Lezione 5

Processi e thread

Sistemi operativi

3 aprile 2012

Marco Cesati

System Programming Research Group
Università degli Studi di Roma Tor Vergata

Di cosa parliamo in questa lezione?

Relazione tra processi e thread, e relative API

1. Gestione dei processi
2. I thread
3. Supporto del nucleo e modelli di implementazione
4. La libreria pthreads
5. Implementazione dei thread in Linux

Creazione di processi ed esecuzione dei programmi

- In alcuni SO la creazione di un processo e l'esecuzione di un programma sono realizzati atomicamente dalla medesima API
  - Operazione chiamata *spawn*
  - Esempio: in MS Windows si utilizza `CreateProcess()`
- In altri SO creazione di un processo ed esecuzione di un programma sono due operazioni diverse
- Nei SO della famiglia Unix si utilizzano:
  - `fork()` per creare un nuovo processo
  - `execve()` o simile per eseguire un programma

Creazione di un processo in Linux

Nei SO della famiglia Unix la creazione di un nuovo processo si realizza principalmente tramite la chiamata di sistema `fork()`

```
pid_t fork(void);
```

- Se la chiamata di sistema fallisce, `fork()` restituisce il valore -1 e la variabile globale `errno` è impostata con il codice d'errore
- Se la chiamata di sistema ha successo, `fork()` restituisce due differenti valori:
  - Al processo che ha invocato `fork()` viene restituito l'identificatore (PID) del processo figlio appena creato
  - Al processo figlio appena creato viene restituito il valore 0
- In tutti i casi l'esecuzione prosegue con l'istruzione successiva alla invocazione di `fork()`

Creazione di un processo in Linux (2)

Il processo *figlio* creato per mezzo di `fork()` è un "duplicato" del processo *padre*:

- Identico programma (istruzioni macchina)
- Identico stato della memoria
- Medesimi file aperti
- ...

Tuttavia il figlio si distingue dal padre per alcuni dettagli, ad esempio:

- L'identificatore (PID) del figlio è diverso da quello del padre
- Il figlio memorizza nel proprio PCB il PID del padre
- Le statistiche di utilizzo delle risorse del figlio sono azzerate
- Il figlio non eredita diversi meccanismi di notificazione asincrona di eventi

Esempio d'uso di fork()

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    pid_t p = fork();
    int v = 0;
    if (p == 0)
        printf("Child process!\n");
    else
        printf("Parent process!\n");
    ++v;
    return v;
}
```

Quale valore viene restituito dal programma?

I due processi terminano restituendo entrambi il valore 1

Eseguire un programma in Linux

I SO della famiglia Unix definiscono diverse API per caricare in memoria ed eseguire un nuovo programma

Essenzialmente le varie API differiscono per

- il modo in cui si cerca il file eseguibile
- il modo in cui si codificano gli argomenti
- il modo in cui si imposta l'ambiente di esecuzione

API	Programma	Argomenti	Ambiente
`execve()`	percorso	vettore	esplicito
`execle()`	percorso	lista	esplicito
`execlp()`	nome+`PATH`	lista	implicito
`execvp()`	nome+`PATH`	vettore	implicito
`execv()`	percorso	vettore	implicito
`execl()`	percorso	lista	implicito

La chiamata di sistema execve()

In Linux l'API `execve()` corrisponde ad una chiamata di sistema:

```
int execve(const char *pathname,
  char *const argv[], char *const envp[]);
```

- Se la chiamata di sistema fallisce, restituisce il valore -1 e viene impostata la variabile globale `errno`
- Se la chiamata di sistema ha successo, la funzione non termina
  - L'intero processo che esegue `execve()` viene "ricostruito" con le informazioni nel file eseguibile
- Quasi tutte le informazioni relative al programma che ha eseguito `execve()` vengono perse
- Possibili eccezioni: i file aperti ed i segnali ignorati

Esempio d'uso di execve()

