Thread

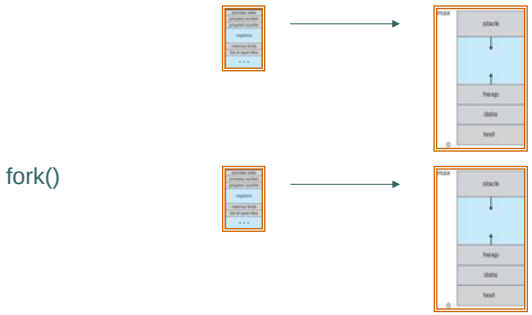
Thread

- Motivazioni
- Modelli Multithread
- Pthread, Threads in Win32, Thread in Java
- Problematiche relative ai Thread
- Thread Windows XP
- Thread Linux

Applicazioni reali

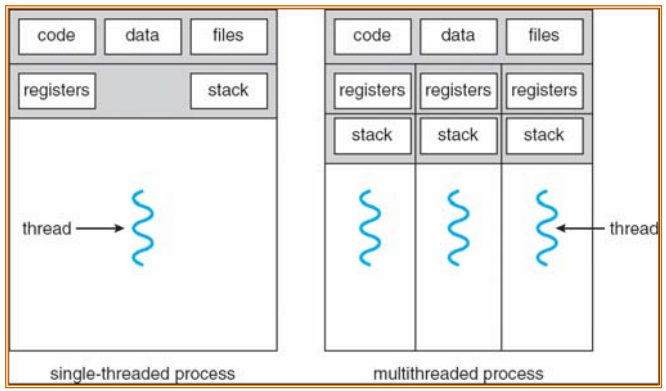
- Browser web
  - Rappresentazione dati sullo schermo
  - Più finestre che caricano dati diversi
  - Reperimento di dati dalla rete
- Elaboratore testi
  - Acquisizione dati da tastiera
  - Visualizzazione sullo schermo
  - Correttore ortografico durante battitura
- Server web
  - Richieste multiple da gestire

Creazione di nuovi processi

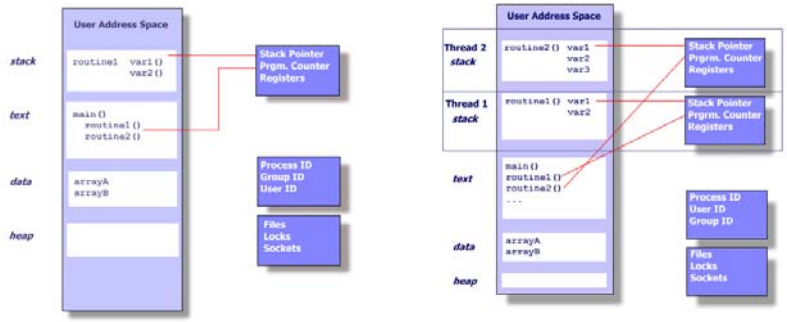


La creazione di un nuovo processo costa
I tempi di context switch sono elevati

Processi a singolo thread e processi multithread



Unix: processo a singolo thread e processo multithread



Benefici

- Prontezza di risposta
- Condivisione delle risorse
- Economia
- Uso di architetture multiprocessore

Thread vs Processi

La tabella che segue compara i tempi richiesti dall'esecuzione di **fork()** e **pthread_create()**, rispettivamente. I tempi riguardano la creazione di 50.000 processi/thread e sono espressi in secondi

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
IBM 375 MHz POWER3	61.94	3.49	53.74	7.46	2.76	6.79
IBM 1.5 GHz POWER4	44.08	2.21	40.27	1.49	0.97	0.97
IBM 1.9 GHz POWER5 p5-575	50.66	3.32	42.75	1.13	0.54	0.75
INTEL 2.4 GHz Xeon	23.81	3.12	8.97	1.70	0.53	0.30
INTEL 1.4 GHz Itanium 2	23.61	0.12	3.42	2.10	0.04	0.01

