Programmazione concorrente tramite threads

Matteo Dessalvi

Sommario

- Cosa è la programmazione concorrente?
- Definire i processi.
- Programmazione multiprocesso.
- Vantaggi e svantaggi.
- Cosa sono i threads?
- Programmazione multithreading.
- Vantaggi e svantaggi.

Linux box: strumenti a bordo

- Distribuzioni utilizzate: Mandrake 9.1 e Slackware 10.
- Kernel 2.4.25 (va comunque bene uno qualunque dei kernel stabili della serie 2.4.x). Kernel 2.6.9.
- Gcc 3.2.2, Glibc 2.3.1 o superiori.
- Gdb 5.3 o superiore, con supporto al debug di applicazioni multi-thread.

Cosa è la programmazione concorrente?

Con questo termine si indica l'insieme di tecniche e di strumenti necessari a poter supportare più attività eseguite simultaneamente da una applicazione software.

Questa è una caratteristica dei cosidetti sistemi multiprogrammati.

Caratteristiche della multiprogrammazione

- Più utenti possono accedere contemporaneamente ad un sistema informatico.
- Un solo utente potrà eseguire più programmi simultaneamente.
- Un singolo programma sarà in grado di scomporre la propria attività in più attività concorrenti.

Processi

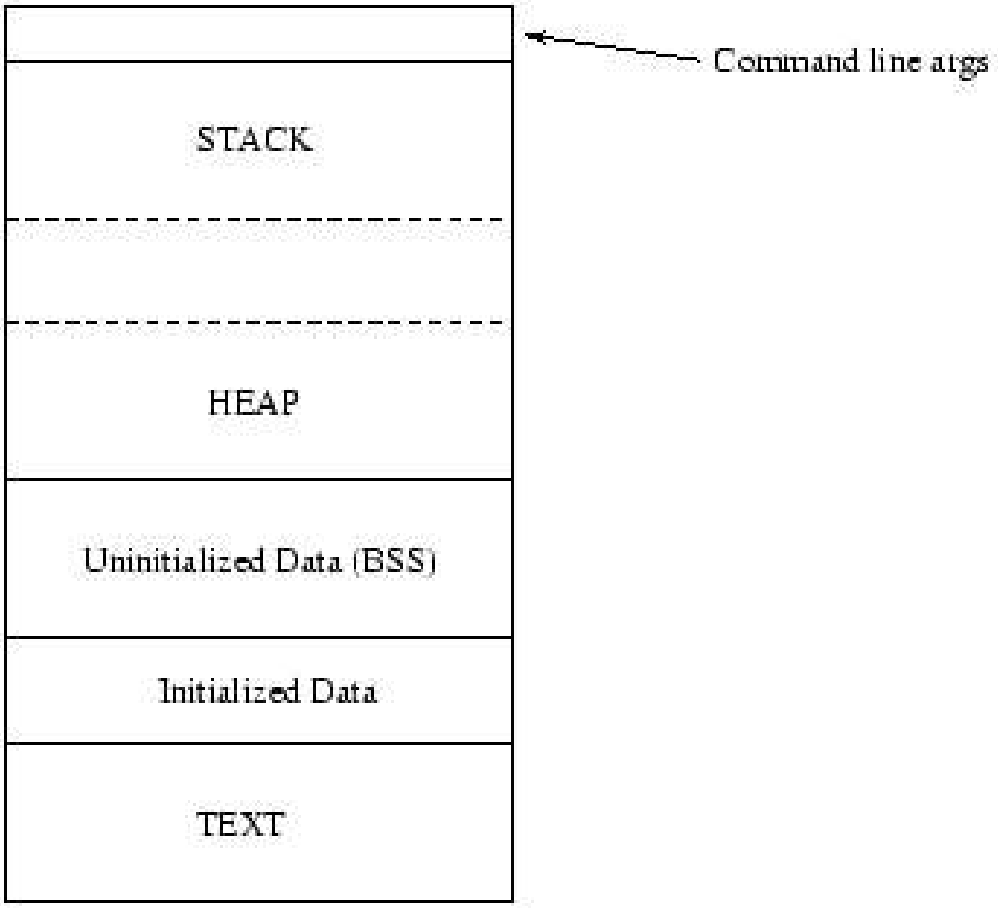
La programmazione concorrente è basata sul concetto di processo.
In un sistema Unix, un processo viene considerato l'unità di esecuzione di base.

