Capitolo 4

Thread

OBIETTIVI

- **Introduzione del concetto di thread, l'unità fondamentale nell'utilizzo della CPU alla base dei moderni sistemi multithread.**
- **Analisi delle API per l'uso dei thread in Java, Win32 e Pthread.**

Nel modello introdotto nel Capitolo 3 un processo è considerato come un programma in esecuzione con un unico percorso di controllo. Molti sistemi operativi moderni permettono che un processo possa avere più percorsi di controllo che comunemente si chiamano *thread*. In questo capitolo sono introdotti diversi concetti associati ai sistemi di calcolo *multithread*, tra i quali un'approfondita descrizione dell'API per le librerie di thread di Pthreads, Win32 e Java. Si esaminano molti aspetti legati alla programmazione multithread, e il modo in cui influenza la progettazione dei sistemi operativi; infine, si analizza il modo in cui alcuni sistemi operativi moderni, come Windows XP e Linux, gestiscono i thread a livello kernel.

4.1 Introduzione

Un thread è l'unità di base d'uso della CPU e comprende un identificatore di thread (ID), un contatore di programma, un insieme di registri, e una pila (*stack*). Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati e altre risorse di sistema, come i file aperti e i segnali. Un processo tradizionale, chiamato anche **processo pesante** (*heavyweight process*), è composto da un solo thread. Un processo multithread è in grado di lavorare a più compiti in modo concorrente. La Figura 4.1 mostra la differenza tra un processo tradizionale, a **singolo thread**, e uno **multithread**.

4.1.1 Motivazioni

Molti programmi per i moderni PC sono predisposti per essere eseguiti da processi **multithread**. Di solito, un'applicazione si codifica come un processo a sé stante comprendente più thread di controllo: un programma di consultazione del Web potrebbe avere un thread per la rappresentazione sullo schermo di immagini e testo, mentre un altro thread potrebbe occuparsi del reperimento dei dati nella rete; un elaboratore di testi potrebbe avere un thread per la rappresentazione grafica, uno per la lettura dei dati immessi con la tastiera e uno per la correzione ortografica e grammaticale eseguita in sottofondo.

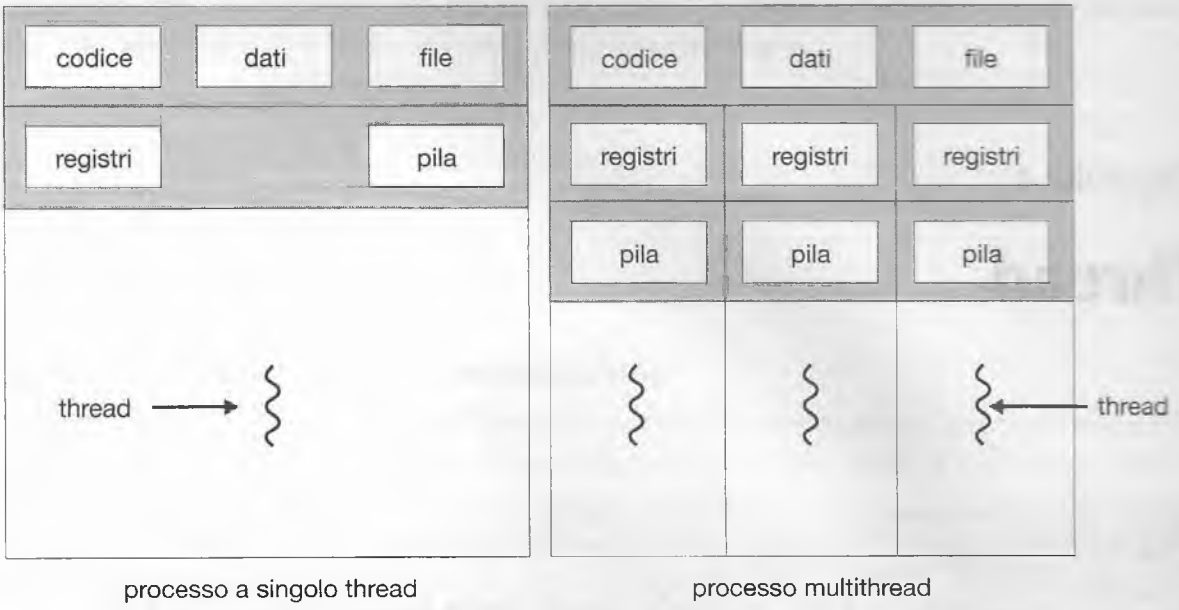

Figura 4.1 Processi a singolo thread e multithread.

In alcune situazioni, una singola applicazione deve poter gestire molti compiti simili tra loro. Per esempio, un server Web accetta dai client richieste di pagine web, immagini, suoni, e altro. Per un noto server Web potrebbero esservi molti (forse centinaia) client che vi accedono in modo concorrente; se il server Web fosse eseguito come un processo tradizionale a **singolo thread**, sarebbe in grado di soddisfare un solo client alla volta e un client potrebbe dover aspettare a lungo prima che venga esaudita la sua richiesta.

Una soluzione è eseguire il server come un singolo processo che accetta richieste. Quando ne riceve una, il server crea un processo separato per eseguirla. In effetti, questo metodo di creazione di processi era molto usato prima che si diffondesse la possibilità di gestione dei thread. La creazione dei processi è molto onerosa, sia a livello dei tempi sia dei costi; se il nuovo processo si deve occupare degli stessi compiti del processo corrente, non c'è alcuna ragione di accettare l'intero carico che la sua creazione comporta. Generalmente, per raggiungere lo stesso obiettivo è più conveniente impiegare un processo multithread. Nel caso del server Web, l'adozione di questo metodo porterebbe alla scelta di un processo multithread: il server genererebbe un thread distinto per ricevere eventuali richieste dei client; alla presenza di una richiesta, anziché creare un altro processo, si creerebbe un altro thread per soddisfarla. Il tutto è illustrato nella Figura 4.2.

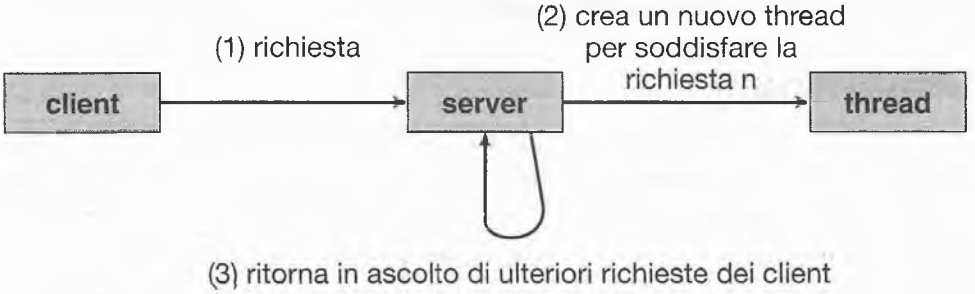

Figura 4.2 Architettura di server multithread.

I thread hanno anche un ruolo primario nei sistemi che impiegano le RPC (*remote procedure call*); si tratta di un sistema che permette la comunicazione tra processi, fornendo un meccanismo di comunicazione simile alle normali chiamate di funzione o procedura (Capitolo 3). Di solito, i server RPC sono multithread; quando riceve un messaggio, il server delega la gestione a un thread separato, in questo modo può gestire diverse richieste in modo concorrente.

Infine, molti kernel di sistemi operativi sono ormai multhithread, con i singoli thread dedicati a specifici servizi – per esempio, la gestione dei dispositivi periferici o delle interruzioni. È il caso di Solaris, il cui kernel avvia uno specifico insieme di thread specializzati nella gestione delle interruzioni; Linux usa un thread a livello kernel per la gestione della memoria libera del sistema.

4.1.2 Vantaggi

I vantaggi della programmazione multithread si possono classificare rispetto a quattro fattori principali.

1. **Tempo di risposta.** Rendere multithread un'applicazione interattiva può permettere a un programma di continuare la sua esecuzione, anche se una parte di esso è bloccata o sta eseguendo un'operazione particolarmente lunga, riducendo il tempo di risposta medio all'utente. Per esempio, un programma di consultazione del Web multithread potrebbe permettere l'interazione con l'utente tramite un thread mentre un'immagine verrebbe caricata da un altro thread.
2. **Condivisione delle risorse.** I processi possono condividere risorse soltanto attraverso tecniche come la memoria condivisa o il passaggio di messaggi. Queste tecniche devono essere esplicitamente messe in atto e organizzate dal programmatore. Tuttavia, i thread condividono d'ufficio la memoria e le risorse del processo al quale appartengono. Il vantaggio della condivisione del codice consiste nel fatto che un'applicazione può avere molti thread di attività diverse, tutti nello stesso spazio d'indirizzi.
3. **Economia.** Assegnare memoria e risorse per la creazione di nuovi processi è costoso; poiché i thread condividono le risorse del processo cui appartengono, è molto più conveniente creare thread e gestirne i cambi di contesto. È difficile misurare empiricamente la differenza del carico richiesto per creare e gestire un processo invece che un thread, tuttavia la creazione e gestione dei processi richiede in generale molto più tempo. In Solaris, per esempio, la creazione di un processo richiede un tempo trenta volte maggiore di quello richiesto per la creazione di un thread, un cambio di contesto per un processo richiede un tempo pari a circa cinque volte quello richiesto per un thread.
4. **Scalabilità.** I vantaggi della programmazione multithread aumentano notevolmente nelle architetture multiprocessore, dove i thread si possono eseguire in parallelo (uno per ciascun processore). Un processo con un singolo thread può funzionare solo su un processore, indipendentemente da quanti ve ne siano a disposizione. Il multithreading su una macchina con più processori incrementa il parallelismo. Esploreremo questo tema nel prossimo paragrafo.