User Thread e Kernel Thread

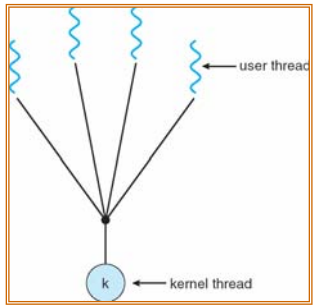
- I thread possono essere distinti in thread a livello utente e thread a livello kernel
- I primi sono gestiti "senza l'aiuto" del kernel
- I secondi sono gestiti direttamente dal sistema operativo
- In ultima analisi deve esistere una relazione tra i thread utente e i thread del kernel

Modelli di programmazione multithread

- Molti-a-uno
- Uno-a-uno
- Molti-a-Molti
- Ibrido

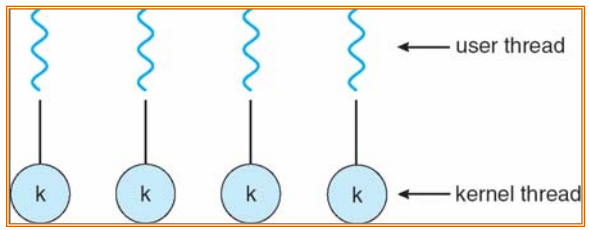
Molti-a-uno

- Molti thread a livello utente vengono mappati su un singolo thread del kernel
- Esempi:
  - Solaris Green Thread
  - GNU Portable Thread



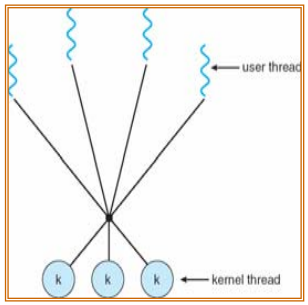
Modello uno-a-uno

- Ogni thread utente viene mappato su un thread del kernel
- Esempi
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 e versioni successive



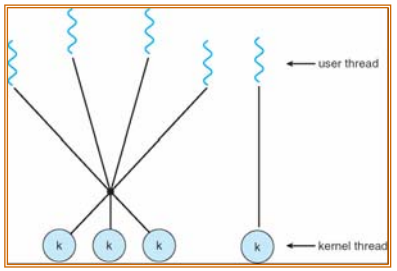
Modello molti-a-molti

- Diversi thread utente vengono mappati su diversi thread del kernel
- Permette al sistema operativo di creare un numero sufficiente di kernel thread per gestire i thread utenti
- Solaris prima della versione 9
- Windows NT/2000 attraverso il pacchetto *ThreadFiber*



Modello a due livelli

- Simile al modello molti-a-molti, eccetto che permette di mappare un thread utente su un kernel thread
- Esempi
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 e versioni precedenti



Pthread

- API disegnata in accordo allo standard POSIX (IEEE 1003.1c) per la creazione e la sincronizzazione dei thread
- L'API specifica il comportamento del thread offerto dalla libreria: l' implementazione dipende dallo sviluppo della libreria
- Comune nei sistemi operativi UNIX

Pthread

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

int sum; /* this data is shared by the thread(s) */

void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
pthread_t tid; /* the thread identifier */
pthread_attr_t attr; /* set of attributes for the thread */

if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        /*exit(1);*/
        return -1;
}

if (atoi(argv[1]) < 0) {
        fprintf(stderr,"Argument %d must be non-negative\n",atoi(argv[1]));
        /*exit(1);*/
        return -1;
}

/* get the default attributes */
pthread_attr_init(&attr);
```

Pthread

```
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);

/* now wait for the thread to exit */
pthread_join(tid,NULL);

printf("sum = %d\n",sum);
}

/**
 * The thread will begin control in this function
 */
void *runner(void *param)
{
int i, upper = atoi(param);
sum = 0;

        if (upper > 0) {
                for (i = 1; i <= upper; i++)
                        sum += i;
        }