Ogni processo ha il suo spazio di indirizzamento privato, il suo stack ed il suo heap.

Caratteristiche di un processo

- Un program counter (PC).
- Un corpo (ossia il codice che viene eseguito).
- Uno spazio di indirizzamento privato (diviso in zona dati, stack e heap).
- Una tabella di descrittori di files.
- Un descrittore di processo (PID).
- Una tabella dei segnali.

Struttura di un processo



Multiprocesso: padri e figli

Nei sistemi Unix il multitasking viene implementato tramite la system call fork(). Questo significa che il processo padre (che ha chiamato la fork) verrà “diviso” in due.

Il processo figlio erediterà tutte le caratteristiche del processo padre: i suoi dati, i file descriptors, il codice.

Vantaggi e svantaggi

- La programmazione multiprocesso permette l'esistenza dei sistemi multitasking.
- La creazione di nuovi processi porta a due problemi principali:
- Un numero elevato di processi in esecuzione può causare un elevato sovraccarico nella macchina.
- La condivisione dei dati è intrinsecamente complessa.

Cosa sono i threads?

Si può definire un thread come un flusso di esecuzione, non necessaria-mente legato ad uno spazio di indirizzamento privato.

A differenza dei processi, quando un thread viene creato, condivide il suo spazio di memoria con tutti gli altri threads che fanno parte del processo.

All'interno

Un thread è composto essenzialmente da tre componenti:

- un program counter privato
- uno stack
- una tabella dei segnali

La creazione di un thread risulta quindi meno dispendiosa rispetto a quella di un processo.

Threads POSIX sotto Linux

La libreria pthread (libpthread) mette a disposizione tutte le funzioni necessarie a creare, distruggere e manipolare i threads ed è da tempo parte delle glibc.

Implementa lo standard POSIX 1003.1c, anche se non è del tutto conforme ad esso per quanto riguarda la gestione dei segnali.

Creazione di un thread

Utilizziamo le funzioni definite nell'header *pthread.h* della libreria Pthread.
La chiamata per creare un thread è definita da questo prototipo:

```
pthread_create(pthread_t ThID,
               pthread_attribute ATTR,
               void* StartRoutine,
               void ARG)
```

Argomenti di pthread_create

- ThID : è l'id del thread creato e costituisce un identificativo univoco mentre esso è in uso. In particolare pthread_t è un tipo di dato opaco.
- ATTR: attributo del thread appena creato. Può indicarne le caratteristiche riguardanti operazioni di join e detach, oppure riferirsi allo scheduling.
- StartRoutine: è la routine di partenza del thread e riceve ARG per argomento.

Il primo programma

Il programma che vediamo esegue la stampa di una stringa sullo schermo, creando due thread.

I threads ricevono come argomento una parola della stringa, la stampano con la funzione printf(), ed all'uscita della funzione terminano il loro compito.

Hello world!

matteo@neptune: /home/matteo/LinuxDay/source - Shell - Konsole

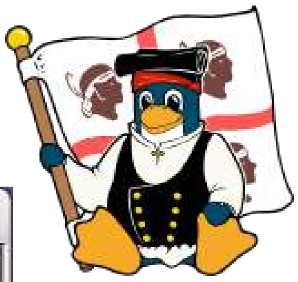
Session Edit View Bookmarks Settings Help

```
/* "Hello world" threads! */

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

void print_func(void *); /* prototipo */

main(void)
{
 pthread_t thread1, thread2;
 char *msg1 = "Hello ";
 char *msg2 = "world!";

 /* Creazione del primo thread: */
 pthread_create (&thread1, NULL, (void *)&print_func, (void*) msg1);

 /* Creazione del secondo thread: */
 pthread_create (&thread2, NULL, (void *)&print_func, (void*) msg2);

 printf("\n");

 exit(0);
}
```

1,1 Top

GULCh – Gruppo Utenti Linux Cagliari

Funzione di stampa



matteo@neptune: /home/matteo/LinuxDay/source - Shell - Konsole

Session Edit View Bookmarks Settings Help

```
/* Funzione di stampa */
void print_func(void *ptr)
{
 char *message;
 message = (char *) ptr;
 printf("%s", message);
}
~
```

36,1 Bot

Compilazione

Lanciamo il compilatore da linea di comando:

```
gcc hellopth.c -o hello -lpthread
```

Il flag -l è una opzione per il linker che consentirà di collegare il programma con la libreria pthread.