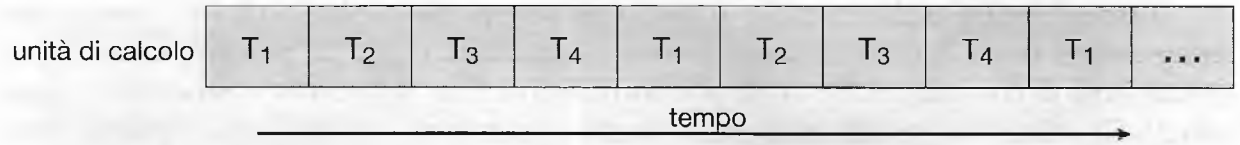

Figura 4.3 Esecuzione concorrente su un sistema a singola unità di calcolo.

4.1.3 Programmazione multicore

Una tendenza recente nel progetto dell'architettura dei sistemi consiste nel montare diverse unità di calcolo (*core*) su un unico processore (un processore *multicore*); ogni unità appare al sistema operativo come un processore separato (Paragrafo 1.3.2). La programmazione multithread offre un meccanismo per un utilizzo più efficiente di questi processori e aiuta a sfruttare al meglio la concorrenza. Si consideri un'applicazione con quattro thread. In un sistema con una singola unità di calcolo "esecuzione concorrente" significa solo che l'esecuzione dei thread è stratificata nel tempo, o come anche si dice, *interfogliata* (*interleaved*) (Figura 4.3), perché la CPU è in grado di eseguire un solo thread alla volta. Su un sistema multicore, invece, "esecuzione concorrente" significa che i thread possono funzionare in parallelo, dal momento che il sistema può assegnare thread diversi a ciascuna unità di calcolo (Figura 4.4).

La tendenza verso i sistemi multicore ha messo sotto pressione i progettisti di sistemi operativi e i programmatori di applicazioni, affinché entrambi utilizzino al meglio unità di calcolo multiple. I progettisti di sistemi operativi devono scrivere algoritmi di scheduling che utilizzano diverse unità di calcolo per permettere un'esecuzione parallela come quella mostrata nella Figura 4.4. Per i programmatori di applicazioni, la sfida consiste nel modificare programmi esistenti e progettare nuovi programmi multithread per trarre vantaggio dai sistemi multicore. In generale, possiamo individuare nelle cinque aree seguenti i principali obiettivi della programmazione dei sistemi multicore.

1. **Separazione dei task.** Consiste nell'esaminare le applicazioni al fine di individuare aree separabili in task distinti e concorrenti che possano essere eseguiti in parallelo su unità di calcolo distinte.
2. **Bilanciamento.** Nell'identificare i task eseguibili in parallelo, i programmatori devono far sì che i vari task eseguano compiti di mole e valore confrontabili. In alcuni casi si verifica che un determinato task non contribuisca al processo complessivo tanto quanto gli altri; in questi casi per eseguire il task può non valere la pena utilizzare un'unità di calcolo separata.
3. **Suddivisione dei dati.** Proprio come le applicazioni sono divise in task separati, i dati a cui i task accedono, e che manipolano, devono essere suddivisi per essere utilizzati da unità di calcolo distinte.

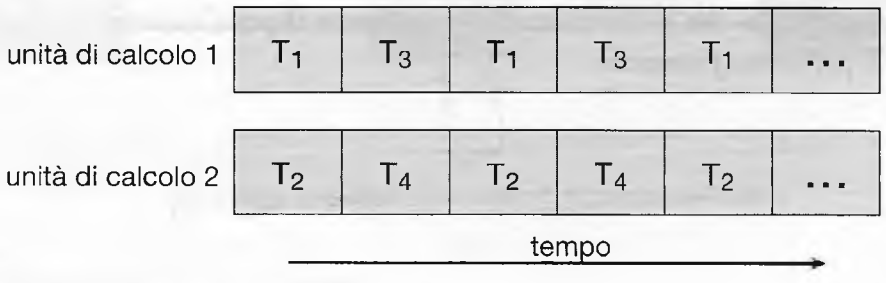

Figura 4.4 Esecuzione parallela su un sistema multicore.

4. **Dipendenze dei dati.** I dati a cui i task accedono devono essere esaminati per verificare le dipendenze tra due o più task. In esempi in cui un task dipende dai dati forniti da un altro, i programmatori devono assicurare che l'esecuzione dei task sia sincronizzata in modo da soddisfare queste dipendenze.
5. **Test e debugging.** Quando un programma funziona in parallelo su unità multiple, vi sono diversi possibili flussi di esecuzione. Effettuare i test e il debugging di programmi concorrenti è per natura più difficile rispetto al caso di applicazioni con un singolo thread.

Molti sviluppatori di software sostengono, in ragione degli obiettivi appena esposti, che l'avvento dei sistemi multicore richiederà in futuro un approccio interamente nuovo al progetto dei sistemi software.

4.2 Modelli di programmazione multithread

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, invece, sono gestiti direttamente dal sistema operativo. Praticamente tutti i sistemi operativi moderni dispongono di thread del kernel, compresi Windows XP, Linux, Mac OS X, Solaris, e Tru64 UNIX (precedentemente noto come Digital UNIX).

In ultima analisi, deve esistere una relazione tra i thread utente e i thread del kernel: in questo paragrafo ne analizziamo la natura attraverso tre casi comuni.

4.2.1 Modello da molti a uno

Il modello da molti a uno (Figura 4.5) fa corrispondere molti thread a livello utente a un singolo thread a livello kernel. Poiché si svolge nello spazio utente, la gestione dei thread risulta efficiente, ma l'intero processo rimane bloccato se un thread invoca una chiamata di sistema di tipo bloccante. Inoltre, poiché un solo thread alla volta può accedere al kernel, è impossibile eseguire thread multipli in parallelo in sistemi multiprocessore; la libreria **green threads**, disponibile per Solaris, usa questo modello, come anche GNU **portable thread**.

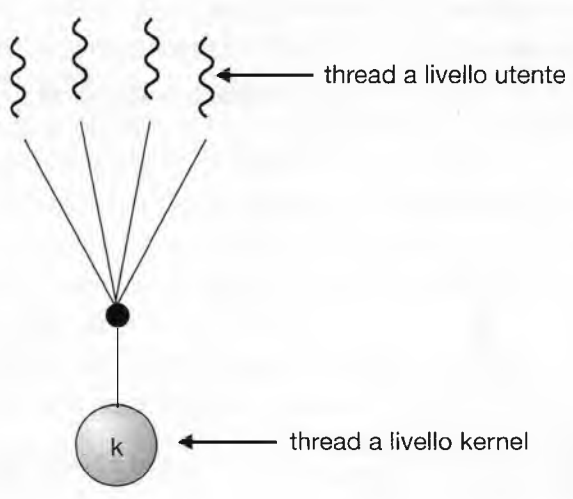

Figura 4.5 Modello da molti a uno.

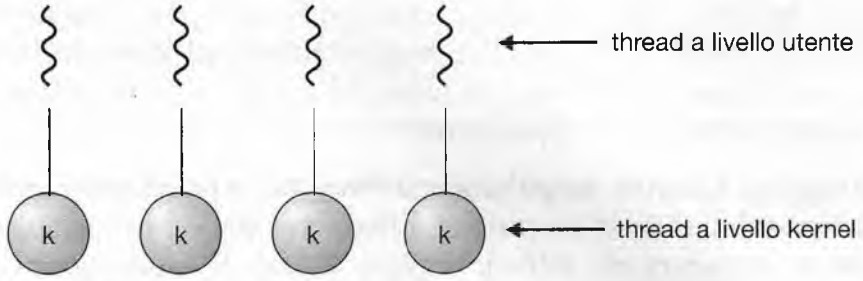

Figura 4.6 Modello da uno a uno.

4.2.2 Modello da uno a uno

Il modello da uno a uno (Figura 4.6) mette in corrispondenza ciascun thread a livello utente con un thread a livello kernel. Questo modello offre un grado di concorrenza maggiore rispetto al precedente, poiché anche se un thread invoca una chiamata di sistema bloccante, è possibile eseguire un altro thread; il modello permette anche l'esecuzione dei thread in parallelo nei sistemi multiprocessore. L'unico svantaggio di questo modello è che la creazione di ogni thread a livello utente comporta la creazione del corrispondente thread a livello kernel. Poiché il carico dovuto alla creazione di un thread a livello kernel può compromettere le prestazioni di un'applicazione, la maggior parte delle realizzazioni di questo modello limita il numero di thread gestibili dal sistema. I sistemi operativi Linux, insieme alla famiglia dei sistemi operativi Windows, adottano il modello da uno a uno.

4.2.3 Modello da molti a molti

Il modello da molti a molti (Figura 4.7) mette in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel; quest'ultimo può essere specifico per una certa applicazione o per un particolare calcolatore (un'applicazione potrebbe assegnare un numero maggiore di thread a livello kernel a un'architettura multiprocessore rispetto quanti ne assegnerebbe a una con singola CPU). Nonostante il modello da molti a uno permetta ai programmatori di creare tanti thread a livello utente quanti ne desiderino, non viene garantita una concorrenza reale, poiché il meccanismo di scheduling del kernel può scegliere un solo thread alla volta. Il modello da uno a uno permette una maggiore con-

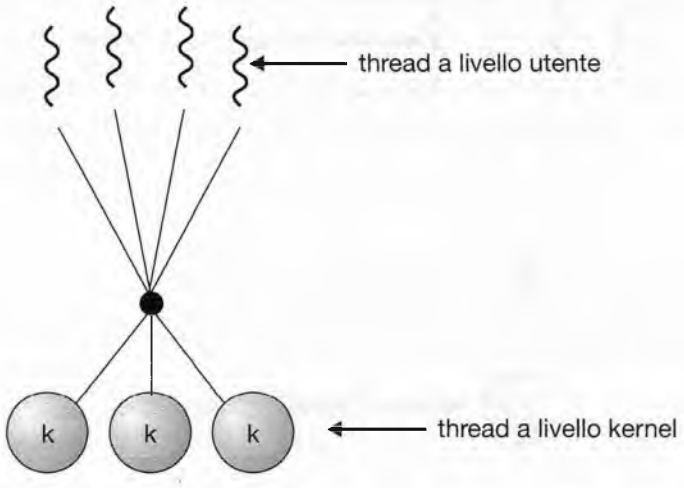

Figura 4.7 Modello da molti a molti.

