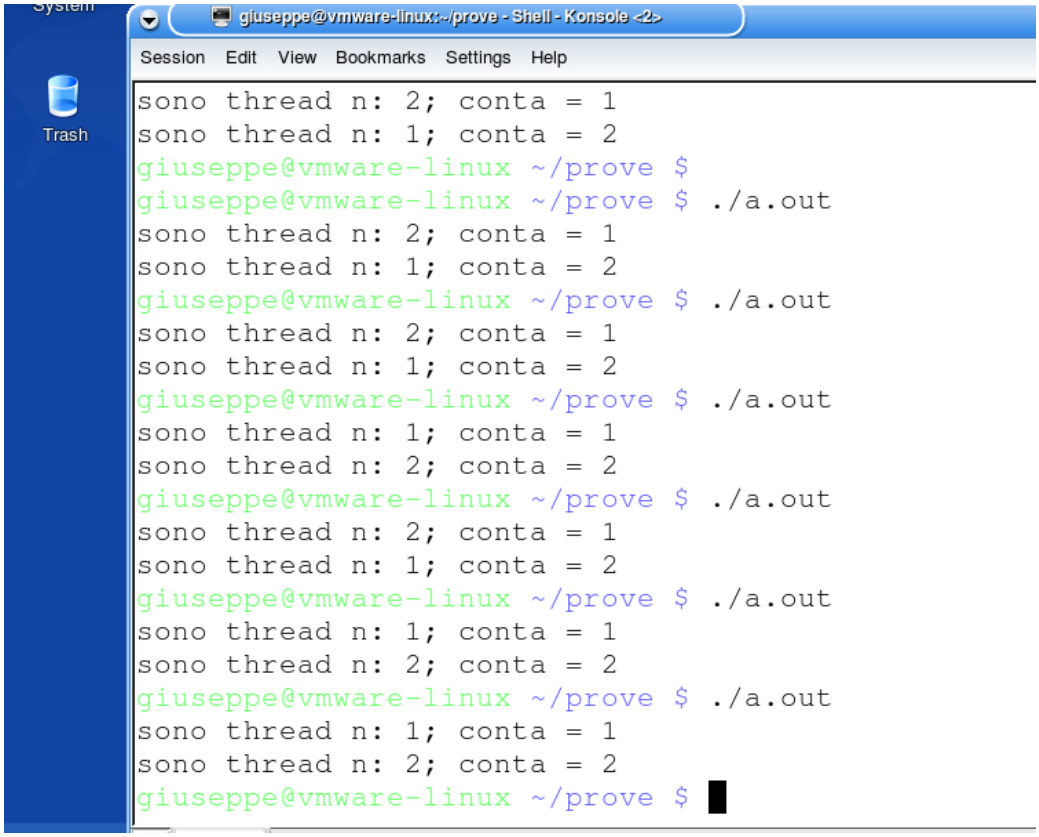


Figura 2.4

Un comportamento simile si sarebbe ottenuto anche dichiarando *conta* come variabile globale. La parte iniziale di una nuova versione del programma, che definisce conta come variabile globale, è mostrata in figura 2.5.

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

int conta =0; // conta è una variabile globale

void * tf1(void *tID)
{
conta++;
printf("sono thread n: %d; conta = %d \n",(int) tID, conta);