In esecuzione

Lanciamo il programma da cmd line:

./hello

Otteniamo in output il messaggio:

Hello world!

Notiamo che la funzione di partenza di entrambi i thread, al momento della creazione, è la print_func(). Un thread giunge al termine quando lascia la sua funzione iniziale.

Debolezze del codice

- I threads vengono eseguiti in modo concorrente. Nel caso precedente non esite alcuna garanzia che il thread1 venga eseguito prima del thread2! Di conseguenza l'output potrebbe anche essere: world! Hello
- Al termine della funzione main() viene chiamata la funzione exit(). Se il 'main thread' esegue questa funzione prima del 'child thread', non verrà stampato alcun output.

Possibili rimedi

- Per eliminare il problema della precedenza nell'esecuzione, è possibile usare la funzione sleep(). Ma essa si riferisce ai processi e dunque non funzionerebbe in modo corretto con i threads.
- Per evitare i problemi con la exit(), è possibile utilizzare la pthread_exit(), che termina esplicitamente un thread, ma non l'intero processo.

Thread joining

Il problema della sincronizzazione del precedente programma non trova alcuna soluzione soddisfacente.
Ci serviremo allora dell'operazione di joining di un thread.

Questa operazione consiste nel bloccare il thread chiamante fino a che il thread (ThID) specificato non termina.

Uso di pthread_ join()

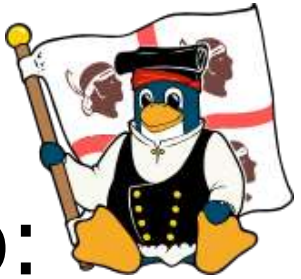
Rifacendoci al programma precedente possiamo aggiungere in coda alla main():

.....

```
pthread_ join (thread1, NULL);
pthread_ join (thread2, NULL);
```

Il secondo argomento della funzione può, eventualmente, contenere il valore di ritorno del thread.

Attributi di un thread



I thread maggiormente utilizzati sono:

Joinable threads: non vengono rimossi quando il loro compito termina, ma attendono che un altro thread chiami la pthread_join().

Detached threads: vengono rimossi istantaneamente quando terminano. Non è possibile effettuare operazioni di join con threads di questo tipo.

Detached Threads

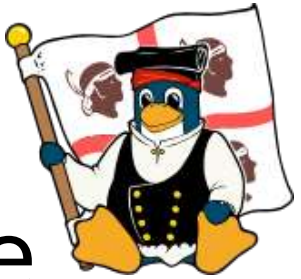
matteo@neptune: /home/matteo/LinuxDay/source - Shell - Konsole

Session Edit View Bookmarks Settings Help

```
/* SCHELETRO DI UN PROGRAMMA CHE USA I DETACHED THREADS */

#include <pthread.h>

void* thread_function(void* thd_arg)
{
  /* Svolge i suoi compiti.... */
}

int main(void)
{
  pthread_attr_t attr;
  pthread_t thread;

  /* Inizializzazione e settaggio degli attributi: */
  pthread_attr_init (&attr);
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);

  pthread_create (&thread, &attr, &thread_function, NULL);

  /* Codice in main()... */

  /* Nessun bisogno di effettuare il join*/
  return 0;
}

~
"skeldetach.c" 26L, 518C                                    23,42         All
```

GULCh – Gruppo Utenti Linux Cagliari

Stati di un thread



- READY: il thread è pronto per essere eseguito, ma si trova in attesa di un processore, oppure può essere stato appena sbloccato od essere stato “interrotto” da un altro thread.
- RUNNING: il thread è in esecuzione.
- BLOCKED: il thread non è in esecuzione poichè sta aspettando il completarsi di qualche operazione o la disponibilità di una risorsa.

Race conditions

E' impossibile conoscere con assoluta precisione quando un thread verrà posto in esecuzione. Questa incertezza non deve quindi influire durante l'esecuzione del programma.