```
#include <unistd.h>
int main(int argc, char *argv[])
{
    execve(argv[1], argv+1, NULL);
    return 1;
}
```

- Il programma esegue il file eseguibile il cui percorso è indicato dal primo argomento
- Gli argomenti successivi del programma sono passati al nuovo programma

Quale valore viene restituito dal processo?

In caso di errore in `execve()`, il valore 1.
In caso di successo, il valore restituito dal nuovo programma

Applicazioni multithread

In una applicazione tradizionale, il programmatore definisce un unico *flusso di esecuzione* delle istruzioni:

- La CPU esegue istruzioni macchina in sequenza
- Il flusso di esecuzione "segue" la logica del programma (cicli, funzioni, chiamate di sistema, gestori di segnali...)
- Quando il flusso di esecuzione arriva ad eseguire la API `exit()` l'applicazione termina

Le *applicazioni multithread* consentono al programmatore di definire diversi flussi di esecuzione:

- Ciascun flusso di esecuzione condivide le strutture di dati principali dell'applicazione
- Ciascun flusso di esecuzione procede in modo concorrente ed indipendente dagli altri flussi
- L'applicazione finisce solo quando tutti i flussi di esecuzione vengono terminati

Motivazioni

I calcolatori elettronici hanno un elevato parallelismo interno:

- DMA (Direct Memory Access): trasferimento dati tra memoria primaria e secondaria senza intervento della CPU
- Hyperthreading: supporto a diversi flussi di esecuzione, ciascuno con un proprio insieme di registri, che si alternano sulle unità funzionali della CPU
- Multicore: diversi core di calcolo integrati sullo stesso chip e che condividono alcune risorse (cache di 2° livello, MMU, ...)
- Multiprocessori: diverse CPU integrate sulla stessa scheda madre

È difficile scrivere applicazioni tradizionali (unico flusso) che sfruttino a fondo il parallelismo interno del calcolatore

Esempio di applicazione multithread

Un browser (client per World Wide Web) potrebbe essere costituito dai seguenti thread:

- Thread principale di controllo dell'applicazione
- Thread per l'interazione con l'utente
- Thread per la visualizzazione (rendering) delle pagine in formato HTML
- Thread per la gestione dei trasferimenti di pagine e file dalla rete
- Thread per l'esecuzione dei frammenti di script integrati nelle pagine Web
- Thread per l'esecuzione dei programmi Java, Flash, ecc.

Ciascun thread compie il proprio lavoro eseguendo un flusso di istruzioni indipendente e cooperando con gli altri thread

Vantaggi del multithreading

- Riduzione del tempo di risposta: anche se una parte dell'applicazione è bloccata in attesa di eventi esterni, un altro thread può essere eseguito per interagire con l'utente o gestire altri eventi
- Migliore condivisione delle risorse: tutti i thread di una applicazione condividono le risorse (strutture di dati in memoria, file aperti), e la comunicazione tra i thread è immediata
- Maggiore efficienza: rispetto ad una applicazione costituita da più processi cooperanti, l'applicazione multithread è più efficiente, perché il SO gestisce i thread più rapidamente
  - In Linux, creare un thread richiede 1/10 del tempo richiesto per la creazione di un processo
- Maggiore scalabilità: i thread possono sfruttare in modo implicito il parallelismo interno del calcolatore

Processi e thread

Un processo per una applicazione monothread è costituito da:

- Codice (istruzioni macchina in memoria)
- Strutture di dati (variabili globali in memoria)
- File aperti
- Contenuto dei registri della CPU (contesto)
- Posizione e contenuto dello stack UM

In una applicazione multithread alcune risorse sono comuni e condivise tra tutti i thread:

- Codice
- Strutture di dati
- File aperti

Altre risorse sono private per ciascun thread:

- Contenuto dei registri della CPU (contesto)
- Posizione e contenuto dello stack UM

Implementazione delle applicazioni multithread

Esistono principalmente due modalità di implementazione delle applicazioni multithread:

- Implementazione a livello utente
- Implementazione a livello kernel

In sostanza tutte le implementazioni sono caratterizzate dalla relazione tra:

Thread utente

il flusso di esecuzione dell'applicazione

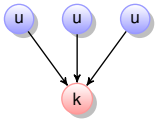
Thread kernel

l'astrazione (strutture di dati, servizi) definita all'interno del nucleo del SO per gestire un flusso di esecuzione

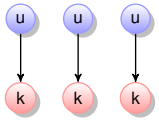
La definizione di thread utente o thread kernel non è correlata con la modalità d'esecuzione (User Mode o Kernel Mode)

