

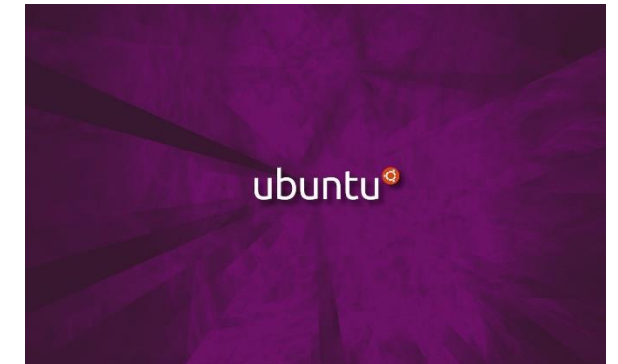
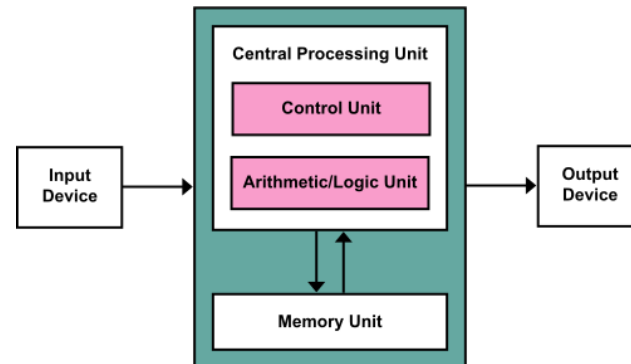
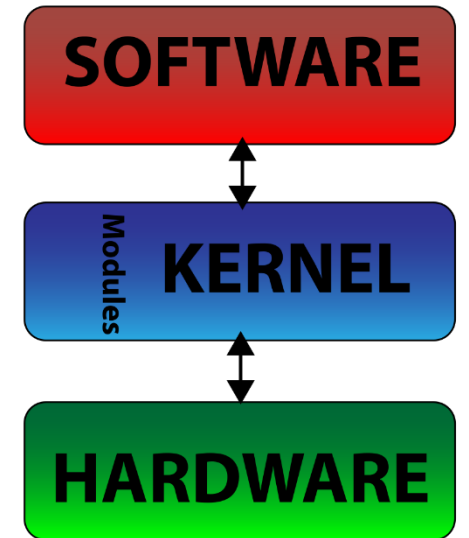
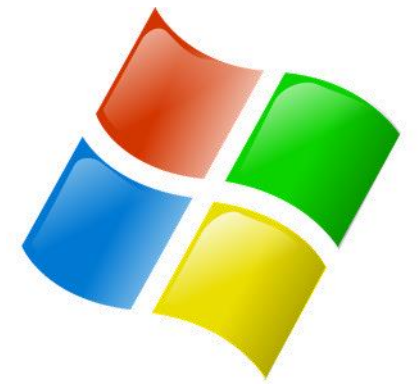