Programmi che funzionano solo se un thread viene eseguito prima o più spesso di altri contengono delle *race conditions*.

Mettiamoci nei guai....

matteo@neptune: /home/matteo/LinuxDay/source - Shell - Konsole

Session Edit View Bookmarks Settings Help

```
#include <malloc.h>

struct job {
            struct job* next; /* puntatore al prossimo elemento in coda */

            /* Altri campi eventuali... */
          };

/* Lista concatenata di jobs pendenti: */
struct jobs* job_queue;

/* Elabora i jobs fino a che la coda non sarà vuota: */
void thread_function(void *arg)
{
  while (job_queue != NULL)
   {
    struct job* next_job = job_queue; /* Preleva il primo job disponibile */

    job_queue = job_queue->next; /* La coda scorre in avanti */

    process_job (next_job); /* Viene svolto il job */

    free(next_job); /* Libera la memoria */
   }
  return NULL;
}
~
                                                    2,0-1         All
```

GULCh – Gruppo Utenti Linux Cagliari

Supponiamo che...

Due thread finiscano il proprio lavoro nello stesso istante, ma rimanga un job da elaborare. Può accadere che:

Thread 1: verifica se la coda è vuota. Non lo è, quindi inizia il lavoro, ma viene bloccato dal kernel.
Thread 2: va in esecuzione e, trovando la coda non vuota, elabora il job.

Thread 1 e 2 elaborano lo stesso job!

Uno alla volta

Ciò di cui abbiamo bisogno è che le operazioni di controllo della consistenza della coda vengano rese *atomiche*.

Ovvero sarà impossibile metterle in pausa oppure interromperle. Per far questo ci avvarremo del supporto fornito dal sistema operativo.

MUTual EXclusion locks

I *mutex* sono simili alle serrature: il primo thread che ha accesso alla coda dei lavori lascia fuori gli altri thread, fino a che non ha portato a termine il suo compito.

I mutex verranno piazzati nelle sezioni di codice dove vengono condivisi i dati.

pthread_mutex

Un mutex viene dichiarato come variabile di tipo pthread_mutex_t e passato come puntatore a pthread_mutex_init:

```
pthread_mutex_t  mutex;
pthread_mutex_init (&mutex, NULL);
```

Oppure con:

```
pthread_mutex_t  mutex =
    PTHREAD_MUTEX_INITIALIZER
```

I mutex in pratica

matteo@neptune: /home/matteo/LinuxDay/source - Shell - Konsole

Session Edit View Bookmarks Settings Help

```
#include <pthread.h>
#include <malloc.h>
/* ... */
pthread_mutex_t job_q_mutex = PTHREAD_MUTEX_INITIALIZER;
void * thread(void * arg)
{
 while (1)
 {
  struct job* next_job;
  pthread_mutex_lock (&job_q_mutex); /* Inseriamo il mutex nella coda dei jobs */

  if(job_queue == NULL) /* Ora è possibile verificare con sicurezza se la coda è vuota */
    next_job = NULL;
  else {
        next_job = job_queue;
        job_queue = job_queue->next;
       }
  pthread_mutex_unlock(&job_q_mutex); /* Mutex unlock! */

  if (next_job == NULL) /* Se la coda è vuota la funzione termina */
    break;

  process_job (next_job); /* Esegue il job */
  free (next_job);      /* Libera la memoria */
 }
 return NULL;
}
```

27,1 All

GULCh – Gruppo Utenti Linux Cagliari

Mutex deadlocks

Una situazione di deadlock accade quando uno o più thread sono bloccati in attesa di un evento che non si verificherà mai.

Il fast mutex, creato di default sui sistemi GNU/Linux , è soggetto a questo tipo di problema.

Tipi di mutex

Fast mutex: può causare un deadlock quando un thread tenta di impostare lo stesso mutex per due volte di seguito.

Recursive mutex: non causa deadlock. Ricorda quanti lock ha applicato e si sblocca solo dopo altrettanti unlock.

Error check mutex: il secondo lock consecutivo di un mutex genera un errore (disponibile solo su Linux).