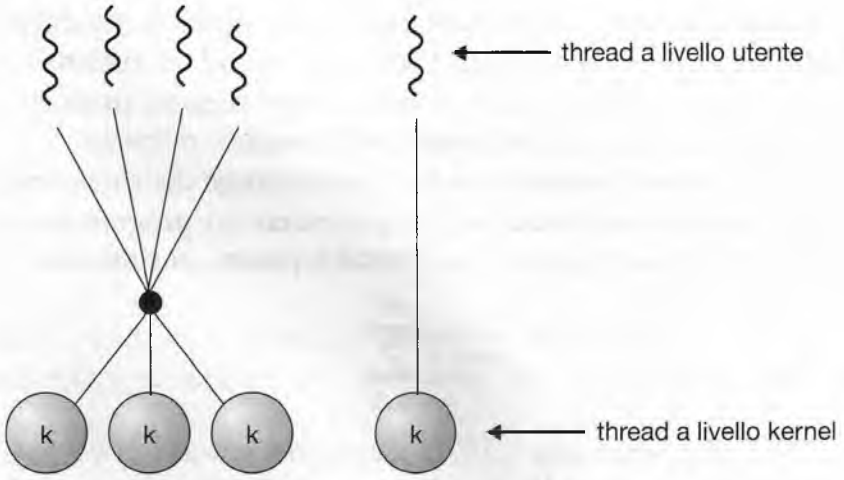

Figura 4.8 Modello a due livelli.

correnza, ma i programmatori devono stare attenti a non creare troppi thread all'interno di un'applicazione (in qualche caso si possono avere specifiche limitazioni sul numero di thread che si possono creare). Il modello da molti a molti non ha nessuno di questi difetti: i programmatori possono creare liberamente i thread che ritengono necessari, e i corrispondenti thread a livello kernel si possono eseguire in parallelo nelle architetture multiprocessore. Inoltre, se un thread impiega una chiamata disistema bloccante, il kernel può fare in modo che si esegua un altro thread.

Una diffusa variante del modello da molti a molti mantiene la corrispondenza fra più thread utente con un numero minore o uguale di thread del kernel, ma permette anche di vincolare un thread utente a un solo thread del kernel. La variante, detta a volte *modello a due livelli* (Figura 4.8), è impiegata per esempio da IRIX, HP-UX, e Tru64 UNIX, e dalle versioni di Solaris precedenti a Solaris 9, a partire dalla quale il sistema segue il modello da uno a uno.

4.3 Librerie dei thread

La **libreria dei thread** fornisce al programmatore una API per la creazione e la gestione dei thread. I metodi con cui implementare una libreria dei thread sono essenzialmente due. Nel primo, la libreria è collocata interamente a livello utente, senza fare ricorso al kernel. Il codice e le strutture dati per la libreria risiedono tutti nello spazio degli utenti. Questo implica che invocare una funzione della libreria si traduce in una chiamata locale a una funzione nello spazio degli utenti e non in una chiamata di sistema.

Il secondo metodo consiste nell'implementare una libreria a livello kernel, con l'ausilio diretto del sistema operativo. In questo caso, il codice e le strutture dati per la libreria si trovano nello spazio del kernel. Invocare una funzione della API per la libreria provoca, generalmente, una chiamata di sistema al kernel.

Attualmente, sono tre le librerie di thread maggiormente in uso: (1) Pthreads di POSIX, (2) Win32 e (3) Java. Pthreads, estensione dello standard POSIX, può presentarsi sia come libreria a livello utente sia a livello kernel. La libreria di thread Win32 è una libreria a livello kernel per i sistemi Windows. La API per la creazione dei thread in Java è gestibile direttamente dai programmi Java. Tuttavia, data la peculiarità di funzionamento della JVM, quasi

sempre eseguita all'interno di un sistema operativo che la ospita, la API di Java per i thread è solitamente implementata per mezzo di una libreria dei thread del sistema ospitante. Perciò, i thread di Java sui sistemi Windows sono in effetti implementati mediante la API Win32; sui sistemi UNIX e Linux, invece, si adopera spesso Pthreads.

Nel seguito affrontiamo i fondamenti della generazione dei thread nelle tre librerie a ciò dedicate. Come esempio dimostrativo, progetteremo un programma che computa la somma dei primi N interi non negativi in un thread separato, in simboli:

$$sum = \sum_{i=0}^{N} i$$

Se, per esempio, $N = 5$, si avrebbe $sum = 15$, la somma dei numeri da 0 a 5. Ciascuno dei tre programmi funzionerà inserendo nella riga di comando l'indice superiore N della sommatoria; inserendo 8, quindi, si otterrà come risultato la somma dei valori interi da 0 a 8.

4.3.1 Pthreads

Col termine **Pthreads** ci si riferisce allo standard POSIX (IEEE 1003.1c) che definisce la API per la creazione e la sincronizzazione dei thread. Non si tratta di una *realizzazione*, ma di una *definizione* del comportamento dei thread; i progettisti di sistemi operativi possono realizzare le API così definite come meglio credono. Sono molti i sistemi che implementano le specifiche Pthreads; fra questi, Solaris, Linux, Mac OS X, e Tru64 UNIX. Per i vari sistemi Windows, sono disponibili implementazioni *shareware* di dominio pubblico.

Il programma C multithread nella Figura 4.9 esemplifica la API Pthreads tramite il calcolo di una sommatoria eseguito da un thread apposito. Nei programmi Pthreads, i nuovi thread sono eseguiti a partire da una funzione specificata. Nel programma in esame si tratta della funzione `runner()`. All'inizio dell'esecuzione del programma c'è un unico thread di controllo che parte da `main()`; dopo una fase d'inizializzazione, `main()` crea un secondo thread che inizia l'esecuzione dalla funzione `runner()`. Entrambi i thread condividono i valori globali di `sum`.

Tutti i programmi che impiegano la libreria Pthreads devono includere il file d'intestazione `pthread.h`. La dichiarazione di variabili `pthread_t tid` specifica l'identificatore per il thread da creare. Ogni thread ha un insieme di attributi che includono la dimensione della pila e informazioni di scheduling. La dichiarazione `pthread_attr_t attr` riguarda la struttura dati per gli attributi del thread, i cui valori si assegnano con la chiamata di funzione `pthread_attr_init(&attr)`. Poiché non sono stati esplicitamente forniti valori per gli attributi, si usano quelli predefiniti. (Nel Capitolo 5 saranno esaminati alcuni degli attributi di scheduling offerti dalle API Pthreads.) La chiamata di funzione `pthread_create` crea un nuovo thread. Oltre all'identificatore del thread e ai suoi attributi, si passa anche il nome della funzione da cui il nuovo thread inizierà l'esecuzione, in questo caso la funzione `runner()`, e il numero intero fornito come parametro alla riga di comando e individuato da `argv[1]`.

A questo punto il programma ha due thread: il thread iniziale (o genitore), in `main()`; e il thread che esegue la somma (o figlio), in `runner()`. Dopo aver creato il secondo, il primo thread attende il completamento del secondo chiamando la funzione `pthread_join()`. Il secondo thread termina quando s'invoca la funzione `pthread_exit()`. Quando il thread che esegue la somma termina, il thread iniziale produce in uscita il valore condiviso `sum` della sommatoria.

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

int sum; /* questo dato è condiviso dai thread */
void *runner(void *param); /* il thread */

int main(int argc, char *argv[])
{
  pthread_t tid; /* identificatore del thread */
  pthread_attr_t attr; /* insieme di attributi del thread */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n", atoi(argv[1]));
    return -1;
  }

  /* reperisce gli attributi predefiniti */
  pthread_attr_init(&attr);
  /* crea il thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* attende la terminazione del thread */
  pthread_join(tid,NULL);

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

/* Il thread assume il controllo da questa funzione */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

Figura 4.9 Programma multithread in linguaggio C che impiega la API Pthreads.

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* dato condiviso dai thread */
/* il nuovo thread è eseguito in questa funzione apposita */

DWORD WINAPI Summation(LPVOID 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;
  /* semplice controllo degli errori sui valori in ingresso
*/
  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;
  }

  // crea il nuovo thread
  ThreadHandle = CreateThread()
    NULL, // attributi di default per la sicurezza
    0, // dimensione di default della pila
    Summation, // funzione del thread
    &Param, // parametro della funzione del thread
    0, // flag di creazione di default
    &ThreadId); // restituisce l'identificatore del thread

  if (ThreadHandle != NULL) {
    // attende che il thread figlio termini
    WaitForSingleObject(ThreadHandle,INFINITE);

    // chiude il riferimento al thread figlio
    CloseHandle(ThreadHandle);

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

Figura 4.10 Programma C multithread che impiega la API Win32.

4.3.2 Thread in Win32

La tecnica usata dalla libreria Win32 per la creazione dei thread può richiamare, per molti versi, quella di Pthreads. Illustriamo la API Win32 nel programma C mostrato dalla Figura 4.10. Si noti che per utilizzare la API Win32 è necessario includere il file d'intestazione `windows.h`.

Come nella versione di Pthreads della Figura 4.9, i dati condivisi da thread separati – nella fattispecie, `Sum` – sono globali (il tipo `DWORD` è un intero di 32 bit privo di segno). Dobbiamo inoltre definire la funzione `Summation()` che il nuovo thread eseguirà. A questa funzione è passato un puntatore a `void`, che in Win32 è `LPVOID`. Il thread che esegue questa funzione imposta la variabile globale `Sum` al valore della sommatoria da 0 fino al parametro passato a `Summation()`.

Nella API Win32 i nuovi thread si generano tramite la funzione `CreateThread()`, che – proprio come in Pthreads – accetta una serie di attributi del thread come parametri. Tali attributi includono le informazioni sulla sicurezza, la dimensione della pila e un indicatore (flag) per segnalare se il thread debba avere inizio nello stato d'attesa. Ci serviremo, nel programma, dei valori di default di questi attributi, che inizialmente non pongono il thread in stato d'attesa, bensì lo rendono eseguibile dallo scheduler della CPU. Una volta creato il nuovo thread, il thread iniziale deve attenderne il completamento prima di produrre in uscita il valore di `Sum`, poiché esso è computato dal nuovo thread. Come si ricorderà, nel programma Pthread (Figura 4.9) il thread iniziale era posto in attesa della terminazione del nuovo thread tramite la funzione `pthread_join()`. La chiamata equivalente nella API Win32 è `WaitForSingleObject()`, che ottiene la sospensione del thread iniziale fintanto che il nuovo thread non abbia terminato (Figura 4.10). (Indagheremo più a fondo sulla sincronizzazione nel Capitolo 6.)

