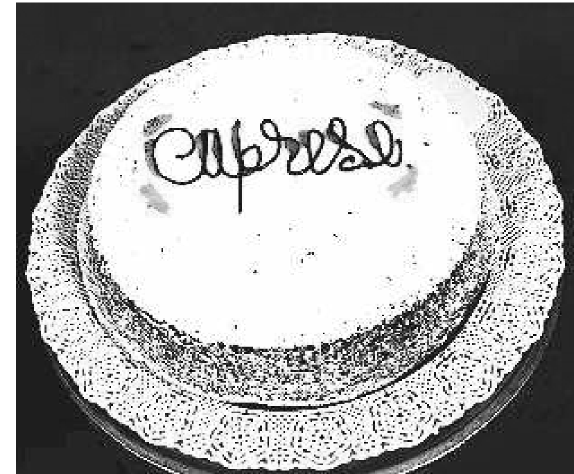
I Thread

- Introduzione
- Parallelismo e concorrenza
- Thread a livello utente e nucleo
- I modelli multithread
- Aspetti sui thread
- Pthreads
- Solaris 2 Threads
- Windows 2000 Threads
- Linux Threads
- Java Threads

La torta caprese



- Ingredienti
  - 5 uova
  - 8 cucchiai di zucchero
  - 1/8kg di burro
  - ¼kg di mandorle dolci
  - 100gr di cioccolato fondente
- Istruzioni:
  1. Tritare le mandorle;
  2. Sciogliere a bagnomaria il cioccolato con il burro;
  3. Unire le uova allo zucchero;
  4. Aggiungere all'impasto il cioccolato fuso;
  5. Aggiungere all'impasto le mandorle tritate;
  6. Mettere il tutto in una pirofila da forno di forma circolare, porla nel forno;
  7. Cuocere per circa 1 ora a 150° C;
  8. Controllare la cottura con uno stecchino.