Mutex error checking

```
int type =
PTHREAD_MUTEX_ERRORCHECK_NP;
pthread_mutexattr_t attr;
pthread_mutex_t mutex;

pthread_mutexattr_init (&attr)
pthread_mutexattr_settype (&attr, type);
pthread_mutex_init (&mutex, &attr);
pthread_mutexattr_destroy(&attr);
```

Semafori

Se i threads lavorano molto rapidamente potrebbero svuotare la coda e terminare le loro operazioni prima che essa sia di nuova piena.

Quindi abbiamo bisogno di un sistema che blocchi i threads fino a che nella coda non vengano immessi nuovi jobs. Entrano in scena i *semafori*.

Cosa sono i semafori?

I semafori sono contatori usati per sincronizzare le operazioni di thread multipli.

GNU/Linux garantisce, anche per queste primitive, l'atomicità. Quindi, ogni modifica o check del valore di un semaforo può essere effettuata senza sollevare race conditions.

Operazioni di base

Wait: decrementa di 1 il valore del semaforo. Se il valore è zero, l'operazione viene bloccata fino a che il valore non ridiviene positivo.

Post: incrementa il valore di 1 del semaforo. Se il valore precedente era zero, e vi sono threads in attesa, uno di questi viene sbloccato e la sua operazione di wait è completata.

Usiamo i semafori

matteo@neptune: /home/matteo/LinuxDay/source - Shell - Konsole

Session Edit View Bookmarks Settings Help

```
#include <malloc.h>
#include <semaphore.h>
#include <pthread.h>
/* ... */
pthread_mutex_t job_q_mutex = PTHREAD_MUTEX_INITIALIZER; /* Mutex per la coda */
sem_t job_q_count; /* Un semaforo che conta il numero di jobs nella coda */

void initialize_job_queue()
{
 job_queue = NULL;
 sem_init(&job_queue_count,0, 0); /* Inizializza il semaforo che conta i jobs in coda.
                                     Il valore iniziale è zero */
}
void * thread_function(void *arg)
{
 while (1) {
  struct job* next_job;
  sem_wait (&job_queue_count); /* Operazione di wait sulla coda dei jobs */
  pthread_mutex_lock (&job_queue_mutex);
  /* Possiamo eseguire operazioni sulla coda perche' sappiamo che il semaforo non
   * darà il via ad altri threads se la coda è vuota */
  /* ... */
  pthread_mutex_unlock (&job_queue_mutex);
  /* Esecuzione del lavoro e rilascio delle risorse occupate */
  /* ... */
 }
}
                                                              23,1          All
```

GULCh – Gruppo Utenti Linux Cagliari

Aggiunta di un nuovo job

matteo@neptune: /home/matteo/LinuxDay/source - Shell - Konsole

Session Edit View Bookmarks Settings Help

```
/* Aggiunge un nuovo job all'inizio della coda */

void job_queue ( /* eventuale passaggio di dati specifici */ )
{
 struct job* new_job;
 /* Allocazione di un nuovo job: */
 new_job = (struct job *) malloc(siezof(struct job));

 /* Settaggio di altri campi della struttura.... */

 pthread_mutex_lock (&job_queue_mutex); /* Set mutex */

 /* Piazziamo il nuovo job alla testa della coda: */
 new_job->next = job_queue;
 job_queue = new_job;

 /* Operazione di post sul semaforo per indicare che un nuovo job è disponibile.
  * In questo modo un thread bloccato sul semaforo viene sbloccato ed elaborare
  * il nuovo job: */
 sem_post (&job_queue_count);

 pthread_mutex_unlock (&job_queue_mutex); /* Unlock mutex! */
}
```

77,2-8 85%

GULCh – Gruppo Utenti Linux Cagliari

Condition variables

Tramite le variabili di condizione si rende possibile una maggiore diversificazione nei criteri che i thread devono soddisfare per poter essere eseguiti.

Linux garantisce che i threads bloccati su una condizione verranno sbloccati quando quest'ultima cambierà.

Condition variables e mutex

Una variabile condizione non fornisce la mutua esclusione. C'è bisogno di un mutex per poter sincronizzare l'accesso ai dati.

Pthreads garantisce che le operazioni di unlock e di wait (o signal) vengano rese atomiche, in modo da evitare che si verifichino race conditions tra le due operazioni durante lo scheduling di due threads.