        pthread_exit(0);
}
```

Thread Windows XP

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


DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(PVOID Param)
{
        DWORD Upper = *(DWORD *)Param;

        for (DWORD i = 0; i <= Upper; i++)
                Sum += i;


        return 0;
}
```

```
int main(int argc, char *argv[])
{
        DWORD ThreadId;
        HANDLE ThreadHandle;
        int Param;

        // do some basic error checking
        if (argc != 2) {
                fprintf(stderr,"An integer parameter is required\n");
                return -1;
        }

        Param = atoi(argv[1]);

        if (Param < 0) {
                fprintf(stderr, "an integer >= 0 is required \n");
                return -1;
        }

        // create the thread
        ThreadHandle = CreateThread(NULL, 0, Summation, &Param, 0, &ThreadId);

        if (ThreadHandle != NULL) {
                WaitForSingleObject(ThreadHandle, INFINITE);
                CloseHandle(ThreadHandle);
                printf("sum = %d\n",Sum);
        }
}
```

Thread Java

- I thread Java vengono gestiti dalla JVM
- I thread Java vengono creati:
  1. Estendendo la classe Thread
  2. Implementando l'interfaccia Runnable

```
public interface Runnable
{
  public abstract void run();
}
```

Esempio diThread Java

```
class Sum
{
  private int sum;

  public int get() {
                return sum;
        }

  public void set(int sum) {
                this.sum = sum;
        }
}
```

```
class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
                if (upper < 0)
                        throw new IllegalArgumentException();

                this.upper = upper;
                this.sumValue = sumValue;
    }

    public void run() {
                int sum = 0;

                for (int i = 0; i <= upper; i++)
                        sum += i;

                sumValue.set(sum);
    }
}
```

Esempio diThread Java

```
public class Driver
{
        public static void main(String[] args) {
                if (args.length != 1) {
                        System.err.println("Usage: Driver <integer>");
                        System.exit(0);
                }

                Sum sumObject = new Sum();
                int upper = Integer.parseInt(args[0]);

                Thread worker = new Thread(new Summation(upper, sumObject));
                worker.start();
                try {
                        worker.join();
                } catch (InterruptedException ie) { }
                System.out.println("The sum of " + upper + " is " + sumObject.get());
        }
}
```

Problematiche relative ai Thread

- Semantica delle chiamate **fork()** ed **exec()**
- Cancellazione dei thread
- Gestione dei segnali
- Gruppi di thread
- Dati specifici del thread
- Attivazione dello schedulatore

Semantica di fork() ed exec()

- La **fork()** duplica solo il thread chiamante o tutti i thread?
- La **exec()** funziona come nei sistemi Unix standard
- Alcuni sistemi implementano due **fork()**, una che duplica tutti i thread, l'altra che duplica solo il chiamante

Cancellazione dei thread

- Terminazione di un thread (thread target) prima che ha finito
- Due approcci generali:
  - **Cancellazione asincrona**: termina il thread target immediatamente
  - **Cancellazione differita**: permette al thread target di controllare periodicamente se deve essere cancellato

Gestione dei segnali

- I segnali sono usati nei sistemi UNIX per comunicare ad un processo che un certo evento si è verificato
- I segnali possono essere sincroni o asincroni rispetto all'esecuzione (e.g., divisione per 0, CTRL C)
- Un **gestore di segnale** viene usato per elaborare il segnale
  1. Un segnale viene generato da un evento particolare
  2. Il segnale viene inviato al processo
  3. Il segnale viene gestito

Esempio d'uso dei segnali

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

#define BUFFER_SIZE 50

static char buffer[BUFFER_SIZE];

/* the signal handler function */
void handle_SIGINT() {
        write(STDOUT_FILENO,buffer,strlen(buffer));

        exit(0);
}

int main(int argc, char *argv[])
{
        /* set up the signal handler */
        struct sigaction handler;
        handler.sa_handler = handle_SIGINT;
        sigaction(SIGINT, &handler, NULL);

        strcpy(buffer,"Caught <ctrl><c>\n");