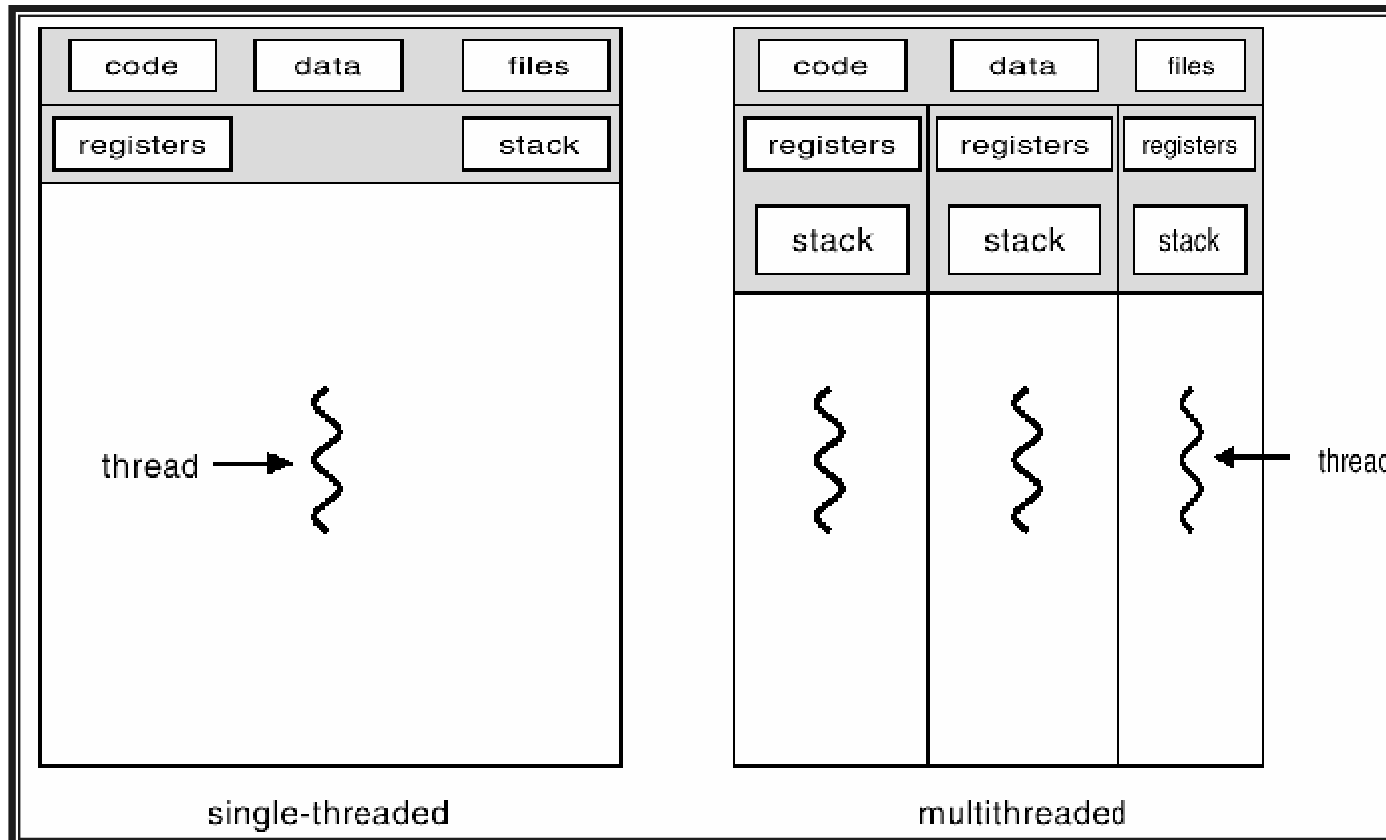
L’analogia

- L’informatico appassionato di cucina legge la ricetta della torta caprese ed esegue le istruzioni, così come una CPU esegue le istruzioni di un programma.
- Egli è in grado di eseguire una singola azione alla volta in sequenza.
- Una CPU che esegue le istruzioni di un programma si comporta alla stessa maniera: le istruzioni passano attraverso il contatore una alla volta e il processore le esegue.
- Come si può migliorare questo comportamento?

L’analogia

- Una ricetta contiene differenti istruzioni, che possono essere eseguite indipendentemente l’una dall’altra.
- Il tempo per eseguire le istruzioni della ricetta può essere migliorato mediante l’esecuzione concorrente o parallela delle singole istruzioni (sciogliere il burro, tritare le mandorle e unire le uova allo zucchero).
- Un programma **multithreaded** contiene due o più flussi di controllo (*thread)*: se un compito può essere suddiviso in due o più sottocompiti indipendenti, l’uso di flussi di controllo multipli è possibile.

Processi con sequenze di esecuzione singole e multiple



Parallelismo e concorrenza

- Un programma può migliorare significativamente le prestazioni attraverso l'uso di thread multipli
  - **Esecuzione concorrente**: i thread sono *progrediscono* nello stesso momento (mentre il burro si scoglie, si tritano le mandorle).
  - **Parallelismo**: i thread sono in esecuzione simultaneamente su più processori. (se ci sono due o più cuochi, più passi possono essere eseguiti contemporaneamente).
- Sebbene in genere parallelismo e concorrenza aumentino le prestazioni, l'uso dei thread ha un costo in termini di:
  - creazione,
  - scheduling,
  - sincronizzazione e
  - terminazione.

Limiti

- Se il compiti da far eseguire ad un thread hanno una durata troppo breve, il costo aggiuntivo di gestione può essere troppo oneroso.
- I thread accedono a risorse condivise, come dati globali e file aperti all’interno del processo: tale accesso può necessitare di sincronizzazione => aumento dei tempi di esecuzione.
- Ci sono compiti che non possono essere eseguiti in parallelo, e devono essere eseguiti *serialmente.*

Benefici

- **Migliori tempi di risposta**
  - Maggiore velocità di risposta delle applicazioni
- **Maggiore throughput**
  - Maggiore lavoro eseguito per unità di tempo
- **Migliore utilizzo delle risorse**
  - La condivisione delle risorse è più rapida e più economica
- **Rapidità d’azione**
  - Operazioni di gestione più rapide
- **Migliore utilizzo delle architetture**
  - Efficace utilizzo dei sistemi multiprocessori
- **Naturale strutturazione dei programmi**
  - Il parallelismo logico è semplice da esprimere

Contesto dei thread

- Un thread esiste all’interno del contesto di un processo.
  - **Struttura del thread:** contiene l’identificativo del thread, la politica di scheduling e la priorità, e la maschera dei segnali.
  - **Struttura utente:** contatore di programma.
  - **Stack,**
  - **Area dati privata,**
  - **Attributi:** caratteristiche specifiche del thread
  - **Istruzioni da eseguire:** una funzione all’interno di un programma.
- Un thread, come un processo, può essere eseguito in modo utente e modo di sistema.
- Possiede gli stessi stati di un processo.

Thread di livello utente e nucleo

- La gestione dei thread può avvenire sia al livello d'utente, sia al livello del nucleo.
  - I **thread a livello utente** sono gestiti come uno strato sopra il nucleo e realizzati tramite una libreria.
    - La loro creazione avviene nello spazio dell'utente,
    - Se il nucleo è a singolo thread, ogni chiamata bloccante causa il blocco dell'intero sistema.
    - Esempi: POSIX *Pthreads,* Mach *C-threads,* Solaris *UI-threads*
  - I **thread a livello nucleo** sono gestiti direttamente dal sistema operativo.
    - La loro creazione è affidata al nucleo.
    - Se un thread esegue una chiamata bloccante, il nucleo può attivare un altro thread d'applicazione.
    - In presenza di più CPU, il nucleo fa eseguire i thread da unità d'elaborazione differenti.
    - Esempi: Windows 95/98/NT/2000, Solaris 2, Tru64 UNIX, BeOS, Linux.