Utilizzare le conditions

Inizializzazione:
pthread_cond_init(pthread_cond_t *COND, NULL) - (nessun attributo).

Pthread_cond_signal(): segnala una variabile condizione.

Pthread_cond_wait(): blocca il thread chiamante finchè la variabile condizione non è segnalata.

Conditions variables

matteo@neptune: /home/matteo/LinuxDay/source - Shell - Konsole

Session Edit View Bookmarks Settings Help

```
#include <pthread.h>

int thread_flag;
pthread_cond_t thread_flag_cv;
pthread_mutex_t thread_flag_mutex;

void init_flag()
{
 pthread_mutex_init (&thread_flag_mutex, NULL); /* inizializzo la cond. variables */
 pthread_cond_init (&thread_flag_cv, NULL); /* inizializzo il mutex */
 thread_flag = 0;
}
/* Svolge ripetutamente qualche compito mentre il flag è settato ad 1, mentre si blocca
 * quando il flag è settato a zero. */
void * thread_function(void * thread_arg)
{
  while(1) {
   pthread_mutex_lock(&thread_flag_mutex); /* inserisce il mutex prima di accedere al flag */
   while(!thread_flag)
   pthread_cond_wait(&thread_flag_cv, &thread_flag_mutex);
   pthread_mutex_unlock(&thread_flag_mutex);
   /* ... Svolge qualche compito ... */
  }
  return NULL;
}
```

-- INSERT -- 16,1 Top

GULCh – Gruppo Utenti Linux Cagliari

matteo@neptune: /home/matteo/LinuxDay/source - Shell - Konsole

Session Edit View Bookmarks Settings Help

```
void set_thread_flag (int flag_value)
{
 pthread_mutex_lock (&thread_flag_mutex); /* lock mutex */
 thread_flag = flag_value;
 pthread_cond_signal (&thread_flag_cv);
 pthread_mutex_unlock (&thread_flag_mutex); /* unlock mutex */
}
```

-- INSERT -- 26,1 Bot

GULCh – Gruppo Utenti Linux Cagliari

Debugging

Lo standard Pthreads non definisce alcun sistema per il debug del codice.

Supporto integrato nel s.o.: puo' essere fornito uno strumento che permetta di verificare lo stato dei thread,mutex, ecc.

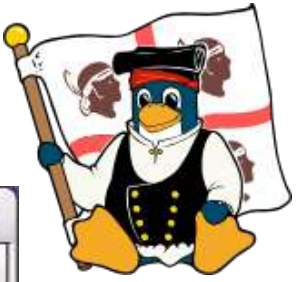
Implementazioni in user-space: è improbabile che vi sia integrazione col debugger. Al massimo si può trovare un comando di stampa per le info.

I threads sono asincroni

E' bene non scordarlo mai. Sopratutto considerando macchine che dispongono di un solo processore, non possiamo fare alcun tipo di previsione sulla velocità con la quale un nuovo thread viene creato e avviato.

Lo vediamo col programma seguente.

Superare l'inerzia...



matteo@neptune: /home/matteo/LinuxDay/source - Shell - Konsole

Session Edit View Bookmarks Settings Help

```
/* inertia.c -- L'idea dietro questo programma è verificare di quanto si può allugare il
 * tempo di creazione del nuovo thread, fino a permettere che il nuovo valore della stringa
 * settata nel main thread, dopo il ciclo for, venga cmq ignorata. */
#include <pthread.h>

void *printer_thread (void *arg)
{
    char *string = *(char**)arg;
    printf ("%s\n", string);
    return NULL;
}
int main (int argc, char *argv[])
{
    pthread_t printer_id;
    char *string_ptr;
    int i, status;

    string_ptr = "Vecchio valore";

    status = pthread_create(&printer_id, NULL, printer_thread, (void*)&string_ptr);

    for (i = 0; i < 10000000; i++); /* ciclo vuoto */

    string_ptr = "Nuovo valore";
    status = pthread_join (printer_id, NULL);
    return 0;
}
                                                              3,57          All
```

GULCh – Gruppo Utenti Linux Cagliari

Stanare le race conditions?