4.3.3 Thread Java

I thread rappresentano il paradigma fondamentale per l'esecuzione dei programmi in ambiente Java; il linguaggio Java, con la propria API, è provvisto di una ricca gamma di caratteristiche per la generazione e la gestione dei thread. Tutti i programmi scritti in Java incorporano almeno un thread di controllo – persino un semplice programma, costituito soltanto da un metodo `main()`, è eseguito dalla JVM come un singolo thread.

In un programma Java vi sono due tecniche per la generazione dei thread. Una è creare una nuova classe derivata dalla classe `Thread` e "sovrascrivere" (*override*) il suo metodo `run()`. L'alternativa, usata più comunemente, consiste nella definizione di una classe che implementi l'interfaccia `Runnable`, corrispondente a:

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

Per implementare `Runnable`, una classe è tenuta a definire il metodo `run()`. Il codice che implementa `run()` sarà eseguito in un thread distinto.

La Figura 4.11 mostra la versione Java di un programma multithread che calcola la somma degli interi da 0 a *N*. La classe `Summation` implementa l'interfaccia `Runnable`. La generazione del thread prevede che si crei un'istanza della classe `Thread` passando al costruttore un oggetto `Runnable`.

La creazione di un oggetto di classe `Thread` non equivale a generare un nuovo thread: è il metodo `start()` che avvia effettivamente il nuovo thread. L'invocazione del metodo `start()` ha il duplice effetto di:

1. allocare la memoria e inizializzare un nuovo thread nella JVM;
2. chiamare il metodo `run()`, cosa che rende il thread eseguibile dalla JVM. Si osservi come il metodo `run()` non sia mai chiamato per via diretta, ma solo tramite la mediazione di `start()`.

```
class Sum
{
  private int sum;

  public int getSum() {
   return sum;
  }

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

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

  public Summation(int upper, Sum sumValue) {
   this.upper = upper;
   this.sumValue = sumValue;
  }

  public void run() {
   int sum = 0;
   for (int i = 0; i <= upper; i++)
      sum += i;
   sumValue.setSum(sum);
  }
}

public class Driver
{
  public static void main(String[] args) {
```

Figura 4.11 Programma Java per il calcolo di una sommatoria. (*continua*) ➤

```
    if (args.length > 0) {
     if (Integer.parseInt(args[0]) < 0)
      System.err.println(args[0] + " must be >= 0.");
     else {
       // crea l'oggetto condiviso dai thread
       Sum sumObject = new Sum();
       int upper = Integer.parseInt(args[0]);
       Thread thrd = new Thread(new Summation(upper,
sumObject));
       thrd.start();
       try {
         thrd.join();
         System.out.println
               ("The sum of "+upper+" is
"+sumObject.getSum());
      } catch (InterruptedException ie) { }
     }
   }
   else
    System.err.println("Usage: Summation <integer value>"); }
  }
```

Figura 4.11 (*continua*) Programma Java per il calcolo di una sommatoria. ■

All'avvio del programma che calcola la somma, la JVM crea due thread. Il primo è il thread genitore, che inizia a essere eseguito dal metodo `main()`. Il secondo thread, figlio del primo, ha origine quando il metodo `start()` è invocato sull'oggetto di classe `Thread`. L'esecuzione di questo thread figlio inizia dal metodo `run()` della classe `Summation`. Dopo aver restituito il valore della somma, il thread termina all'uscita dal proprio metodo `run()`.

La condivisione dei dati tra thread diversi è comune in Win32 e in Pthreads: i dati condivisi sono semplicemente dichiarati globali. In quanto linguaggio orientato agli oggetti puro, Java non contempla la nozione di variabile globale: la condivisione dei dati fra più thread in Java avviene tramite il passaggio di riferimenti a uno stesso oggetto. Nel programma in Java della Figura 4.11 il thread principale e quello che calcola la somma condividono un oggetto di classe `Sum`. A tale oggetto condiviso si accede attraverso gli appositi metodi `getSum()` e `setSum()`. Ci si potrebbe chiedere perché non si usi un oggetto di classe `Integer` anziché predisporre una nuova classe `sum`. La ragione è che la classe `Integer` è **immutabile** – ovvero, una volta impostato il valore di una sua istanza, non può più cambiare.

Si tenga presente che i thread genitori nelle librerie Pthreads e Win32 usano, rispettivamente, `pthread_join()` e `WaitForSingleObject()` per attendere la conclusione del thread che esegue la somma prima di procedere. Il metodo `join()` in Java fornisce una simile funzionalità. (Si noti che `join()` può sollevare una `InterruptedException`, che nel codice abbiamo deciso di non gestire.)

LA JVM E IL SISTEMA OPERATIVO RESIDENTE

La macchina virtuale del linguaggio Java (JVM) è solitamente implementata sulla base di un sistema operativo sottostante. Questo assetto permette alla JVM di nascondere i dettagli del sistema operativo e di offrire un ambiente astratto e coerente che consente ai programmi scritti in Java di essere eseguiti su qualsiasi piattaforma che disponga di una JVM. Le specifiche della JVM non prescrivono come i thread Java debbano corrispondere ai servizi del sistema operativo sottostante, lasciando i dettagli all'implementazione. Il sistema operativo Windows XP, per esempio, adotta il modello da uno a uno; perciò, ogni thread Java di una JVM installata su questo sistema corrisponde a un thread a livello kernel. Sui sistemi che adottano il modello da molti a molti, come il Tru64 UNIX, i thread di Java sono associati a thread sottostanti secondo il modello da molti a molti. La JVM per Solaris impiegava inizialmente il modello da molti a uno (la libreria *green threads*, citata in precedenza); versioni successive della JVM per Solaris hanno eletto il modello da molti a molti. A partire da Solaris 9, i thread di Java rimangono associati ai thread sottostanti secondo il modello da uno a uno. Oltre a queste considerazioni, può esistere una relazione tra la libreria dei thread di Java e la libreria dei thread del sistema operativo residente. Le varie versioni della JVM per la famiglia di sistemi operativi Windows, per esempio, potrebbero ricorrere alla API Win32 al fine di implementare thread di Java; i sistemi Linux e Solaris potrebbero impiegare la API Pthreads.

4.4 Questioni di programmazione multithread

In questo paragrafo si affrontano alcune problematiche legate ai programmi multithread.

4.4.1 Chiamate di sistema `fork()` ed `exec()`

Nel Capitolo 3 è descritto l'uso della chiamata di sistema `fork()` per la creazione di un nuovo processo tramite la duplicazione di un processo esistente. In un programma multithread la semantica delle chiamate di sistema `fork()` ed `exec()` cambia: se un thread in un programma invoca la chiamata di sistema `fork()`, il nuovo processo potrebbe, in generale, contenere un duplicato di tutti i thread oppure del solo thread invocante.

Alcuni sistemi UNIX includono entrambe le versioni. La chiamata di sistema `exec()` di solito funziona nello stesso modo descritto nel Capitolo 3: se un thread invoca la chiamata di sistema `exec()`, il programma specificato come parametro della `exec()` sostituisce l'intero processo, inclusi tutti i thread.

L'uso delle due versioni della `fork()` dipende dall'applicazione. Se s'invoca la `exec()` immediatamente dopo la `fork()`, la duplicazione dei thread non è necessaria, poiché il programma specificato nei parametri della `exec()` sostituirà il processo. In questo caso conviene duplicare il solo thread chiamante. Tuttavia, se la `exec()` non segue immediatamente la `fork()`, potrebbe essere utile una duplicazione di tutti i thread del processo genitore.

4.4.2 Cancellazione

La **cancellazione dei thread** è l'operazione che permette di terminare un thread prima che completi il suo compito. Per esempio, se più thread eseguono una ricerca in modo concorrente in una base di dati e un thread riporta il risultato, gli altri thread possono essere annullati. Una situazione analoga potrebbe verificarsi quando un utente preme il pulsante di

terminazione di un programma di consultazione del Web per interrompere il caricamento di una pagina. Spesso il caricamento di una pagina è gestito da un thread distinto; quando l'utente preme il pulsante di terminazione, il thread che sta caricando la pagina viene cancellato.

Un thread da cancellare è spesso chiamato **thread bersaglio** (*target thread*). La cancellazione di un thread bersaglio può avvenire in due modi diversi:

1. **cancellazione asincrona**. Un thread fa immediatamente terminare il thread bersaglio;
2. **cancellazione differita**. Il thread bersaglio può periodicamente controllare se deve terminare, in modo da riuscirvi in modo opportuno.

Si presentano difficoltà con la cancellazione nei casi in cui ci siano risorse assegnate a un thread cancellato, o se si cancella un thread mentre sta aggiornando dei dati che condivide con altri thread. Quest'ultimo caso è particolarmente problematico se si tratta di cancellazione asincrona. Il sistema operativo di solito si riappropria delle risorse di sistema usate da un thread cancellato, ma spesso non si riappropria di tutte le risorse. Quindi, la cancellazione di un thread in modo asincrono potrebbe non liberare una risorsa necessaria per tutto il sistema.

La cancellazione differita invece funziona tramite un thread che segnala la necessità di cancellare un certo thread bersaglio; la cancellazione avviene soltanto quando il thread bersaglio verifica se debba essere o meno cancellato. Questo metodo permette di programmare la verifica in un punto dell'esecuzione in cui il thread sia cancellabile senza problemi. Nella libreria Pthreads questi punti si chiamano **punti di cancellazione** (*cancellation point*).