Thread vincolati e svincolati

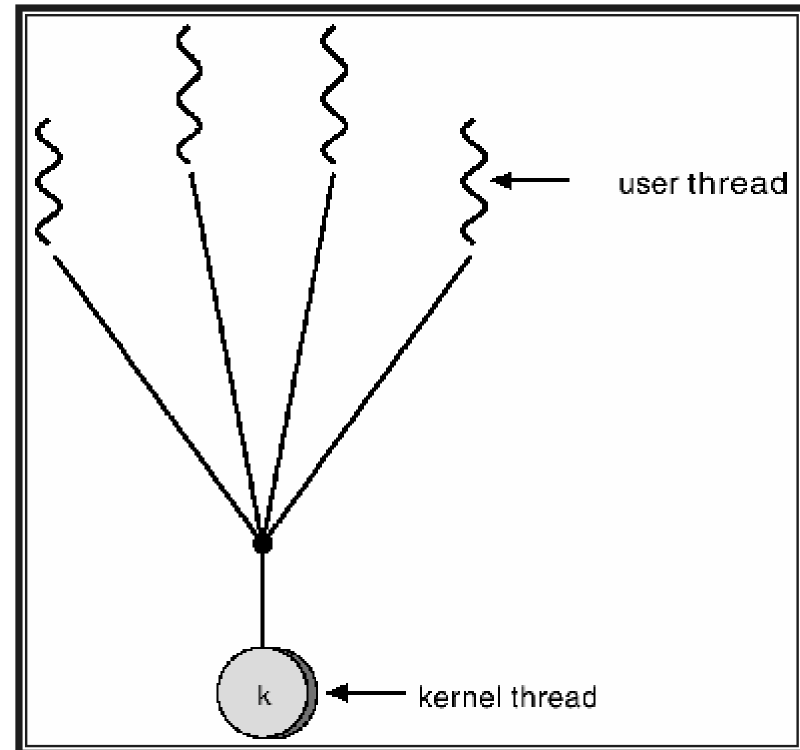
- Un **thread vincolato** è un thread utente che è associato in maniera permanente ad un thread del nucleo.
- Un **thread svincolato** è un thread utente non associato in maniera permanente a ciascun thread del nucleo.
- Ciascun processo può avere molti thread a livello utente, che sono selezionati dallo scheduler della libreria dei thread.
- Ciascun thread del nucleo è scelto per l’esecuzione dallo scheduler.

Modelli di programmazione multithread

- Un sistema operativo in grado di gestire processi a singolo thread e thread multipli (*multithread*) è un sistema **orientato ai thread** (*thread model*).
- Esistono differenti modelli:
  - Molti a uno (M x 1)
  - Uno a uno (1 x 1)
  - Molti a molti (M x N)
- Questi nomi forniscono una buona descrizione di come ciascun modello supporti un’applicazione multithread.

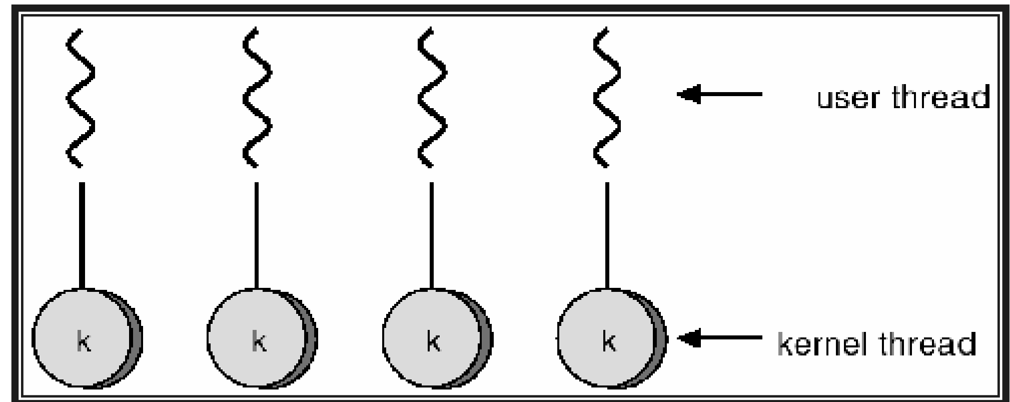
Molti a uno

- M x 1: molti thread a livello d’utente corrispondono ad un singolo thread del nucleo.
- Implementato nello spazio utente, non ha bisogno di modifiche al kernel.
- Lo scheduling è gestito in modo utente
- Vantaggi:
  - Concorrenza
  - Operazioni di gestione dei thread rapide
  - Conservazione delle risorse di sistema
  - Struttura della programmazione naturale
  - Pronti per l’esecuzione parallela.
- Svantaggi:
  - Parallelismo fisico non supportato
  - Blocco dei thread in modo di sistema
  - Profiling per singolo thread non disponibile