**UNIVERSITÀ DEGLI STUDI  
DELLA BASILICATA**

## *Corso di Sistemi Operativi*

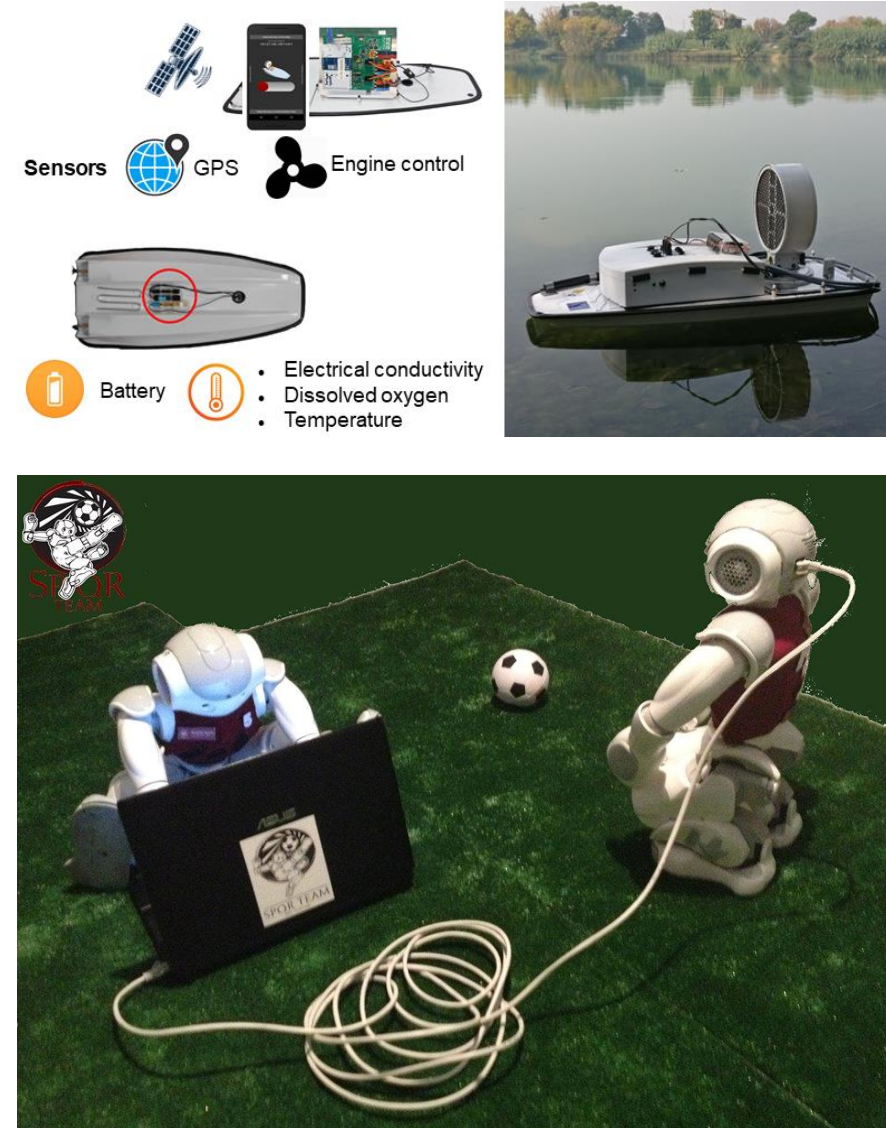
# Thread

Docente:  
**Domenico Daniele  
Bloisi**



# Domenico Daniele Bloisi

- Ricercatore RTD B  
Dipartimento di Matematica, Informatica  
ed Economia  
Università degli studi della Basilicata  
<http://web.unibas.it/bloisi>
- SPQR Robot Soccer Team  
Dipartimento di Informatica, Automatica  
e Gestionale Università degli studi di  
Roma “La Sapienza”  
<http://spqr.diag.uniroma1.it>



# Informazioni sul corso

---

- Home page del corso:  
<http://web.unibas.it/bloisi/corsi/sistemi-operativi.html>
- Docente: Domenico Daniele Bloisi
- Periodo: I semestre ottobre 2021 – febbraio 2022
  - Lunedì dalle 15:00 alle 17:00 (Aula Leonardo)
  - Martedì dalle 12:30 alle 14:00 (Aula 1)

# Ricevimento

---

- Durante il periodo delle lezioni:  
Martedì dalle 10:00 alle 11:30 → Edificio 3D, II piano, stanza 15  
**Si invitano gli studenti a controllare regolarmente la [bacheca degli avvisi](#) per eventuali variazioni**
- Al di fuori del periodo delle lezioni:  
da concordare con il docente tramite email

Per prenotare un appuntamento inviare  
una email a  
[domenico.bloisi@unibas.it](mailto:domenico.bloisi@unibas.it)