        /* wait for <control> <C> */
        while (1)
                ;

        return 0;
}
```

Gestione segnali in presenza di più thread

Opzioni:

- Inviare il segnale al thread a cui il segnale si applica
- Inviare il segnale a tutti i thread del processo
- Inviare il segnale ad alcuni thread del processo
- Designare uno specifico thread alla ricezione di tutti i segnali per il processo

Gruppi di thread

- Crea un certo numero di thread in un pool dove essi attendono in attesa di lavoro
- Vantaggi:
  - Usualmente leggermente più veloce servire una richiesta attraverso un thread esistente piuttosto che creandone uno nuovo
  - Permette di limitare il numero di thread nelle applicazioni alla taglia del pool

Win32 Thread Pool

• Simile a creare un singolo thread

• Un puntatore ad una funzione, e.g. **PoolFunction**, viene passato ad una delle funzioni per i pool di thread

```
QueueUserWorkItem(&PoolFunction, NULL,0);
```

• Un thread dal pool viene chiamato ad eseguire la funzione

• Il package *java.util.concurrent* (Java 1.5) fornisce funzioni per pool di thread

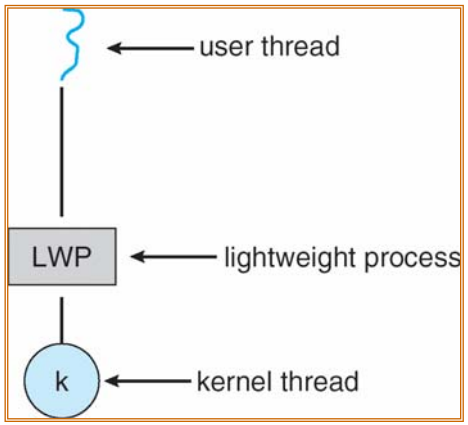
Dati specifici del thread

- Ogni thread ha la propria copia di dati
- Utile quando non si ha controllo sul processo di creazione dei thread (i.e., quando si usa un pool di thread)

Thread utente e kernel thread

- Sia il modello molti-a-molti che il modello a due livelli richiedono comunicazione per mantenere il numero appropriato di kernel thread per gestire i thread utenti creati dall' applicazione
- Solitamente, ad un thread utente è allocato un LWP (lightweight process), che la libreria dei thread utente vede come un processore virtuale
- Ogni LWP viene eseguito da un thread del kernel

Thread utente e kernel thread



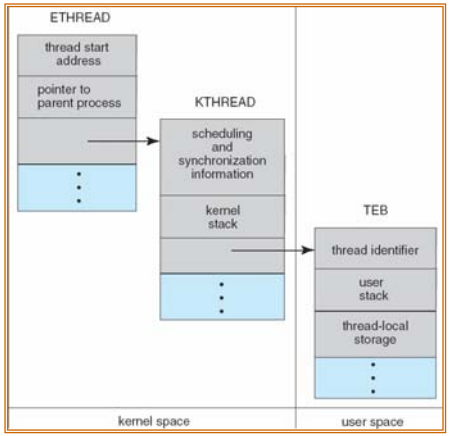
Comunicazione tra thread utente e kernel thread

- Schedulatore – attivato da una **upcall**, gestito da un **upcall handler**
- Il kernel fornisce all'applicazione LWP, su cui i thread utenti sono eseguiti
- Un evento (e.g., un thread sta per bloccarsi per I/O) viene comunicato dal kernel alla libreria tramite una upcall
- Un upcall handler (thread speciale della libreria) riceve un LWP e gestisce l'evento. Nel nostro caso, salva il contesto del thread che si blocca e recupera il LWP su cui "girava"
- Assegna, eventualmente, l 'LWP ad un altro thread pronto

Implementazione dei Kernel Thread in Windows XP

- Implementa il modello uno-a-uno
- Ogni thread contiene
  - Un thread id
  - Un insieme di registri
  - Fornisce stack utente e kernel separati
  - Area di memorizzazione di dati privati
- L'insieme dei registri, gli stack e l'area privata di memorizzazione dei dati, sono noti come **contesto** dei thread
- Le strutture di dati primarie di un thread includono:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Implementazione dei Kernel Thread in Windows XP



Implementazione dei kernel thread in Linux

- Linux usa il nome generico di *task* piuttosto che i nomi di *processi* e *thread*
- La creazione di un thread avviene attraverso la chiamata di sistema **clone()**
- **clone()** permette ad un task figlio di condividere lo spazio di indirizzamento del task (processo) padre

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.