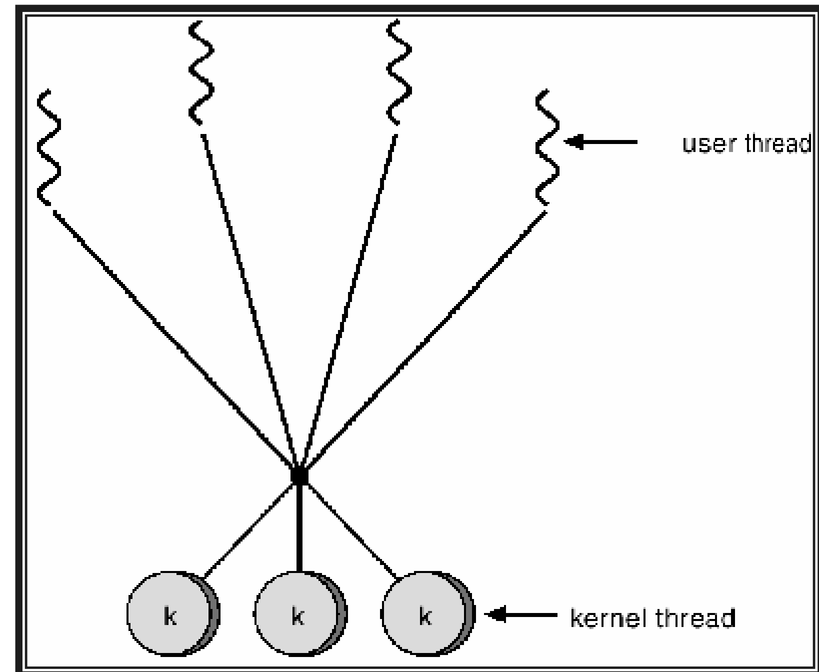
Uno a uno

- Il modello 1 x 1 supporta sia processi a singolo thread sia multithread; è implementato nello spazio del nucleo.
- Ciascun thread di livello utente è vincolato ad un thread del nucleo.
- Lo scheduling è implementato nel nucleo.
- Vantaggi:
  - Tutti i vantaggi del modello M x 1,
  - Esecuzione parallela,
  - Informazioni di profiling.
- Svantaggi:
  - Maggiore costo di gestione
  - Necessità di più risorse del kernel
- Esempi: Windows 95/98/NT/2000, OS/2



Molti a molti

- Il modello M x N supporta sia processi a singolo thread sia multithread; è implementato sia in spazio utente, sia del nucleo *(kernel threads).*
- Mette in corrispondenza più thread del livello utente con un numero <= di thread del nucleo.
- Lo scheduling è implementato nel nucleo (ciascun thread su CPU disponibili) + scheduling in spazio utente (ciascun thread utente su un thread del nucleo disponibile).
- Vantaggi:
  - Tutti quelli di 1 x 1 e M x 1
- Svantaggi:
  - Tutti quelli di 1 x 1 e M x 1
  - Maggiore complessità
- Esempi: Solaris 2, HP-UX, True 64, NT/2K (*ThreadFiber* package)



Esempio: il gestore delle finestre*

- In un programma di gestione delle finestre vi sono migliaia di *widgets*. Ciascun widget ha due thread:
  - Uno gestisce l'interfaccia grafica del widget,
  - L'altro le azioni appropriate basate sull'input dell'utente
- Se tutti i thread utenti fossero vincolati a thread del nucleo, perché solo alcuni widget sono contemporaneamente attivi.
- Si può:
  - Vincolare il thread che controlla i movimenti del mouse, ad uno del nucleo ad alta priorità, così che il movimento del puntatore sia continuo.
  - Altri thread utente possono essere associati ai thread del nucleo disponibili: poiché solo pochi widget sono attivi insieme, è possibile che le richieste possano essere soddisfatte.

* Posix 1003.1c

Esempio: il database server*

- Un server di un database con migliaia di client deve gestire le loro richieste.
- Se solo un numero limitato di client è attivo allo stesso momento, è inutile vincolare ciascun client ad un thread del nucleo.
- Conviene avere un numero limitato di thread vincolati dedicati a soddisfare le richieste per un numero di thread client svincolato.