// ... prosegue come nel programma precedente ...
```

Figura 2.5

In tutti gli esempi precedenti il thread principale (la funzione main( )) aspettava la terminazione dei thread che aveva creato. Possiamo chiederci cosa sarebbe accaduto se non avessimo incluso le due invocazioni di pthread_join( ) nel main( ). Un esempio di alcune esecuzioni dello stesso programma di figura 2.1 privato delle *pthread_join*( ) è mostrato in figura 2.6: si vede che quando il thread principale termina la shell riprende il controllo della finestra di comando; talvolta sono stati già eseguiti i due thread creati, talvolta ne è stato eseguito uno solo. Questo comportamento si spiega nel modo seguente:

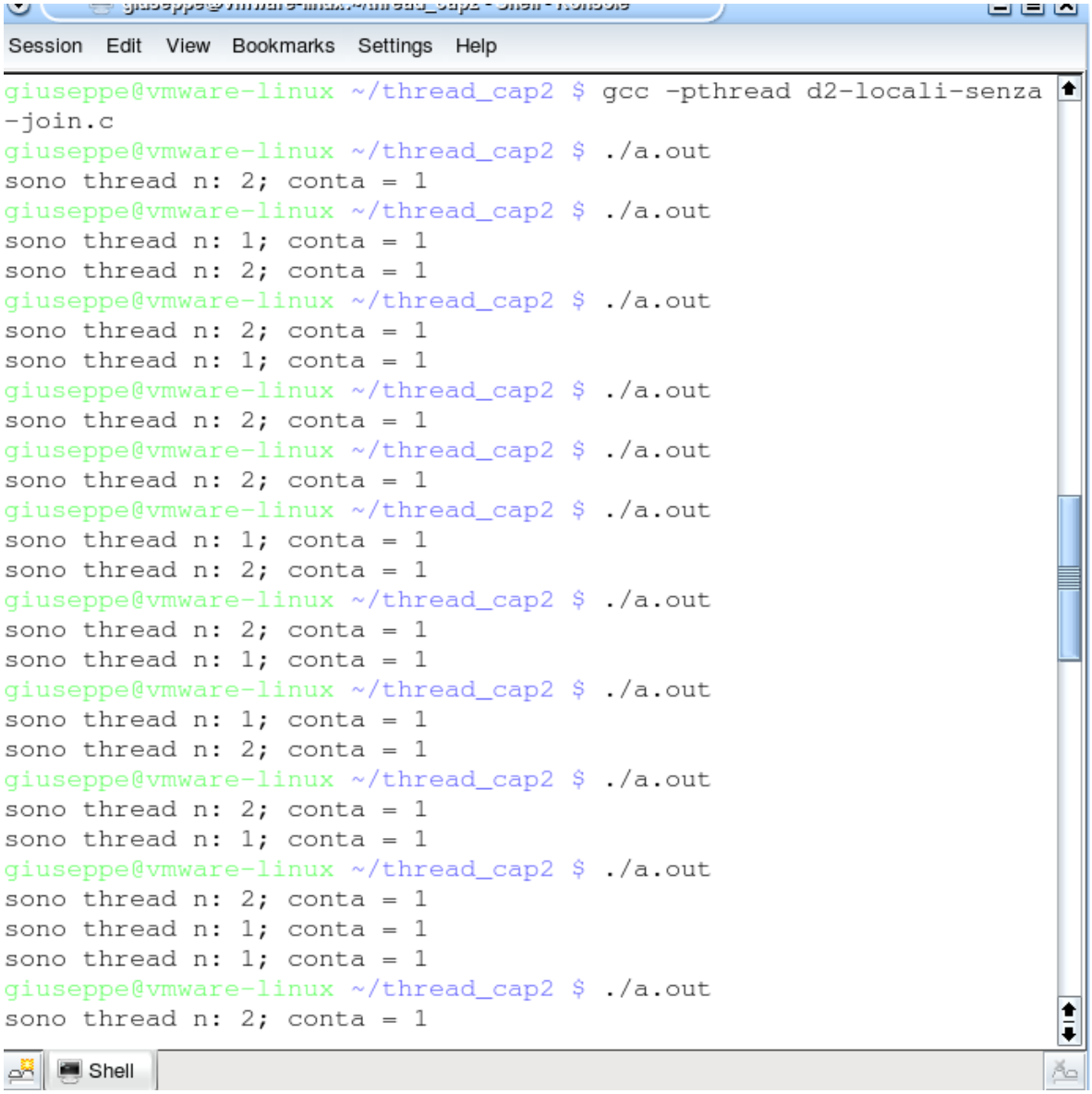
- la terminazione del processo è determinata dalla terminazione del thread principale
- quando il processo termina tutti i thread del processo vengono terminati

Il modello di programmazione dei thread è in realtà più complesso di quanto abbiamo visto, perchè esiste la possibilità di modificare il comportamento dei thread tramite la modifica degli "attributi" di un thread; questi argomenti non vengono trattati qui e noi faremo riferimento ad un contesto semplificato.

In ogni caso noi considereremo buona norma negli esempi seguenti attendere sempre la terminazione di tutti i thread secondari prima di chiudere il thread principale.

2.5 Passaggio parametri alla thread_function e restituzione dello stato di terminazione del thread

Negli esempi precedenti non si è passato nessun parametro alla thread_function e non si è restituito alcun valore al termine dell'esecuzione del thread. In ambedue i casi i Pthread prevedono la possibilità di passare un unico parametro di tipo puntatore (void *). Per passare molti parametri alla thread_function è quindi necessario creare un'opportuna struttura dati e passare un puntatore a tale struttura. Noi ci limiteremo a illustrare il passaggio di un parametro costituito da un numero intero.



Fugyra 2.6 Esecuzioni del programma di figura 2.1 privato delle pthread_join

In figura 2.7 è mostrato il codice di un programma nel quale il main crea un thread e passa alla thread function l'indirizzo della variabile "argomento"; la thread function copia il valore dell'argomento ricevuto nella variabile "i", poi incrementa tale variabile e restituisce il valore di tale variabile nell'istruzione return. Il thread principale (main) attende la terminazione del thread secondario e riceve il risultato della return nella variabile "thread_exit". Si può osservare che i dettagli programmativi sono complicati dall'esigenza di numerosi recasting dovuti al tipo di parametri richiesto, ma il principio di funzionamento è molto semplice. Il risultato

dell'esecuzione del programma è mostrato nella stessa figura e non contiene particolarità degne di nota.

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


void * tf1(void *arg)
{
int i = * ((int*) arg);
printf("sono thread_function: valore argomento= %d \n", i);
i ++;
return (void *) i;
}

int main(void)
{
int argomento=10;
int thread_exit;
pthread_t tID1;
pthread_create(&tID1, NULL, &tf1, &argomento);

pthread_join(tID1, (void *) &thread_exit);
printf("sono il main; thread_exit= %d\n", thread_exit);
return 0;
}
```