4.4.3 Gestione dei segnali

Nei sistemi UNIX si usano i **segnali** per comunicare ai processi il verificarsi di determinati eventi. Un segnale si può ricevere in modo sincrono o asincrono, secondo la sorgente e la ragione della segnalazione dell'evento. Indipendentemente dal modo di ricezione sincrono o asincrono, tutti i segnali seguono lo stesso schema:

1. all'occorrenza di un particolare evento si genera un segnale;
2. s'invia il segnale a un processo;
3. una volta ricevuto, il segnale deve essere gestito.

Un accesso illegale alla memoria o una divisione per zero generano segnali sincroni. In questi casi, se un programma in esecuzione compie le suddette azioni, viene generato un segnale. I segnali sincroni s'inviano allo stesso processo che ha eseguito l'operazione causa del segnale (questo è il motivo per cui si chiamano sincroni).

Quando un segnale è causato da un evento esterno al processo in esecuzione, tale processo riceve il segnale in modo asincrono. Esempi di segnali di questo tipo sono la terminazione di un processo richiesta con specifiche combinazioni di tasti (come `<control><C>`) oppure la scadenza di un timer. Di solito un segnale asincrono s'invia a un altro processo.

Ogni segnale si può gestire in due modi:

1. tramite un gestore predefinito di segnali;
2. tramite un gestore di segnali definito dall'utente.

Per ogni segnale esiste un **gestore predefinito del segnale** che il kernel esegue quando deve gestire il segnale. La gestione predefinita è sostituibile da una funzione di **gestione del segnale definita dall'utente**, richiamata per gestire il segnale. Sia i segnali sincroni sia quelli

asincroni sono gestibili in modi diversi: alcuni si possono semplicemente ignorare (per esempio, il ridimensionamento di una finestra); altri si possono gestire terminando l'esecuzione del programma (per esempio, un accesso illegale alla memoria).

Poiché nell'inviare un segnale è sufficiente fare riferimento al processo interessato, per i processi a singolo thread la gestione dei segnali è semplice. Per i processi multithread si pone il problema del thread cui si deve inviare il segnale. In generale esistono le seguenti possibilità:

1. inviare il segnale al thread cui il segnale si riferisce;
2. inviare il segnale a ogni thread del processo;
3. inviare il segnale a specifici thread del processo;
4. definire un thread specifico per ricevere tutti i segnali diretti al processo.

Il metodo per recapitare un segnale dipende dal tipo di segnale. I segnali sincroni, per esempio, si devono inviare al thread che ha generato l'evento causa del segnale e non ad altri thread nel processo. Se si tratta di segnali asincroni la situazione non è invece così chiara; alcuni segnali asincroni, come il segnale che termina un processo (come `<control><C>`), si devono inviare a tutti i thread. La maggior parte delle versioni multithread del sistema operativo UNIX permettono che per ciascun thread si indichino i segnali da accettare e quelli da bloccare. Quindi, alcuni segnali asincroni si potrebbero recapitare soltanto ai thread che non li bloccano.

Tuttavia, poiché i segnali vanno gestiti una sola volta, di solito un segnale è recapitato solo al primo thread che non lo blocca. La funzione UNIX per recapitare i segnali è `kill(aid_t aid, int signal)`, dove `aid` specifica il processo a cui recapitare il segnale `signal`. La API Pthreads POSIX, però, dispone anche della funzione `pthread_kill(pthread_t tid, int signal)` che permette di specificare il thread (`tid`) cui recapitare il segnale.

Sebbene Windows non preveda la gestione esplicita dei segnali, questi si possono emulare con le **chiamate di procedure asincrone** (*asynchronous procedure call*, APC). Le funzioni APC permettono a un thread a livello utente di specificare la funzione da richiamare quando il thread riceve la comunicazione di un particolare evento. Come s'intuisce dal nome, una APC è grosso modo equivalente a un segnale asincrono di UNIX. Mentre tuttavia in un ambiente multithread UNIX necessita di un criterio di gestione dei segnali, il sistema delle APC è più semplice, poiché una APC è rivolta a un particolare thread e non a un processo.

4.4.4 Gruppi di thread

Nel Paragrafo 4.1 è descritto lo scenario di un server Web multithread in cui. per ogni richiesta ricevuta, il server crea un thread distinto per fornire il servizio richiesto. Nonostante la creazione di un thread distinto sia molto più vantaggiosa della creazione di un nuovo processo, un server multithread presenta diversi problemi. Il primo riguarda il tempo richiesto per la creazione del thread prima di poter soddisfare la richiesta, considerando anche il fatto che questo thread sarà terminato non appena avrà completato il proprio lavoro. La seconda questione è più problematica: se si permette che tutte le richieste concorrenti siano servite da un nuovo thread, non si è posto un limite al numero di thread attivi in modo concorrente nel sistema. Un numero illimitato di thread potrebbe esaurire le risorse del sistema, come il tempo di CPU o la memoria. L'impiego dei **gruppi di thread** (*thread pool*) è una possibile soluzione a questo problema.

L'idea generale è quella di creare un certo numero di thread alla creazione del processo, e organizzarli in un gruppo (*pool*) in cui attendano il lavoro che gli sarà richiesto. Quando un server riceve una richiesta, attiva un thread del gruppo – se ce n'è uno disponibile – e

gli passa la richiesta; dopo aver completato il suo lavoro, il thread rientra nel gruppo d'attesa. Se il gruppo non contiene alcun thread disponibile, il server attende fino al rientro di un thread. I vantaggi offerti sono i seguenti:

1. di solito il servizio di una richiesta tramite un thread esistente è più rapido, poiché elimina l'attesa della creazione di un nuovo thread;
2. un gruppo di thread limita il numero di thread esistenti a un certo istante; ciò è particolarmente rilevante per sistemi che non possono sostenere un elevato numero di thread concorrenti.

Il numero di thread di un gruppo si può determinare tramite euristiche che considerano fattori come il numero di CPU nel sistema, la quantità di memoria fisica e il numero atteso di richieste concorrenti da parte dei client. Architetture più raffinate per la gestione dei gruppi di thread possono correggere dinamicamente il numero di thread di un gruppo secondo schemi d'uso. Queste architetture hanno l'ulteriore vantaggio di presentare gruppi più piccoli – comportando quindi un minore impegno di memoria – quando il carico del sistema è basso.