* Posix 1003.1c

Programmazione dei thread

- Chiamate a **fork** ed **exec**
  - Se un thread invoca una chiamata a **fork**, il nuovo processo può contenere un duplicato di tutti i thread, oppure no.
  - Se un thread invoa una chiamata ad **exec**, il nuovo processo sostituisce l'originale.
- Cancellazione dei thread
  - **Cancellazione sincrona**: il bersaglio è immediatamente terminato
  - **Cancellazione asincrona**: il bersaglio periodicamente controlla se terminare.
- Gestione dei segnali
  - Ricezione sincrona ed asincrona
  - Gestione dei segnali
- Gruppi di Thread
- Dati specifici

Chiamate di sistema fork e exec

- In un programma multithread la semantica di **fork** ed **exec** cambia: se un thread invoca **fork**, il nuovo processo può contenere un duplicato di tutti i thread o solo del chiamante.
- Alcuni sistemi UNIX includono entrambe le versioni.
- Un nuovo processo che invoca **exec**, sostituisce completamente (inclusi tutti i thread) il vecchio.
- L’uso delle due versioni della **fork** dipende dall’applicazione: se dopo la **fork**, il processo figlio invoca una **exec**, non è necessario duplicare tutti i thread.

Cancellazione

- **Cancellare un thread** significa terminare un thread prima che completi il suo compito. Il thread da cancellare è detto **thread bersaglio**.
- Ad esempio, in un web browser il caricamento delle pagine è gestito da un thread separato: quando l'utente preme il pulsante di interruzione, il thread viene cancellato.
- La cancellazione può avvenire:
  - **Cancellazione sincrona**: il bersaglio è immediatamente terminato,
  - **Cancellazione asincrona**: il bersaglio periodicamente controlla se terminare.
- Nel caso di cancellazione sincrona, il thread può non rilasciare tutte le risorse acquisite.
- Poiché l'API Pthreads permette la cancellazione differita, ogni sistema operativo che la impiega, implicitamente la permette.

Gestione dei segnali

- Nei sistemi UNIX si utilizzano i **segnali** per comunicare ai processi il verificarsi di determinati eventi, intercettati dall’hardware (divisione per zero), generati dai processi (chiamata a **kill**, **alarm**,…) o dagli utenti (<CTRL-C>).
- Ogni segnale può essere gestito:
  - tramite un gestore predefinito dei segnali,
  - tramite un gestore definito dall’utente.
- Per i processi multithread è possibile:
  1. inviare il segnale al thread cui si riferisce,
  2. inviare il segnale ad ogni thread,
  3. inviare il segnale a thread specifici,
  4. definire un thread specifico per ricevere tutti i segnali.
- Ad esempio, in Solaris 2 si usa la quarta alternativa.
- In Windows 2K i segnali possono essere emulati tramite le **chiamate a procedure asincrone** (*asynchronous procedure call – APC*)
- Queste permettono ad un thread di livello utente di specificare la funzione da chiamare quando esso riceve la notifica di un evento.

Gruppi di thread

- In applicazioni multithread, un nuovo thread viene creato per eseguire un determinato compito: il tempo per creare un thread è molto inferiore al tempo per creare un processo.
- Se si vuole che ogni nuovo compito venga eseguito da un nuovo thread, le risorse del sistema potrebbero esaurirsi.
- Per risolvere questo problema si possono utilizzare **gruppi di thread**, che vengono creati ed attendono di eseguire il lavoro che gli sarà richiesto.
- Se il gruppo non contiene alcun thread disponibile, sarà necessario attendere il rientro di un thread.
  - Se il thread già esiste, non è necessario attendere la sua creazione;
  - un gruppo di thread limita il numero di thread esistenti in un certo istante.
- Il numero di thread in un gruppo viene determinato euristicamente.

Dati specifici

- Uno dei vantaggi della programmazione multithread è che i thread che appartengono ad uno stesso processo condividono i dati.
- In particolari situazioni può essere necessario avere a disposizione dei **dati specifici del thread**.
- La maggior parte delle librerie di thread e l'ambiente Java consentono l'impiego di dati specifici di thread.

Pthreads

- IEEE Portable Operating System Interface (POSIX) 1003.1c è la parte dello standard POSIX che copre i thread.
- Definisce le funzioni e le interfacce di programmazione alle applicazioni (API) per la gestione e sincronizzazione dei thread.
- Si tratta d una *definizione* del comportamento dei thread, non di una *realizzazione.*
- Le implementazioni sono limitate ai sistemi UNIX; i sistemi operativi Windows non dispongono in genere di una libreria Pthreads, sebbene vi siano versioni di pubblico dominio.