Modello "da molti a uno"

- Una applicazione multithread è costituita da un singolo processo del SO
- A diversi thread utente corrisponde un singolo thread kernel
  - Un singolo processo tradizionale
- Il nucleo del SO non è coinvolto nella gestione dei flussi dell'applicazione
- L'applicazione (eventualmente usando una libreria di sistema) gestisce autonomamente i thread utente:
  - Schedula i vari flussi di esecuzione
  - Salva e ripristina gli stack UM e i contesti
- Se un thread invoca una chiamata di sistema bloccante, il processo (e quindi tutti i thread utente) vengono bloccati
- È impossibile sfruttare in modo implicito il parallelismo interno del calcolatore
- Implementazione detta "a livello utente" o *green threads*

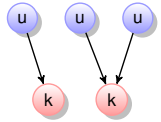


Modello “da uno a uno”



- Ciascun thread utente della applicazione corrisponde un singolo thread kernel
- Il nucleo del SO si occupa della gestione e schedulazione dei thread kernel
  - Perciò gestisce anche i thread utente
- L’applicazione utilizza le API definite in una libreria di sistema:
  - Crea e distrugge i thread utente
  - Gestisce comunicazione e sincronizzazione
- La libreria implementa i servizi richiesti dall’applicazione invocando opportune chiamate di sistema
- Ciascun thread utente può invocare chiamate di sistema bloccanti senza bloccare gli altri thread
- L’applicazione sfrutta in modo implicito il parallelismo interno del calcolatore
- Implementazione detta “a livello kernel” o *native threads*

Modello “da molti a molti”



- Gli n_u thread utente della applicazione corrispondono a n_k thread kernel ($n_k \leq n_u$)
- Il nucleo del SO si occupa della gestione e schedulazione dei thread kernel
  - Non gestisce direttamente i thread utente
- L’applicazione utilizza le API definite in una libreria di sistema:
  - Definisce il numero n_k di thread kernel
  - Crea e distrugge i thread utente
  - Mappa i thread utente sui thread kernel
- La libreria di sistema:
  - Usa chiamate di sistema per gestire i thread kernel
  - Gestisce la schedulazione e lo stack UM dei thread utente mappati sullo stesso thread kernel
  - Gestisce comunicazione e sincronizzazione

Esempi di implementazioni

- Esempi di implementazioni con modello “da molti a uno”:
  - La libreria *green threads* disponibile in Sun Solaris
  - La libreria GNU Pth (GNU Portable Threads)
  - Le librerie che implementano i thread a livello utente possono essere utilizzate con qualunque sistema operativo
- Tutti i maggiori SO oggi supportano i thread nativi, e dunque i modelli “da uno a uno” e “da molti a molti”
  - Linux, MS Windows, MacOS X tendono ad adottare il modello “da uno a uno”
  - IRIX, HP-UX, Tru64 UNIX adottano il modello “da molti a molti”
  - In ogni caso la scelta del modello “da uno a uno” piuttosto che “da molti a molti” dipende dalla libreria di thread utilizzata, e non dal sistema operativo

Librerie dei thread

- Generalmente il programmatore utilizza una libreria di sistema per realizzare una applicazione multithread
- Le API offerte dalla libreria non sono direttamente correlate con la tipologia di thread utilizzata
  - Possono esistere diverse versioni di una libreria con identiche API ma thread a livello utente oppure kernel
  - Esempio: la libreria *pthreads* (POSIX threads)
- Alcune librerie e relative API sono specifiche per un determinato sistema operativo e tipologia di thread
  - Esempio: la libreria per i thread delle API Win32
- Alcune librerie di thread e relative API sono specifiche di un linguaggio ad alto livello
  - L’uso della libreria è implicito e automatico
  - La libreria utilizza una libreria di thread di livello più basso
  - Esempio: la libreria di thread del linguaggio Java

La libreria *pthreads*

- La libreria *pthreads* è definita dallo standard POSIX (IEEE 1003.1c)
- Lo standard *definisce* le API, ma non stabilisce quale debba essere la loro implementazione in uno specifico SO
- In Linux sono coesistite tre diverse implementazioni:
  - LinuxThreads: prima implementazione, basata sul modello "da uno a uno", non più supportata
  - NGPT (Next Generation POSIX Threads): sviluppata dalla IBM, sul modello "da molti a molti", ora non più supportata
  - NPTL (Native POSIX Threads Library): ultima implementazione, più efficiente, più aderente allo standard, sul modello "da uno a uno"
- Oggi in Linux si utilizza esclusivamente NPTL

Creazione di un nuovo thread in *pthreads*

La funzione di libreria `pthread_create()` crea un nuovo thread:

```
int pthread_create(ptid, pattr, start, arg)
```

- `ptid`: puntatore a variabile di tipo `pthread_t` che conterrà l'identificatore del nuovo thread (TID)
- `pattr`: puntatore ad una variabile contenente attributi (flag) per la creazione del thread
- `start`: funzione inizialmente eseguita dal thread, con prototipo:

  ```
  void *start(void *);
  ```

- `arg`: puntatore passato come argomento a `start()`

Terminazione di un nuovo thread in *pthreads*

La funzione di libreria `pthread_exit()` termina l'esecuzione del thread che la invoca:

```
void pthread_exit(void *value_ptr);
```

- Il valore passato come argomento può essere passato ad altri thread dello stesso processo (*join*)
- La funzione viene implicitamente invocata quando la funzione iniziale `start` del thread termina
- Se viene eseguita dall'ultimo thread di un processo, il processo stesso termina con una `exit(0)`

Attesa della terminazione di un thread in *pthreads*

La funzione di libreria `pthread_join()` attende la conclusione di un thread:

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

- `tid` è l'identificatore del thread di cui si vuole attendere la terminazione
- `pret` è l'eventuale indirizzo di una variabile che riceverà il valore passato dal thread terminato in `pthread_exit()`
- Non esiste modo di indicare che si vuole attendere la terminazione di un thread qualunque

Esempio d'uso della libreria *pthreads*

```
#include <pthread.h>
long start(long v)
{
    return v+1;
}
int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid, 0, start, argc);
    pthread_join(tid, &argc);
    return argc;
}
```

NB: compilando con GCC si deve usare l'opzione `-pthread`

Quale valore viene restituito dal processo?

Il numero di argomenti con cui il programma è stato invocato maggiorato di due

Terminazione di un processo multithread

In Unix un processo viene terminato con `exit()`:

```
void exit(int status);
```

In effetti in Linux le cose sono più complicate:

- La chiamata di sistema `_exit` termina un singolo thread
- La chiamata di sistema `exit_group` termina tutti i thread di un processo
- La funzione wrapper `_exit()` esegue la chiamata di sistema `exit_group`, non `_exit`
- La funzione di libreria `exit()` invoca, alla fine, la funzione wrapper `_exit()`
- Eseguire `return` in `main()` è equivalente ad invocare la funzione di libreria `exit()`
- La funzione wrapper `_Exit()` (standard C99) è equivalente alla funzione wrapper `_exit()`
- La funzione di libreria `pthread_exit()` invoca direttamente la chiamata di sistema `_exit`

Implementazione dei thread in Linux

L'implementazione dei thread in Linux è basata sul concetto di *processo leggero* o LWP (Light Weight Process)

Un processo leggero è un processo che condivide alcune risorse selezionate con il proprio genitore

Per creare processi leggeri si utilizza l'API `clone()` con diversi possibili flag `CLONE_XXX`, ad esempio

`CLONE_FILES`	descrittori di file
`CLONE_FS`	file system (ad es., directory corrente)
`CLONE_SIGHAND`	gestori dei segnali
`CLONE_THREAD`	stesso "processo" (gruppo di thread)
`CLONE_VM`	spazio di memoria

- `fork()` ≡ `clone()` senza alcuno dei flag precedenti
- `pthread_create()` ≡ `clone()` con tutti i flag

La funzione di libreria clone()

La funzione di libreria `clone()` è simile alla funzione `pthread_create()`:

```
int clone(start, stack, flags, arg, ...);
```

- `start` è la funzione inizialmente eseguita dal processo leggero
- `stack` è l'indirizzo della cima dello stack UM del nuovo processo leggero
- `flags` sono i flag `CLONE_XXX`
- `arg` è l'argomento passato a `start()`

La funzione di libreria `clone()` si basa su una chiamata di sistema `clone` con semantica differente

La chiamata di sistema clone

La chiamata di sistema `clone` è simile alla `fork()`:

```
int clone(flags, stack);
```

- `flags` sono i flag `CLONE_XXX`
- `stack` è l'indirizzo della cima dello stack UM del nuovo processo leggero
  - se è nullo, il figlio utilizza una copia dello stack del padre
- L'esecuzione del figlio inizia con l'istruzione seguente l'invocazione di `clone`