La API Win32 mette a disposizione diverse funzioni legate ai gruppi di thread, il cui uso è simile alla creazione di un thread tramite la funzione `Thread Create()` presentata al Paragrafo 4.3.2. Si definisce una funzione da eseguire in un nuovo thread, come per esempio:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /**
    * Questa funzione gira come nuovo thread.
    **/
}
```

Un puntatore a `PoolFunction()` è poi passato a una delle apposite funzioni nella API per i gruppi di thread, il che avvierà uno dei thread del gruppo. Una di tali apposite funzioni è `QueueUserWorkItem()`, che accetta tre parametri:

- `LPTHREAD_START_ROUTINE Function` – un puntatore alla funzione da eseguire in un nuovo thread;
- `PVOID Param` – il parametro passato a `Function`;
- `ULONG Flags` – indica come il gruppo di thread debba creare e gestire l'esecuzione del nuovo thread.

Un esempio di chiamata è:

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

A seguito di questa invocazione, uno dei thread del gruppo richiamerà `PoolFunction()` per conto del programmatore. In questo esempio, `PoolFunction()` non riceve parametri, e il valore 0 di `Flags` indica l'assenza di indicazioni particolari per la creazione del thread.

Altre funzioni della API Win32 per i gruppi di thread offrono servizi di invocazione periodica di funzioni, o invocazioni guidate dal completamento di un'operazione di I/O asincrona. Anche il package `java.util.concurrent` di Java 1.5 offre funzionalità per la gestione di gruppi di thread.

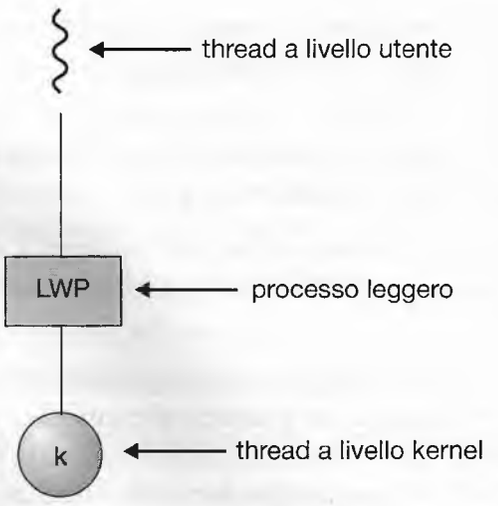

Figura 4.12 Processo leggero LWP.

4.4.5 Dati specifici dei thread

Uno dei vantaggi principali della programmazione multithread è dato dal fatto che i thread appartenenti allo stesso processo ne condividono i dati. Tuttavia, in particolari circostanze, ogni thread può necessitare di una copia privata di certi dati, chiamati **dati specifici di thread**. Per esempio, in un sistema per transazioni si può svolgere ciascuna transazione tramite un thread distinto e un identificatore unico per ogni transazione. Per associare ciascun thread al relativo identificatore si possono usare dati specifici dei thread. La maggior parte delle librerie di thread – incluse la Win32 e la Pthreads – e l'ambiente del linguaggio Java consentono l'impiego dei dati specifici di thread.

4.4.6 Attivazione dello scheduler

Un'ultima questione da affrontare in merito ai programmi multithread riguarda la comunicazione tra la libreria del kernel e la libreria per i thread, che può rendersi necessaria nel modello a due livelli e in quello da molti a molti (Paragrafo 4.2.3). È proprio grazie a questa forma di coordinamento che il numero dei thread nel kernel è modificabile dinamicamente, con l'obiettivo di conseguire le migliori prestazioni.

Molti sistemi che implementano o il modello da molti a molti o quello a due livelli collocano una struttura dati intermedia tra i thread del kernel e dell'utente. Questa struttura dati, nota come processo leggero o LWP (acronimo di *Lightweight Process*) è mostrata nella Figura 4.12. Dal punto di vista della libreria di thread a livello utente, LWP si presenta come un *processore virtuale* a cui l'applicazione può richiedere lo scheduling di un thread a livello dell'utente. Ciascun LWP è associato a un thread del kernel, e sono proprio i thread del kernel che il sistema operativo pone in esecuzione sui processori fisici. Se un thread del kernel si arresta (mentre attende il completamento di un'operazione di I/O, per esempio) anche LWP si blocca. L'effetto a catena risale fino al thread a livello utente associato a LWP, che si blocca anch'esso.

Per un'efficiente esecuzione un'applicazione può aver bisogno di un numero imprecisato di LPW. Si consideri un processo con prevalenza di elaborazione eseguito da un singolo processore. In questa situazione è eseguibile solo un thread per volta, dunque un LWP è sufficiente. Un'applicazione con prevalenza di I/O potrebbe, tuttavia, richiedere l'esecuzione di

molteplici LWP. Di solito, è necessario un LWP per ogni chiamata di sistema concorrente bloccante. Supponiamo, per esempio, che giungano allo stesso tempo cinque richieste differenti per la lettura di file. Sono necessari cinque LWP, nel caso che tutte le richieste risiedano nel kernel in attesa di essere completate. Se un processo ha soltanto quattro LWP, la quinta richiesta deve attendere che uno degli LWP sia rilasciato dal kernel.

Uno dei modelli di comunicazione tra la libreria a livello utente e il kernel è conosciuto come **attivazione dello scheduler**. Il suo funzionamento è il seguente: il kernel fornisce all'applicazione una serie di processori virtuali (LWP), mentre l'applicazione esegue lo scheduling dei thread dell'utente sui processori virtuali disponibili. Inoltre, il kernel deve informare l'applicazione se si verificano determinati eventi, seguendo una procedura nota come **upcall**. Le upcall sono gestite dalla libreria dei thread mediante un apposito gestore, eseguito su un processore virtuale. Una situazione capace di innescare una upcall si verifica quando il thread di un'applicazione è sul punto di bloccarsi. In questo caso il kernel, tramite una upcall, informa l'applicazione che un thread è prossimo a bloccarsi, e identifica il thread in oggetto. Il kernel, quindi, assegna all'applicazione un nuovo processore virtuale. L'applicazione esegue un gestore della upcall su questo nuovo processore: il gestore salva lo stato del thread bloccante e rilascia il processore virtuale su cui era stato eseguito. Il gestore della upcall pianifica allora l'esecuzione di un altro thread sul processore virtuale che si è appena liberato. Quando si verifica l'evento atteso dal thread bloccante, il kernel fa un'altra upcall alla libreria dei thread per comunicare che il thread bloccato è nuovamente in condizione di essere eseguito. Il gestore di questa upcall necessita anch'esso di un processore virtuale: il kernel può crearne uno *ex novo*, o sottrarlo a un thread utente per prelazione. L'applicazione contrassegna il thread fino ad allora bloccato come pronto per l'esecuzione, ed esegue lo scheduling di un thread pronto per l'esecuzione su un processore virtuale disponibile.

4.5 Esempi di sistemi operativi

In questo paragrafo si descrive il modo in cui i thread sono implementati nei sistemi Windows XP e Linux.

4.5.1 Thread nel sistema Windows XP

Il sistema operativo Windows XP offre la API Win32; si tratta dell'API principale della famiglia dei sistemi operativi di Microsoft (Windows 95, 98, NT, 2000 e XP). La maggior parte del contenuto di questo paragrafo si applica all'intera famiglia Microsoft.

Un'applicazione per l'ambiente Windows XP si esegue come un processo separato; ogni processo può contenere uno o più thread. La API Win32 per la creazione dei thread è trattata nel Paragrafo 4.3.2. Il sistema Windows XP impiega il modello da uno a uno, descritto nel Paragrafo 4.2.2, secondo cui ogni thread a livello utente si associa a un thread del kernel. Tuttavia è disponibile anche la libreria **fiber**, che implementa il modello da molti a molti (Paragrafo 4.2.3). Ogni thread che appartiene a un processo può accedere allo spazio d'indirizzi di quel processo.

I componenti generali di un thread includono:

- un identificatore di thread (ID), che identifica univocamente il thread;
- un insieme di registri che rappresentano lo stato del processore;

- una pila (stack) utente, usata quando il thread è eseguito in modalità utente, e una pila del kernel, usata quando il thread è eseguito in modalità kernel;
- un'area di memoria privata, usata da diverse librerie di fase d'esecuzione e dinamiche (DLL).

L'insieme di registri, le pile e la memoria privata è detto **contesto** del thread. Le strutture dati principali di un thread includono:

- ETHREAD (*executive thread block*);
- KTHREAD (*kernel thread block*);
- TEB (*thread environment block*).

I componenti chiave dell'ETHREAD sono un puntatore al processo a cui il thread appartiene e l'indirizzo della funzione in cui il thread assume il controllo. La struttura ETHREAD contiene anche un puntatore alla corrispondente struttura KTHREAD. Quest'ultima include informazioni per il thread relative allo scheduling e alla sincronizzazione. Inoltre, KTHREAD contiene la pila del kernel (usata quando il thread viene eseguito in modalità kernel) e un puntatore alla struttura TEB.
Le strutture ETHREAD e KTHREAD risiedono interamente nello spazio del kernel; ciò implica che solo il kernel vi può accedere. La struttura dati TEB appartiene invece allo spazio utente e vi si accede quando il thread è eseguito in modalità utente. Tra gli altri campi, il TEB contiene una pila per la modalità utente e un vettore per dati specifici del thread che Windows XP chiama **memoria locale del thread** (*thread-local storage*). La struttura di un thread di Windows XP è illustrata nella Figura 4.13.

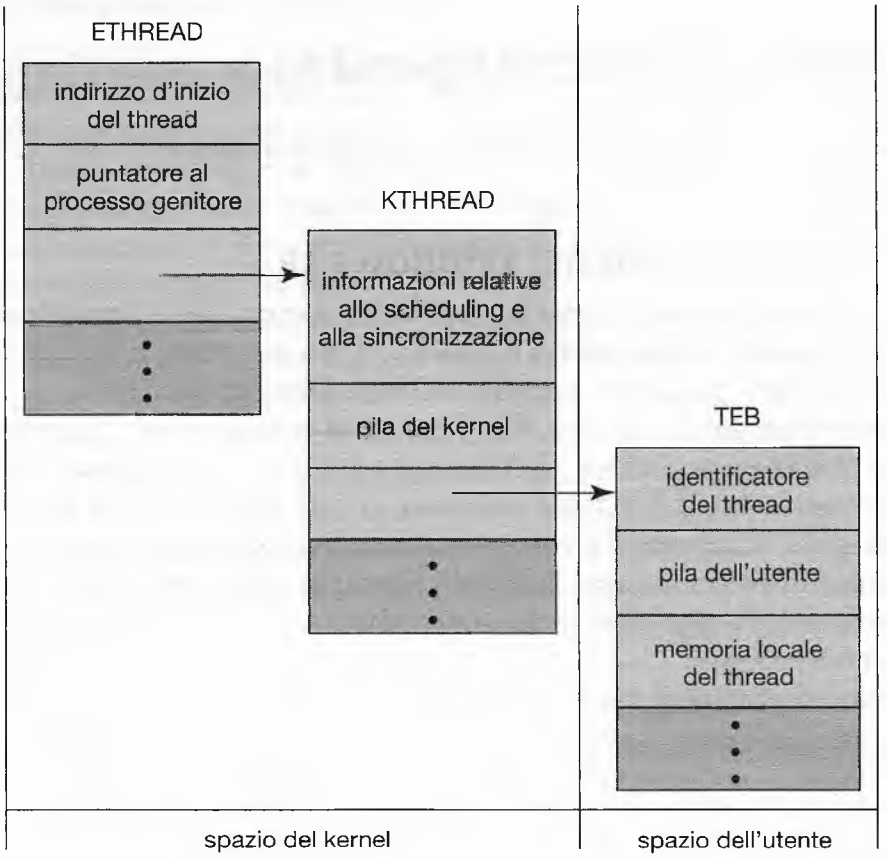

Figura 4.13 Strutture dati di un thread Windows XP.

4.5.2 Thread di Linux

Come si è visto nel Capitolo 3 Linux offre la chiamata di sistema `fork()` per duplicare un processo, e prevede inoltre la chiamata di sistema `clone()` per generare nuovo thread. Tuttavia Linux non distingue tra processi e thread, impiegando generalmente al loro posto il termine **task** (*operazione*) in riferimento al flusso del controllo di un programma. Quando `clone()` è invocata, riceve come parametro un insieme di indicatori (*flag*), al fine di stabilire quante e quali risorse del task genitore debbano essere condivise dal task figlio. Alcuni di questi flag sono illustrati nello schema seguente.

flag	significato
`CLONE_FS`	Condivisione delle informazioni sul file system
`CLONE_VM`	Condivisione dello stesso spazio di memoria
`CLONE_SIGHAND`	Condivisione dei gestori dei segnali
`CLONE_FILES`	Condivisione dei file aperti

Per esempio, qualora `clone()` riceva i flag `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND` e `CLONE_FILES`, il task genitore e il task figlio condivideranno le medesime informazioni sul file system (come la directory attiva), lo stesso spazio di memoria, gli stessi gestori dei segnali e lo stesso insieme di file aperti. Adoperare `clone()` in questo modo è equivalente a creare thread, dal momento che il task genitore condivide la maggior parte delle proprie risorse con il task figlio. Tuttavia, se nessuno dei flag è impostato al momento dell'invocazione di `clone()`, non si ha alcuna condivisione, e la funzionalità ottenuta diventa simile a quella fornita dalla chiamata di sistema `fork()`.

Questa condivisione a intensità variabile è resa possibile dal modo in cui un task è rappresentato nel kernel di Linux. Per ogni task, nel kernel esiste un'unica struttura dati (e precisamente, `struct task_struct`). Questa struttura, invece di memorizzare i dati del task relativo, utilizza dei puntatori ad altre strutture dove i dati sono effettivamente contenuti: per esempio, strutture dati che rappresentano l'elenco dei file aperti, le informazioni per la gestione dei segnali e la memoria virtuale. Quando si invoca fork(), si crea un nuovo task insieme con una *copia* di tutte le strutture dati del task genitore. Anche quando s'invoca la chiamata `clone()` si crea un nuovo task, ma anziché ricevere una copia di tutte le strutture dati, il nuovo task *punta* a queste o a quelle strutture dati del task genitore, a seconda dell'insieme di flag passati a `clone()`.

4.6 Sommario

Un thread è un percorso di controllo d'esecuzione all'interno di un processo. Un processo multithread contiene più percorsi di controllo diversi, ma che condividono lo stesso spazio d'indirizzi. I vantaggi della programmazione multithread includono un miglioramento del tempo di risposta, la condivisione di risorse all'interno del processo, il risparmio e la capacità di sfruttare le architetture dotate di più unità d'elaborazione.

I thread a livello utente sono thread visibili al programmatore e sconosciuti al kernel. Il kernel del sistema operativo gestisce thread a livello kernel. In generale, i thread a livello

utente richiedono minor tempo per essere creati e gestiti rispetto a quelli a livello kernel, senza necessità d'intervento da parte del kernel. Ci sono tre tipi diversi di modelli che descrivono le relazioni fra thread a livello utente e a livello kernel: il modello da molti a uno associa più thread a livello utente a un singolo thread a livello kernel; il modello da uno a uno associa ciascun thread a livello utente a un corrispondente thread a livello kernel; il modello da molti a molti associa dinamicamente più thread a livello utente a un numero minore o uguale di thread a livello kernel.

Molti sistemi operativi moderni prevedono la gestione dei thread a livello kernel: tra questi i sistemi Windows 98, NT, 2000 e XP, oltre a Solaris e Linux.

Per la creazione e la gestione dei thread le relative librerie forniscono una API al programmatore di applicazioni. Le tre più comuni librerie di thread sono: POSIX Pthread, Win32 per i sistemi Windows e i thread Java.

I programmi multithread presentano molti aspetti critici per il programmatore, tra cui la semantica delle chiamate di sistema `fork()` ed `exec()`; altri aspetti sono per esempio la cancellazione, la gestione dei segnali e i dati privati dei thread.

Esercizi pratici

4.1 Fornite due esempi di programmi nei quali il multithread offra prestazioni migliori rispetto a soluzioni con un singolo thread.

4.2 Quali sono due differenze tra i thread a livello utente e i thread a livello kernel? In quali circostanze un tipo è meglio dell'altro?

4.3 Descrivete le azioni intraprese da un kernel per cambiare contesto tra i thread a livello kernel.

4.4 Quali risorse vengono utilizzate quando si crea un thread? Come differiscono da quelle utilizzate quando si crea un processo?

4.5 Assumete che un sistema operativo mappi i thread a livello utente sul kernel utilizzando il modello molti a molti e che la mappatura avvenga tramite LWP. Assumete inoltre che il sistema permetta agli sviluppatori di creare dei thread real-time da utilizzare in sistemi real-time. È necessario vincolare un thread real-time a un LWP? Fornite una spiegazione.

4.6 Nel Paragrafo 4.3.1 si è descritto un programma Pthread che esegue una somma. Riscrivete il programma in Java.

Esercizi

4.7 Descrivete due esempi di programmazione multithread che offrano prestazioni migliori rispetto alla corrispondente soluzione a singolo thread.

4.8 Descrivete le azioni intraprese da una libreria di thread per il cambio di contesto tra thread a livello utente.

4.9 In quali circostanze una soluzione basata sulla programmazione multithread che sfrutta thread multipli del kernel offre prestazioni migliori di una soluzione a singolo thread implementata su un sistema monoprocessore?

4.10 Quali tra i seguenti componenti dello stato di un programma sono condivisi tra thread in un processo multithread?

a. Valori dei registri.
b. Memoria heap.
c. Variabili globali.
d. Pila.

4.11 È possibile che una soluzione multithread, impiegando thread multipli a livello utente, consegua prestazioni migliori su un sistema multiprocessore piuttosto che su un sistema a singolo processore?

4.12 Come descritto nel Paragrafo 4.5.2 Linux non fa distinzione tra processi e thread, cosicché un task apparirà più affine a un processo, oppure a un thread, a seconda dell'insieme di flag passati alla chiamata di sistema `clone()`. Tuttavia, Windows XP e Solaris, come molti altri sistemi operativi, trattano processi e thread in maniera diversa. In genere, per descrivere un processo, questi sistemi usano strutture dati contenenti un puntatore per ciascun thread appartenente al processo. Ponete a confronto queste due tecniche per rappresentare i processi e i thread nel kernel.

4.13 Il programma contenuto nella Figura 4.14 utilizza la API Pthreads. Quali dati in uscita verrebbero prodotti dal programma alla `RIGA C` e alla `RIGA P`?

4.14 Considerate un sistema multiprocessore e un programma multithread scritto con il modello da molti a molti. Ipotizziamo un numero più alto di thread a livello utente nel programma rispetto al numero di processori nel sistema. Analizzate che cosa implichi, in termini di efficienza, ciascuna delle seguenti possibilità.

a. Il numero di thread del kernel assegnati al programma è minore del numero di processori.
b. I thread del kernel assegnati al programma sono in numero uguale al numero dei processori.
c. Il numero di thread del kernel assegnati al programma è maggiore del numero di processori, ma minore del numero di thread a livello utente.

4.15 Scrivete un programma multithread che impieghi la libreria Pthreads, WIN32 o Java per la generazione di numeri primi. Il programma deve avere il seguente comportamento: l'utente avvia l'esecuzione del programma e inserisce un numero alla riga di comando; il programma crea un thread distinto che riporta tutti i numeri primi minori o uguali al numero inserito dall'utente.

4.16 Modificate il server basato sulle socket della Figura 3.19 nel Capitolo 3 in modo che esso dedichi un thread separato a ciascuna richiesta del client.

4.17 La successione di Fibonacci inizia con 0, 1, 1, 2, 3, 5, 8, ...

Essa è definita da:

$fib_0 = 0$
$fib_1 = 1$
$fib_n = fib_{n-1} + fib_{n-2}$

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

int value = 0;
void *runner(void *param); /* il thread */

int main(int argc, char *argv[])
{
int pid;
pthread_t tid;
pthread_attr_t attr;

  pid = fork();

  if (pid == 0) {/* processo figlio */
    pthread_attr_init(&attr);
    pthread_create(&tid,&attr,runner,NULL);
    pthread_join(tid,NULL);
    printf("CHILD: value = %d",value); /* RIGA C */
  }
  else if (pid > 0) {/* processo genitore */
    wait(NULL);
    printf("PARENT: value = %d",value); /* RIGA P */
  }
}
void *runner(void *param) {
  value = 5;
  pthread_exit(0);
}
```