Esempio: Somma di n interi

```
/**
 * A pthread program illustrating how to
 * create a simple thread and some of the pthread API.
 * Usage on Solaris/Linux/Mac OS X:
 * gcc thrd.c -lpthread
 * a.out <number>
 **/

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

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

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

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();
}

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

/* get the default attributes */
```

Esempio: Somma di n interi

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

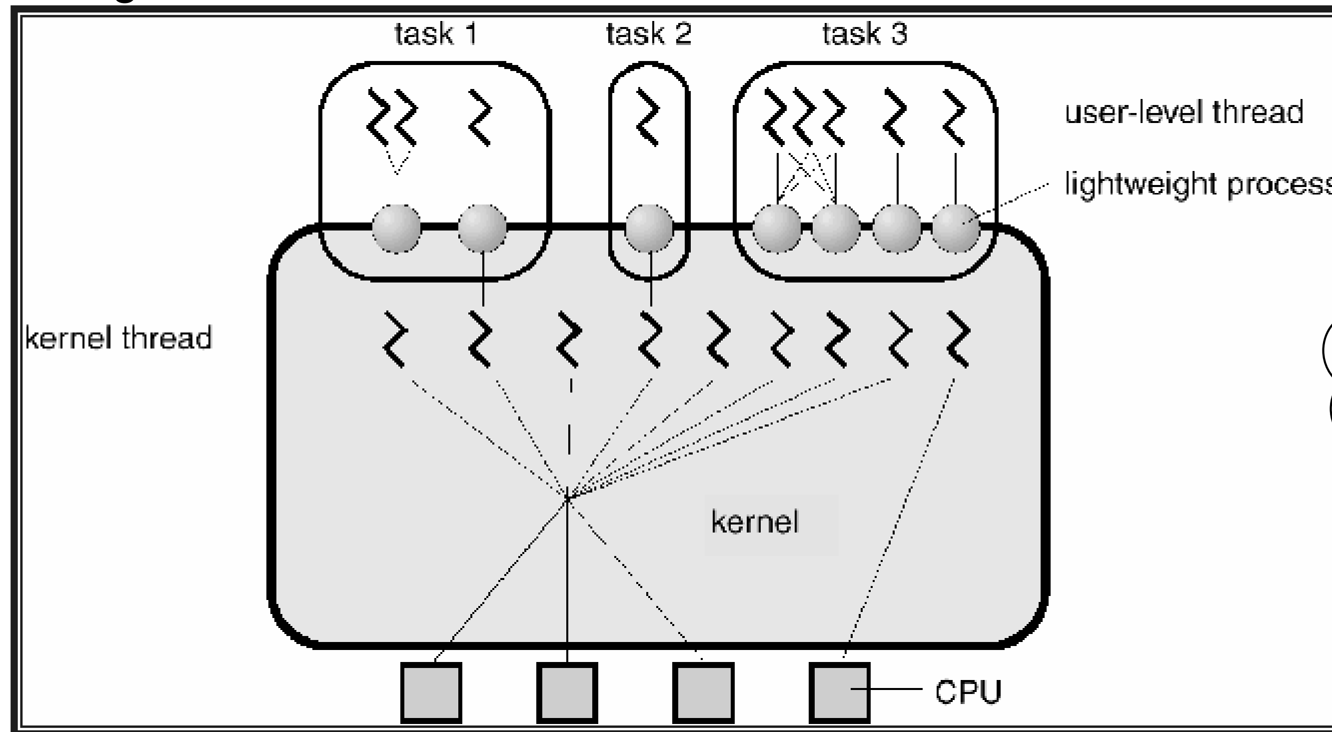
```
/* get the default attributes */
pthread_attr_init(&attr);

/* 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 upper = atoi(param);
int i;
sum = 0;

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

Thread in Solaris 2

- Sun Solaris 2 prevede thread del livello utente e del livello del nucleo, SMP e lo scheduling per elaborazioni in tempo reale.
- Per i thread a livello utente sono disponibili le librerie Pthread e UI-threads.
- Solaris 2 definisce un livello intermedio di thread: tra i thread di livello utente e quelli del nucleo ci sono i **processi leggeri** (*lightweight processes - LWP*).
- La libreria dei thread mette dinamicamente in corrispondenza i thread di livello utente con gli LWP di quel processo; solo i thread associati ad un LWP possono essere in esecuzione.

Thread in Solaris 2

- I thread ordinari del nucleo eseguono le operazioni all'interno del nucleo; ogni LWP ha un thread del nucleo associato.
- Alcuni thread del nucleo sono eseguiti per conto del nucleo stesso (es. gestione delle richieste d'accesso ad un disco) e non sono associati ad alcun LWP.
- Solo ai thread del nucleo viene applicato l'algoritmo di scheduling.



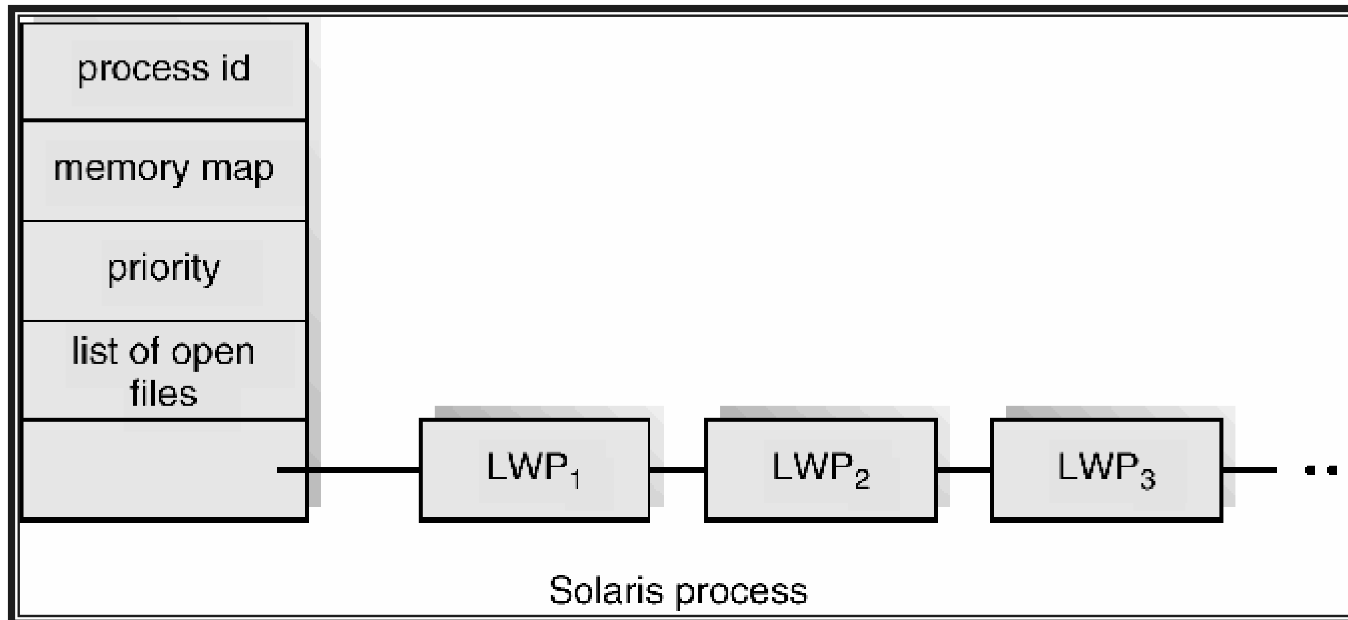
Il modello è M x N

Thread in Solaris 2

- Ciascun processo può avere molti thread del livello utente, che possono essere selezionati dallo scheduler e associati agli LWP dalla libreria senza l'intervento del nucleo.
- Ciascun LWP è associato ad un thread del livello del nucleo. Molti LWP potrebbero essere parte dello stesso processo, ma sono richiesti solo se un thread deve comunicare col nucleo.
- I thread del nucleo sono scelti per l'esecuzione dallo scheduler: se un thread si blocca, la CPU può eseguire un altro thread del nucleo.
- Se un thread del nucleo si blocca, si blocca anche l'LWP ad esso associato, e di conseguenza il thread a livello utente.
- Se più LWP sono associati ad uno stesso processo, un altro LWP tra questi può andare in esecuzione.

Thread in Solaris 2

- I thread di Solaris 2 utilizzano le seguenti strutture dati:
  - Ogni **thread di livello utente** contiene in spazio utente: un identificatore, un insieme di registri, la pila, la priorità.
  - Ogni **LWP** è una struttura dati del nucleo che contiene un insieme di registri relativi al thread che esegue, risorse di memoria e informazioni di contabilizzazione.
  - Ogni **thread del nucleo** ha una struttura dati con le copie dei registri del nucleo, un puntatore all'LWP, informazioni su scheduling e priorità, e una pila.



Thread in Windows 2000

- Un programma per l'ambiente Windows si esegue come un processo separato e ogni processo può contenere uno o più thread.