Le CPU eseguono i threads a differenti velocità, dipendenti dal carico.
Il timeslicing può interrompere un thread in qualunque punto e per un tempo variabile. Ricreare le condizioni per una race condition può essere impossibile!

"**Threads will run in the most evil order possible**".

Deadlocks

I deadlocks si verificano quando vi sono dei conflitti nella sincronizzazione.

In questo caso il debugging evidenzia, dopo qualche tempo, che vi sono dei threads che semplicemente stanno fermi ad aspettare, per un tempo indefinito, una qualche risorsa che per varie cause non otterrano mai.

Vantaggi del multithreading

- I threads sono molto più leggeri dei processi. Questo comporta un minor utilizzo delle risorse per crearli e per effettuare un context switch.
- I threads condividono la stessa area di memoria e lo scambio di dati è molto più semplice rispetto alle primitive IPC.
- I threads dovrebbero essere usati per programmi che richiedano“parallelismo fine”.

E svantaggi...

- La facilità nello scambio di dati può essere anche un punto debole: si possono avere race conditions sui dati condivisi.
- E' più difficile effettuare il debug a causa della difficile riproducibilità di molte situazioni di errore.
- Un thread errato può corrompere lo spazio di memoria condiviso, causando un SEGFAULT.

Modelli di implementazione

- 1:1 Ogni thread viene visto come un processo separato. Lo scheduler li tratterà come processi. I kernel Linux 2.4.x implementano questo modello.
- M:1 M threads in user space sono mappati in un solo thread in kernel-space.
- M:N M threads in user-space sono mappati in N threads in kernel-space (AIX, Solaris, IRIX e Compaq True64).

User-space vs Kernel-space

- Kernel-space: quando i threads sono implementati a questo livello, la commutazione di contesto avviene senza che l'applicazione ne sappia nulla.
- User-space: in questo caso la commutazione di contesto avviene ad opera della libreria applicativa che fornisce il supporto per i threads, senza che il kernel se ne accorga.

Il modello LinuxThreads

LinuxThreads è l'implementazione dei threads POSIX supportata da Linux a livello kernel.

Ogniqualvolta utilizziamo la chiamata *pthread_create*, ciò che stiamo facendo è utilizzare la system call *__clone()*,una versione più generica della *fork()*.

Vediamolo col programma seguente.

Threads pid

matteo@neptune: /home/matteo/LinuxDay/source - Shell - Konsole

Session Edit View Bookmarks Settings Help

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

void *thread_function (void *arg)
{
  fprintf (stderr, "child thread pid: %d\n", (int) getpid());
  /* Loop infinito */
  while(1);
  return NULL;
}

int main(void)
{
  pthread_t thread;
  fprintf(stderr, "main thread pid: %d\n", (int) getpid());
  pthread_create (&thread, NULL, &thread_function, NULL);
  /* Loop infinito */
  while(1);
  return 0;
}
~
~
~
~
~
~
                                                    21,1          All
```

GULCh – Gruppo Utenti Linux Cagliari

Contolliamo l'output

Compiliamo il programma lanciando il gcc:

```
gcc pidthd.c -o pid -lpthread
```

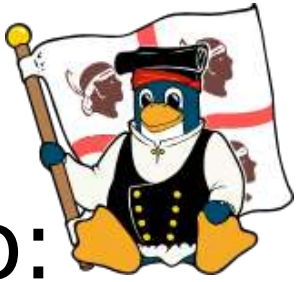
Eseguiamo il programma in background:

```
./pid &
```

L'output sarà:

```
[1] 2092
main thread pid 2092
child thread pid 2094
```

Output di ps



Lanciando da cmd line ps -x vediamo:

```
2092   pts/2     R   1.46   ./pid
2093   pts/2     S   0.00   ./pid
2094   pts/2     R   1.46   ./pid
```

Il thread 2093 è detto manager thread ed è parte dell'implementazione interna dei LinuxThreads. Viene creato la prima volta che il programma fa ricorso alla pthread_create.

Signal handling & threads

Visto che su GNU/Linux i threads sono implementati come processi non esiste ambiguità nello spedire o ricevere segnali.

Di solito, i segnali spediti dall'esterno al programma vengono inviati al manager thread e da lì vengono ridistribuiti ai threads destinatari.

Problemi con LinuxThreads

- Limite nel numero di threads per processo (8192).
- Il thread manager deve coordinare tutte le attività tra i threads di un processo.
- Le operazioni di SIGNAL HANDLING vengono effettuate per processo e non per thread.

Nuove librerie

- NGPL (Next Generation Posix Library): inizialmente sviluppata all'interno dell'IBM, il suo sviluppo si interruppe quando la NPTL dimostrò la sua superiorità.
- NPTL (Native Posix Thread Library): è la nuova thread library, introdotta a partire dai kernel di sviluppo 2.5.x. Sviluppata da Ingo Molnar e Ulrich Drepper, è possibile leggere alcune sue caratteristiche su: http://people.redhat.com/drepper

Caratteristiche della NPTL

- Maggiore aderenza allo standard POSIX.
- Low startup cost : creare nuovi threads avrà un costo molto basso in termini di risorse utilizzate.
- La nuova implementazione funzionerà bene anche con un numero elevato di processori.
- Scalabilità software : idealmente, non vi sarà limite nel numero di threads creati.
- Supporto ai sistemi NUMA (Non-Uniform Memory Architectures).

Scelte di design della NPTL

La nuova libreria si basa sempre su un sistema 1:1. Questa decisione è stata presa per ragioni di performance e di semplicità nella progettazione.

Un sistema M:N avrebbe richiesto due scheduler operanti per gestire il context switch e le operazioni fra threads in user space e threads in kernel space.

Modifiche ai kernel 2.6.x

- Estensione della system call clone() per ottimizzare le operazioni di creazione e terminazione dei threads, senza il supporto del thread manager.
- La gestione dei segnali per processi multithread è implementata nel kernel.
- Viene introdotta una variante della syscall exit(), exit_group(). Consente di terminare l'intero processo.

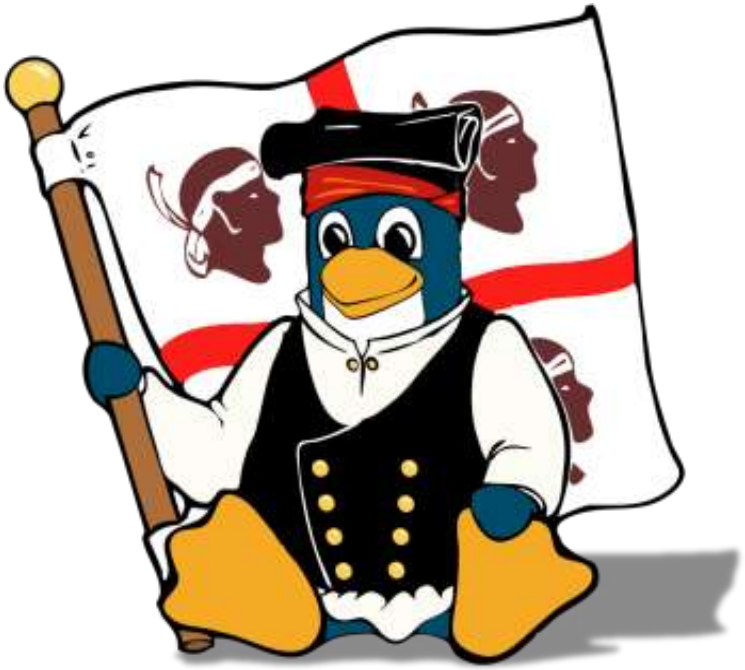
Info sugli strumenti utilizzati

- Info sul GDB: (http://sources.redhat.com/gdb)
- Info specifiche su tutti gli strumenti GNU: (http://www.gnu.org)
- Lanciando da linea di comando:
- info libc (le informazioni sulla lib C)
- info gdb (le informazioni sul debugger)
- info gcc (tutte le informazioni sul compilatore).

Links utili

- Su thread e dintorni (not always Linux related!): comp.programming.threads
- FAQ di comp.programming.threads: http://www.lambdacs.com/cpt/FAQ.html
- Tests sulla libreria NPTL: http://nptl.bullopensource.org/
- Tutorial programmazione Threads: www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html

The end



- Grazie a tutti per l'attenzione.
- Appuntamento al LinuxDay 2005!
- Happy hacking!