# Programma – Sistemi Operativi

---

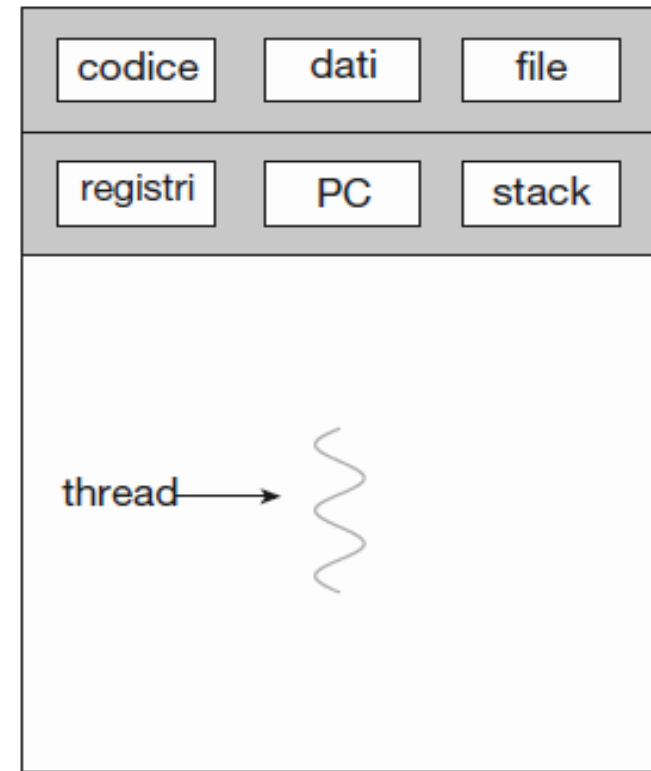
- Introduzione ai sistemi operativi
- Gestione dei processi
- Sincronizzazione dei processi
- Gestione della memoria centrale
- Gestione della memoria di massa
- File system
- Sicurezza e protezione

# Definizione di thread

---

Un **thread** è l'unità di base d'uso della CPU e comprende

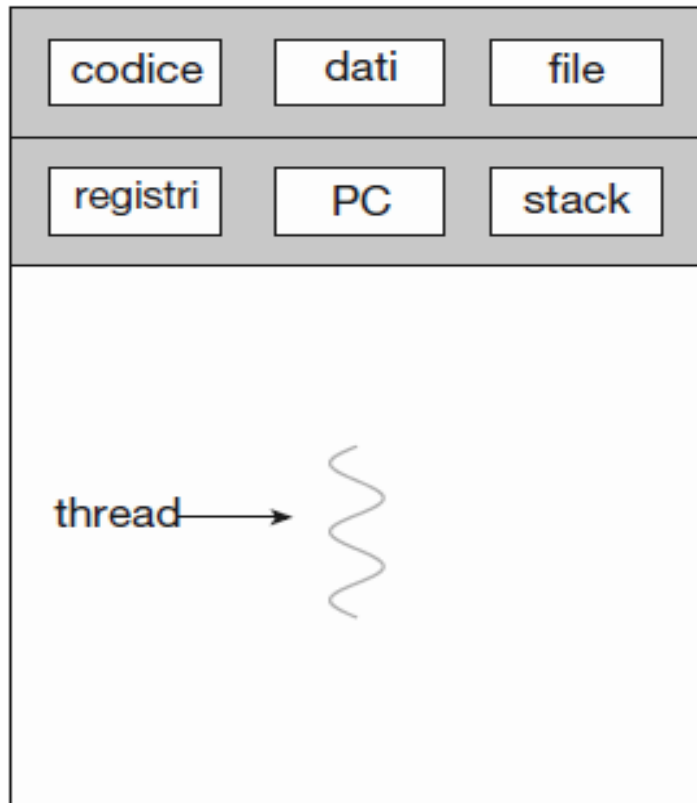
- un **identificatore di thread (ID)**
- un **contatore di programma (PC)**
- un insieme di **registri**
- una **pila (*stack*)**