Figura 4.14 Programma C dell'Esercizio 4.13.

Scrivete un programma multithread usando la libreria Java, oppure quella di Pthreads o di Win32, che generi la successione di Fibonacci. Il programma deve avere il seguente comportamento: l'utente avvia l'esecuzione del programma e inserisce alla riga di comando il numero di termini della successione di Fibonacci che il programma deve generare. Il programma crea un thread separato per la generazione dei numeri di Fibonacci, ma colloca i termini della successione in dati condivisi dai thread (un vettore è probabilmemte la struttura dati più adatta). Quando il thread figlio conclude l'esecuzione, il thread genitore emette la sequenza generata dal figlio. Poiché il thread genitore non può produrre in uscita la sequenza prima che il thread figlio abbia terminato, sarà necessario applicare la tecnica illustrata nel Paragrafo 4.3, per sincronizzare il thread genitore con il thread figlio.

4.18 Nell'Esercizio 3.18 del Capitolo 3 si spiega come progettare un server eco tramite la API di Java per i thread. Tale server, tuttavia, è a singolo thread, ossia non è in grado di rispondere a richieste simultanee dei client finché il client attualmente servito non termini. Si modifichi la soluzione dell'Esercizio 3.18 affinché il server eco possa servire separatamente ciascun client.

Progetti di programmazione

I progetti che seguono affrontano due distinti argomenti: il servizio di risoluzione dei nomi e la moltiplicazione fra matrici.

Progetto 1: Progetto sul servizio di risoluzione dei nomi

Un servizio di risoluzione dei nomi come DNS (*domain name system*, sistema dei nomi di dominio) può essere utilizzato per tradurre nomi IP in indirizzi IP. Ad esempio, nell'accedere all'host www.westminstercollege.edu si utilizza un servizio di risoluzione dei nomi per determinare l'indirizzo IP associato al nome www.westminstercollege.edu. Questo progetto consiste nello scrivere un servizio di risoluzione dei nomi multithread in Java utilizzando delle socket (Paragrafo 3.6.1).

Per risolvere i nomi IP la API `java.net` fornisce il seguente meccanismo:

```
InetAddress hostAddress =
        InetAddress.getByName("www.westminstercollege.edu");
String IPaddress = hostAddress.getHostAddress();
```

dove `getByName()` solleva una `UnknownHostException` se non è in grado di risolvere il nome host.

Il server

Il server ascolterà sulla porta 6052, in attesa della connessione dei client. Una volta instaurata una connessione con un client, il server erogherà il servizio di risoluzione dei nomi tramite un thread separato e ritornerà in ascolto sulla porta 6052 per ricevere ulteriori richieste di connessione dei client. Dopo la connessione al server, il client scriverà sulla socket il nome IP che vuole far risolvere dal server (ad esempio, www.westminstercollege.edu). Il thread del server leggerà questo nome IP dalla socket e tradurrà il suo indirizzo IP oppure, se non potrà localizzare l'indirizzo host, solleverà una `UnknownHostException`. Il server restituirà l'indirizzo IP al client o, in caso di `UnknownHostException`, scriverà il messaggio "`Unable to resolve host <host name>`." Ciò fatto, il server chiuderà la socket.