- Windows 2K impiega il modello 1 x 1, in cui ogni thread di livello utente è associato ad un thread del nucleo. E' disponibile la libreria *fiber* che offre funzioni che realizzano il modello M x N.

- Ciascun thread contiene: un identificativo di thread, un insieme di registri, pile utente e una del nucleo separate e un'area dati privata.
- Le strutture principali di un thread includono:
  - ETHREAD (*executive thread block)*
  - KTHREAD (*kernel thread block)*
  - TEB (*thread environment block*)

Thread in LINUX

- Presenti a partire dalla versione 2.2.
- Affianco alla chiamata di sistema **fork,** per la creazione di processi, è presente **clone**, che crea un processo distinto che condivide lo spazio di indirizzi del processo chiamante.
- La condivisione è permessa dal modo in cui un processo è rappresentato nel nucleo: per ogni processo è presente un’unica struttura dati, con puntatori ai dati effettivi.
- La chiamata a **clone** riceve come parametro un insieme di cinque indicatori che specifica quanto deve essere condiviso con il figlio.
- Il sistema non distingue tra processi e thread e in generale si usa il termine *task*.
- Sono disponibili varie versioni della libreria Pthread.

La chiamata clone di LINUX

- Il kernel di Linux usa la funzione **clone** per creare nuovi processi. I flag controllano quali risorse genitore e figlio condividono.
- Genitore e figlio possono condividere da tutto (memoria, gestori dei segnali, file aperti,…) a niente. Mentre con la **fork** il figlio eredita le risorse del padre, con **clone** è possibile non condividere nulla.
- Un programma può chiamare direttamente **clone** per produrre un programma multithread; ciò rende il programma specifico per Linux, poiché non è conforme ad alcuno standard esterno.
- Le librerie per i thread utilizzano la **clone** al loro interno, e rendono trasparente gli aggiornamenti del kernel agli utenti.

La libreria LinuxThreads

- L’implementazione LinuxThreads dello standard POSIX, originariamente scritta da Xavier Leroy, è stata dominante per anni e incorporata nella **glibc**.
- I problemi principali sono:
  - Compatibilità con lo standard POSIX,
  - Problemi di prestazioni, in termini di scalabilità.
- Tali problemi sono dovuti alla scelta di utilizzare il modello 1 x 1 come base per l’implementazione, che rende difficile il porting di applicazioni esterne su Linux.
- I problemi di prestazioni si riferiscono al loro degrado al crescere del numero di thread (es. assegnazione del **tid**).
- I sostenitori del modello M x N possono utilizzare la libreria *New Generation POSIX Threads* (NGPT), mentre gli assertori del modello 1 x 1 la *Native POSIX Thread Library* (NPTL).

Thread in Java

- La gestione e la creazione dei thread avviene a livello del linguaggio di programmazione.
- Poiché sono gestiti dalla macchina virtuale (JVM), non si possono considerare né di livello utente, né di nucleo.
- Tutti i programmi scritti in Java prevedono almeno un thread di controllo: anche un programma composto dal solo metodo **main** è eseguito dalla JVM come un unico thread.
- Per creare esplicitamente un thread si può creare una nuova classe derivata dalla classe **Thread** e soprascrivere il metodo **run**.
- Un oggetto di questa classe derivata è eseguito come un thread distinto nella JVM.
- La maniera in cui i thread della JVM corrispondono ai servizi del sistema operativo sottostante dipende dalla specifica implementazione.

Esempio: Somma di n interi