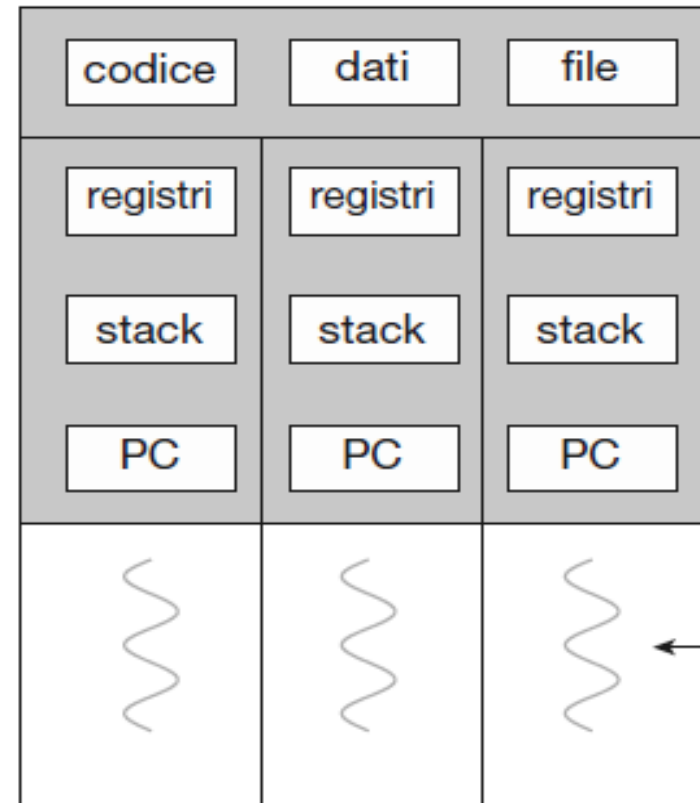
processo a singolo thread

# Singlethread vs. Multithread process

differenza tra un processo tradizionale, **a singolo thread**, e uno **multithread**.



processo a singolo thread



memoria  
condivisa

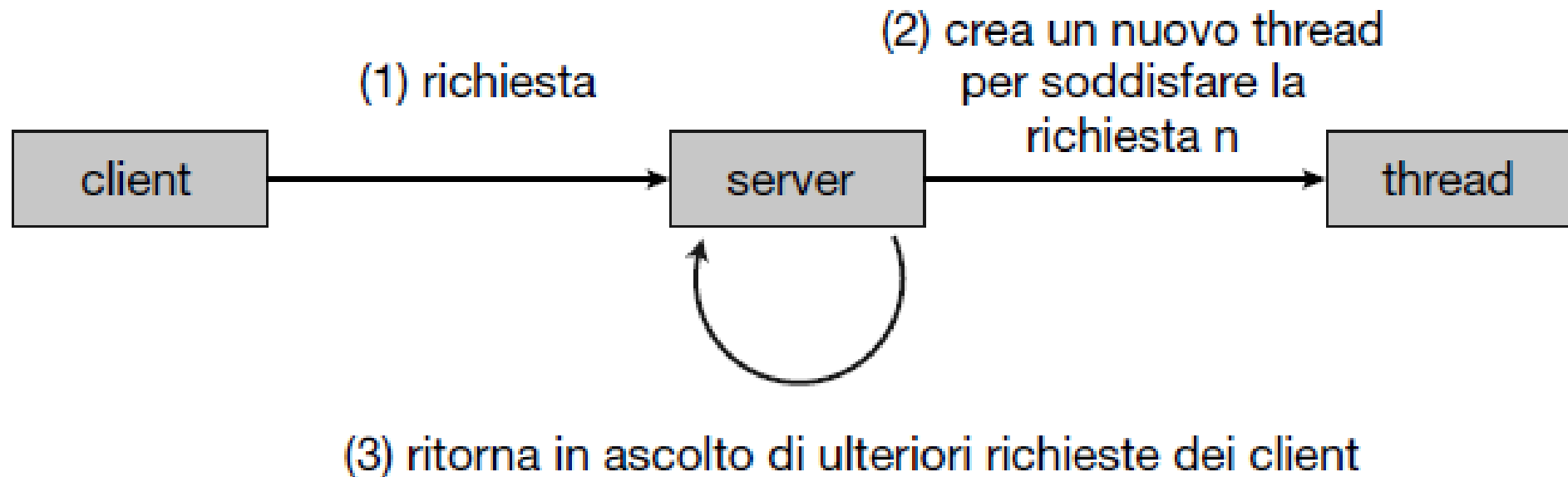
processo multithread

**Figura 4.1** Processi a singolo thread e multithread.

# Processi multithread

La maggior parte delle applicazioni per i moderni computer è **multithread**

Esempio web server:



**Figura 4.2** Architettura di server multithread.



# Programmazione multithread

---

## VANTAGGI

Tempo di  
risposta

Condivisione  
delle risorse

Economia

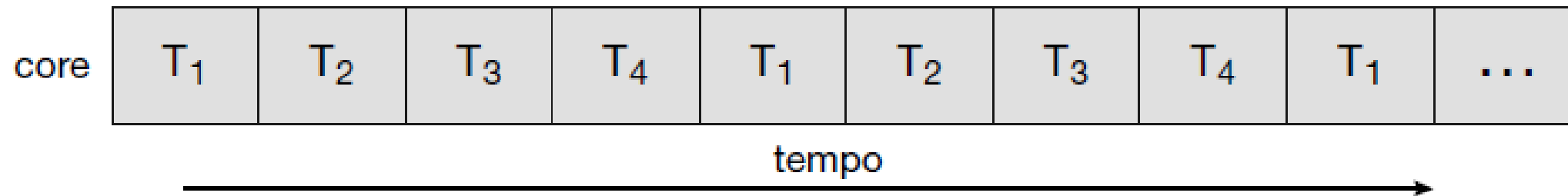
Scalabilità

# Concorrenza vs. Parallelismo

sistema concorrente



supporta più task  
permettendo  
a ciascuno di progredire  
nell'esecuzione

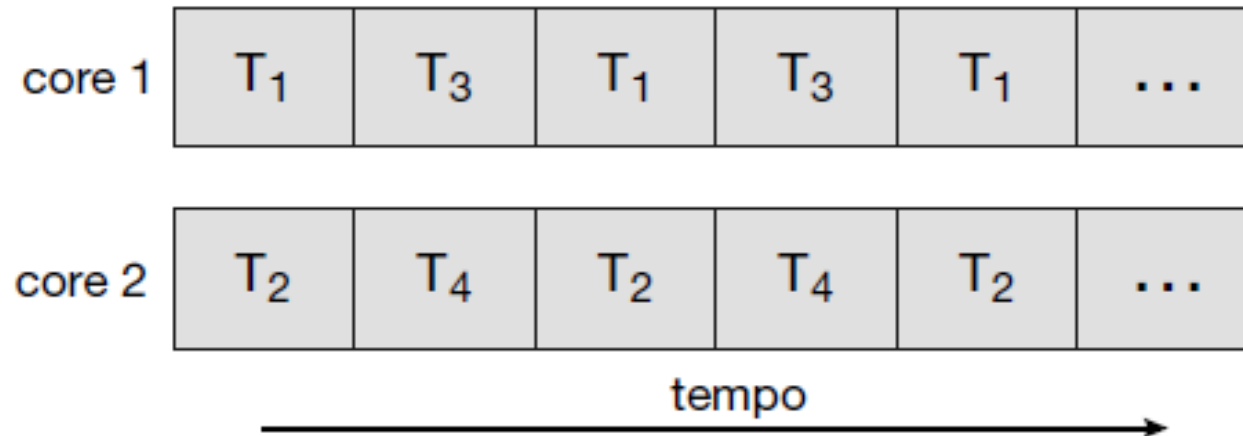


**Figura 4.3** Esecuzione concorrente su un sistema a singolo core.

# Esecuzione concorrente su multicore

Su un **sistema multicore** per “esecuzione concorrente” si intende che i thread possono funzionare *in parallelo*, dal momento che il sistema può assegnare thread diversi a ciascun core.