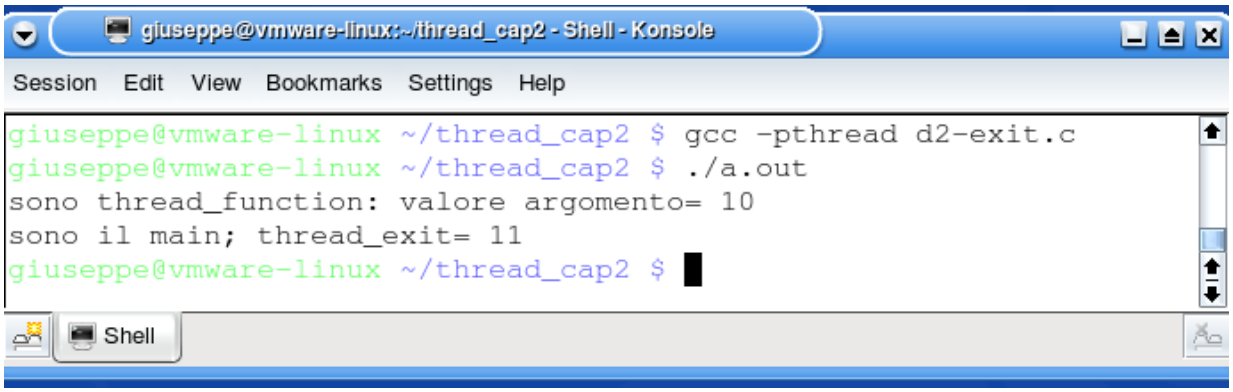


Figura 2.7 – Un programma che passa un parametro a un thread e il risultato della sua esecuzione

2.6 Uso dei thread o dei processi per realizzare il parallelismo

In alcuni programmi che potrebbero beneficiare di un'esecuzione concorrente può essere difficile decidere se creare diversi processi oppure diversi thread in un unico processo. Alcuni criteri generali sono i seguenti:

- Efficienza: La copia di memoria per un nuovo processo costituisce un onere nella creazione di un processo rispetto alla creazione di un thread. In generale i thread sono più efficienti dei processi
- Protezione: Un thread che contiene errori può danneggiare altri thread nello stesso processo; invece un processo non può danneggiarne un altro. I processi sono più protetti uno dall'altro
- Cambiamento di eseguibile: Un thread può eseguire solamente il codice della funzione associata, che deve già essere presente nel programma eseguibile del processo; invece un processo figlio può, tramite exec, sostituire l'intero programma eseguibile
- Condivisione dei dati: La condivisione di dati tra thread è molto semplice, tra processi è complicati (richiede l'impiego di meccanismi di comunicazione tra processi (IPC) che non trattiamo in questo testo).

Analizziamo ora alcuni esempi.

Consideriamo la realizzazione di un'interfaccia grafica, nella quale le diverse funzioni invocabili devono strettamente cooperare lavorando su strutture dati comuni.

Nelle interfacce grafiche ogni azione svolta da un utente costituisce un evento che il programma deve gestire. Esempi di eventi sono quindi i click del mouse, le selezioni da menu, la pressione di tasti della tastiera, ecc... Tipicamente il programma applicativo che gestisce l'interfaccia grafica contiene una diversa funzione per ogni possibile evento; ogni azione dell'utente causa quindi l'invocazione della funzione associata all'evento provocato da tale azione.

In un modello di esecuzione sequenziale una funzione invocata in base a un evento deve terminare prima che il sistema possa invocare un'altra funzione. Se tale funzione non termina abbastanza velocemente, perchè si tratta di una funzione complessa che deve ad esempio leggere dati da dispositivi periferici, eventi successivi

resteranno in attesa di essere serviti e l'utente avrà la sensazione di un'interfaccia che non risponde prontamente alle sue azioni.

Per eliminare questo difetto possiamo adotare un modello di esecuzione parallela delle diverse funzioni; in questo modo il sistema potrà mettere in esecuzione una funzione associata ad un evento successivo anche prima che sia terminata l'esecuzione della funzione precedente: il risultato è che l'interfaccia continua a rispondere prontamente all'utente anche se alcune delle funzioni invocate sono lente. In questo contesto l'uso dei thread sembra il più indicato.

Se riconsideriamo gli esempi di parallelismo trattati nel capitolo precedente, in alcuni casi è evidente quale dei due modelli, processi o thread, è più adatto, ma in altri casi la scelta è controversa e richiede un'analisi più approfondita.

Sicuramente per eseguire diversi programmi indipendenti il modello a processi separati è l'unico adottabile; in caso contrario i programmi potrebbero interferire tra loro producendo risultati indesiderati.

Nel caso dei server la situazione è invece controversa; a favore dei thread abbiamo il minor costo di creazione di un thread rispetto a un processo, ma la scelta dipenderà in generale anche dal grado di cooperazione necessario tra servizi diversi e dal grado richiesto di sicurezza che un servizio non possa danneggiare altri servizi.