Il client

Inizialmente, scrivete solo l'applicazione server e connettetevi tramite telnet. Ad esempio, supponendo che il server stia funzionando su localhost, una sessione telnet apparirebbe come segue. (Riportiamo in grassetto quanto digitato dall'utente).

```
telnet localhost 6052
Connected to localhost.
Escape character is '^]'.
```

```
www.westminstercollege.edu
146.86.1.17
Connection closed by foreign host.
```

Usando inizialmente telnet in funzione di client, potrete eseguire più facilmente il debugging di problemi che possono insorgere con il server. Una volta che sarete sicuri del corretto funzionamento del server, potrete scrivere l'applicazione client. Il client riceverà come parametro il nome IP che deve essere risolto, aprirà una connessione socket al server, scriverà il nome IP da risolvere e leggerà la risposta restituita dal server. Ad esempio, se il client si chiama `NSClient`, potrà essere invocato come segue:

```
java NSClient www.westminstercollege.edu
```

e il server risponderà con il corrispondente indirizzo IP oppure con il messaggio "host sconosciuto". Il client chiuderà la sua connessione socket dopo aver restituito l'indirizzo IP.

Progetto 2: Progetto di moltiplicazione di matrici

Date due matrici *A* e *B*, dove *A* è una matrice con *M* righe e *K* colonne, mentre la matrice *B* contiene *K* righe e *N* colonne, il **prodotto** di *A* e *B* è la matrice *C*, contenente *M* righe e *N* colonne, definita come segue. L'elemento $C_{i,j}$ della matrice *C* alla riga *i* e colonna *j* è la somma dei prodotti degli elementi che appartengono alla riga *i* nella matrice *A* e alla colonna *j* nella matrice *B*. Quindi,

$$C_{i,j} = \sum_{n=1}^{K} A_{i,n} \times B_{n,j}$$

Posto, per esempio, che *A* sia una matrice 3 per 2 e B una matrice 2 per 3, l'elemento $C_{3,1}$ risulterebbe dalla somma di $A_{3,1} \times B_{1,1}$ e $A_{3,2} \times B_{2,1}$.

In questo progetto dovrete calcolare ogni elemento $C_{i,j}$ in un *thread di lavoro* separato. Sarà dunque opportuno creare $M \times N$ thread di lavorazione. Il thread principale – o genitore – inizializzerà le matrici *A* e *B*, allocando memoria in quantità sufficiente per la matrice *C*, che conterrà il prodotto delle matrici *A* e *B*. Tali matrici saranno dichiarate come dati globali, di modo che ciascun thread di lavorazione abbia accesso ad *A*, *B* e *C*.

Le matrici *A* e *B* possono essere inizializzate staticamente, come illustrato di seguito.

```
#define M 3
#define K 2
#define N 3
int A [M][K] = { {1,4}, {2,5}, {3,6} };
int B [K][N] = { {8,7,6}, {5,4,3} };
int C [M] [N];
```

In alternativa, possono essere riempite leggendo i valori da un file.

Passaggio dei parametri a ciascun thread

Il thread genitore darà luogo a $M \times N$ thread di lavoro, passando a ciascun di loro i valori della riga *i* e della colonna *j* di cui ha bisogno per determinare il prodotto della matrice. È quindi necessario passare due parametri a ogni thread. Il modo più semplice con Pthreads e

in Win32 è creare una struttura dati tramite `struct`. I membri di questa struttura sono *i* e *j*, e l'aspetto della struttura è il seguente:

```
/* struttura per il passaggio dei dati ai thread */
struct v
{
   int i; /* riga */
   int j; /* colonna */
};
```

I programmi Pthreads e Win32 useranno una strategia simile a quella riportata di seguito per creare i thread di lavoro.

```
/* Occorre creare M * N thread di lavoro */
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++ ) {
    struct v *data = (struct v *) malloc(sizeof(struct v));
    data->i = i;
    data->j = j;
    /* Adesso create il thread passandogli data come parametro */
  }
}
```

Il puntatore `data` verrà passato alla funzione `pthread_create()` (Pthreads) o alla funzione `CreateThread()` (Win32), che a sua volta lo passerà come parametro alla funzione da eseguire in qualità di thread separato.

La condivisione dei dati tra i thread di Java differisce da quanto appena visto. Una possibilità è di creare e inizializzare le matrici *A*, *B* e *C* nel thread principale (contenente il `main()`), che genererà quindi i thread di lavoro, passando le tre matrici – insieme alla riga *i* e alla colonna *j* – al costruttore di ogni thread di lavoro. Un thread di lavoro assume, dunque, l'aspetto riportato nella Figura 4.15.

Attesa per il completamento dei thread

Una volta che tutti i thread di lavoro abbiano terminato, il thread principale restituirà il prodotto contenuto nella matrice *C*. Esso dovrà perciò attendere la conclusione di tutti i thread di lavoro prima di poter emettere il valore del prodotto della matrice. Esistono molti modi diversi per sospendere un thread nell'attesa che gli altri si concludano. Il Paragrafo 4.3 spiega come un thread genitore possa attendere che il thread figlio termini usando le librerie Win32, Pthreads e Java. Win32 fornisce la funzione `WaitForSingleObject()`, mentre Pthreads e Java adoperano, rispettivamente, `pthread_join()` e `join()`. Negli esempi esaminati fin qui, però, il thread genitore attende che un solo thread figlio termini; lo svolgimento dell'esercizio implica, invece, l'attesa del completamento di più thread.

Nel Paragrafo 4.3.2 diamo conto della funzione `WaitForSingleObject()`, che ha lo scopo di attendere la terminazione di un singolo thread. Ma la API Win32 possiede anche la funzione `WaitForMultipleObjects()`, usata quando è necessario attendere che terminino diversi thread. Essa accetta quattro parametri.

```
public class WorkerThread implements Runnable
{
  private int row;
   private int col;
   private int[][] A;
   private int[][] B;
   private int[][] C;

 public WorkerThread(int row, int col, int[][] A,
  int[][] B, int[][] C) {
   this.row = row;
   this.col = col;
   this.A = A;
   this.B = B;
   this.C = C;
 }
 public void run() {
  /* calcolate il prodotto riga per colonna */
  /* e ponetelo in C[row] [col] */
 }
}
```

Figura 4.15 Codice Java di WorkerThread.

1. Il numero di oggetti da attendere.
2. Un puntatore al vettore degli oggetti.
3. Un flag che indichi se occorre attendere la terminazione di tutti i thread, o di almeno uno di loro.
4. Un limite di tempo massimo per l'attesa, o il valore convenzionale `INFINITE`.

Supponiamo, per esempio, che `THandles` sia un vettore di dimensione `N` contenente thread rappresentati da oggetti di tipo `HANDLE`. In questa ipotesi, il thread genitore può attendere che tutti i propri thread figli terminino con l'istruzione:

```
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);
```

Una soluzione comoda per attendere la terminazione di più thread usando `pthread_join()` di Pthreads o `join()` di Java è racchiudere questi comandi in un semplice ciclo `for`. Potreste, per esempio, attendere la terminazione di dieci thread usando il codice Pthreads riportato nella Figura 4.16. Nel caso di Java, il codice equivalente è mostrato nella Figura 4.17.

```
#define NUM_THREADS 10

/* Array dei thread di cui si attende la terminazione */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
  pthread_join(workers[i], NULL);
```

Figura 4.16 Codice Pthread per attendere la terminazione di dieci thread.

```
final static int NUM_THREADS = 10;

/* Array dei thread di cui si attende la terminazione */
Thread[] workers = new Thread[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++) {
  try {
    workers[i].join();
   }catch (InterruptedException ie) {}
}
```

Figura 4.17 Codice Java per attendere la terminazione di dieci thread.

4.7 Note bibliografiche

I thread hanno avuto una lunga evoluzione, a partire dalla "concorrenza a basso costo" nei linguaggi di programmazione, passando dai "processi leggeri", con i primi esempi tra i quali il sistema Thoth (Cheriton at al. [1979] e il sistema Pilot (Redell at al. [1980]). Binding [1985] ha descritto come i thread siano stati incorporati all'interno del kernel UNIX. Mach (Accetta et al. [1986], Tevanian et al. [1987a] e V (Cheriton [1988]) hanno fatto ampio uso dei thread. Alla fine di questo percorso, quasi tutti i principali sistemi operativi hanno implementato i thread in una forma o nell'altra.

Gli aspetti relativi alle prestazioni dei thread sono affrontati in [Anderson et al. 1989] e in [Anderson et al. 1991], dove si valutano le prestazioni dei thread a livello utente. [Bershad et al. 1990] trattano la combinazione di thread e RPC. [Engelschall 2000] illustra una tecnica per l'implementazione dei thread a livello utente. Un'analisi della dimensione ottimale dei gruppi di thread (thread pool) si trova in Ling et al. [2000]. Le attivazioni dello scheduler sono state presentate per la prima volta in Anderson et al. [1991], mentre Williams [2002] ha discusso le attivazioni dello scheduler nel sistema NetBSD. Altri meccanismi per la cooperazione fra la libreria dei thread a livello utente e del kernel sono esposti in [Marsh et al. 1991], [Govindan e Anderson 1991], [Draves et al. 1991] e [Black 1990]. [Zabatta e Young 1998] confrontano i thread del sistema Windows NT e di Solaris su un'architettura SMP. [Pinilla e Gill 2003] confrontano le prestazioni dei thread di Java sui sistemi Linux, Windows e Solaris.

[Vahalia 1996] tratta l'uso dei thread in diverse versioni di UNIX. [Mauro e McDougall 2007] descrivono i recenti sviluppi relativi all'uso dei thread nel kernel del Solaris. [Solomon e Russinovich 2000] descrivono la realizzazione dei thread nel sistema operativo Windows 2000. [Bovet e Cesati

2006] e Love [2004] descrivono il threading in Linux, mentre Singh [2007] tratta lo stesso argomento per Mac OS X.

Informazioni sulla programmazione Pthreads si trovano in [Lewis e Berg 1998], oltre che in [Butenhof 1997]. [Oaks e Wong 1999], [Lewis e Berg 2000] e [Holub 1998] analizzano la programmazione multithread nel linguaggio Java. Goetz et al. [2006] presentano una trattazione dettagliata della programmazione concorrente in Java. [Beveridge e Wiener 1997] e [Cohen e Woodring 1997] affrontano il tema della programmazione multithread con Win32.