**sistema parallelo**  può eseguire simultaneamente più di un task



**Figura 4.4** Esecuzione parallela su un sistema multicore.

# Sfide nella programmazione multithread

---



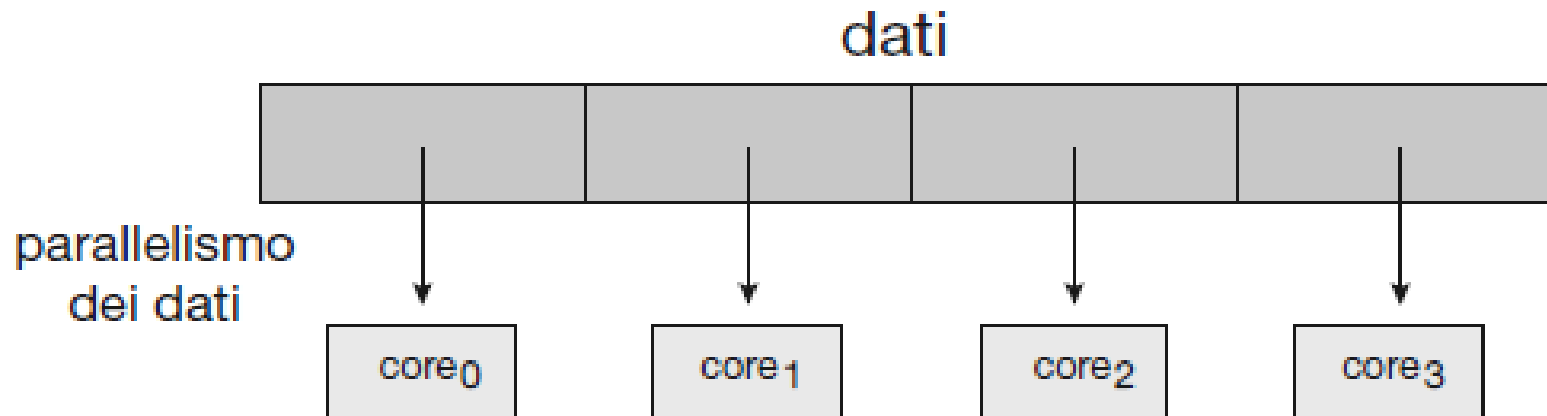
# Parallelismo dei dati

---

parallelismo dei dati



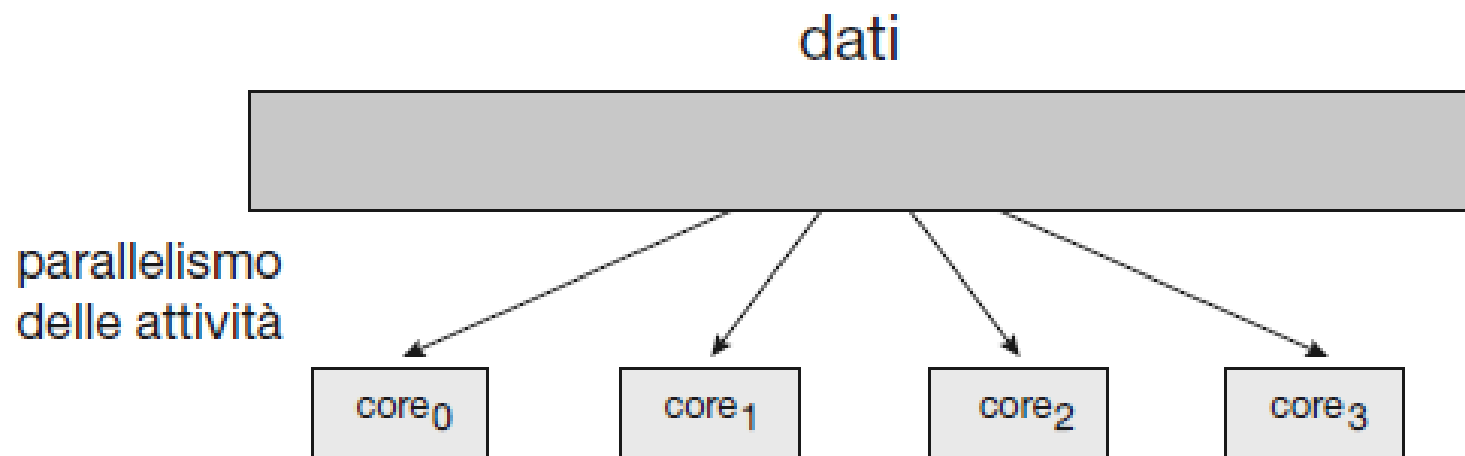
distribuzione di sottoinsiemi dei dati su più core di elaborazione ed esecuzione della stessa operazione su ogni core