```
public class Summation extends Thread
{
        public Summation(int n) {
                upper = n;
        }

        public void run() {
                int sum = 0;

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

                System.out.println("The  summation  of  "  +
  upper + " is " + sum);
        }

        private int upper;
}
```

Esempio (cont.)

```
public class ThreadTester
{
   public static void main(String[] args) {
        if (args.length > 0) {
               if (Integer.parseInt(args[0]) < 0)
                      throw                                        new
   IllegalArgumentException(args[0]   +   "  must   be   non-
   negative.");
               else {
                      Summation summationThread =
                             new
   Summation(Integer.parseInt(args[0]));
                      summationThread.start();
               }
        }
        else
               System.err.println("Usage: Summation <integer
   value>");
   }
}
```

Sommario

- Un thread è un percorso di controllo d'esecuzione all'interno di un processo. In un processo multithread vi sono più percorsi di controllo che condividono lo stesso spazio di indirizzi.
- I vantaggi della programmazione multithread includono un miglioramento nei tempi di risposta, la condivisione delle risorse e la capacità di sfruttare le architetture multiprocessore.
- I thread di livello utente sono gestiti dal programmatore ed ignoti al nucleo. Il nucleo gestisce i thread del nucleo. I thread utente richiedono in genere minor tempo per creazione e gestione.
- Esistono diversi modelli che descrivono le relazioni tra thread del livello utente e del livello del nucleo: M x 1, 1 x 1, M x N.
- La semantica delle chiamate di sistema, quando invocate all'interno di un thread, può cambiare.
- Molti sistemi operativi moderni prevedono la gestione dei thread a livello del nucleo: Win NT, 2K, Solaris 2, LINUX.
- Pthread offre un'insieme di API per gestire i thread al livello utente.
- Java ha un'API simile; i thread sono gestiti dalla JVM.