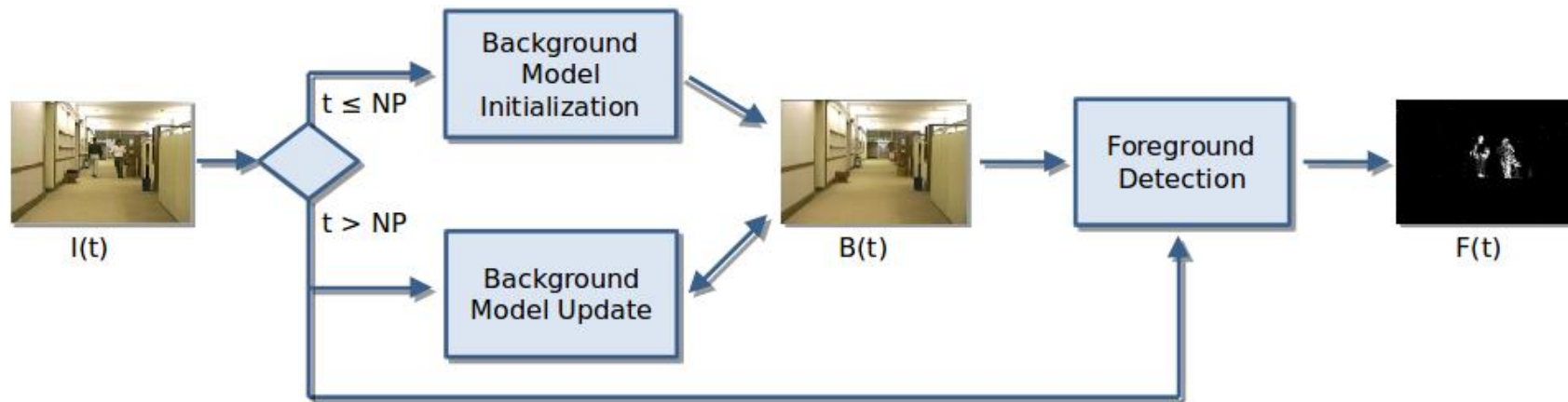
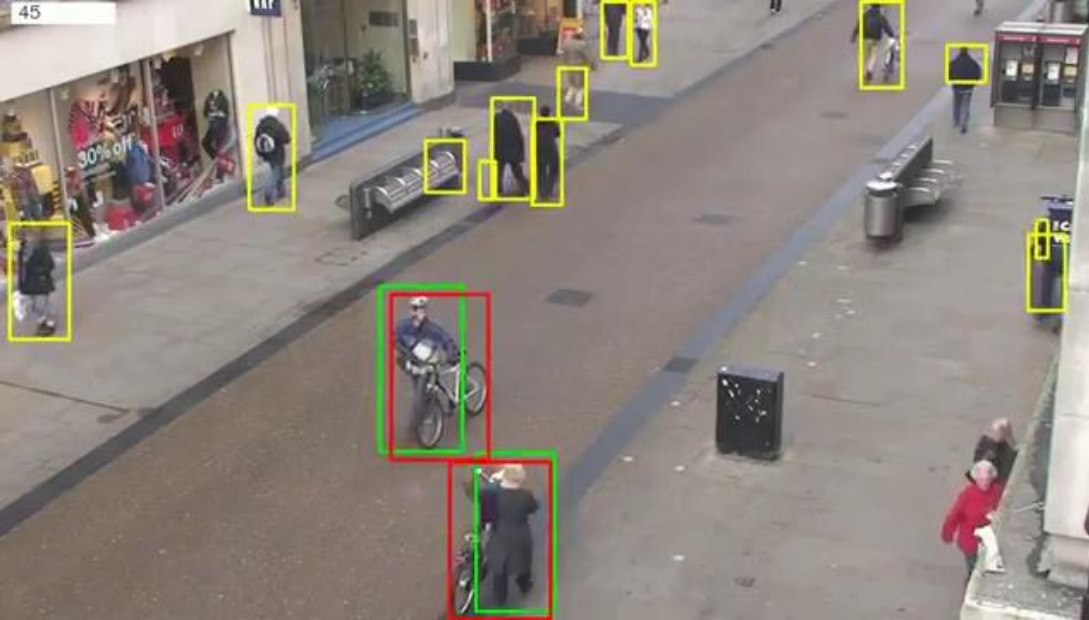
# Parallelismo delle attività

---

**parallelismo delle attività** ➡ distribuzione di attività (thread) e non di dati, su più core

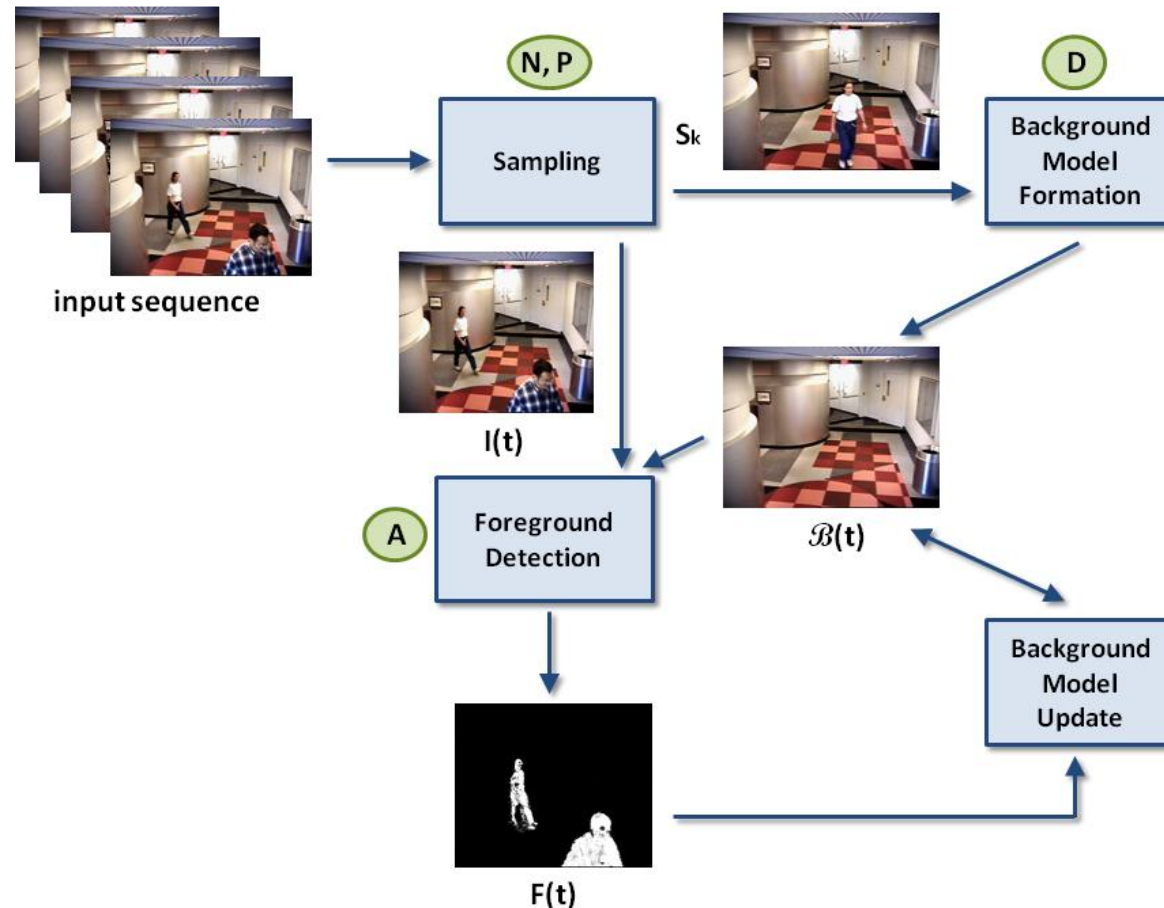


# Esempio background subtraction



# Esempio background subtraction

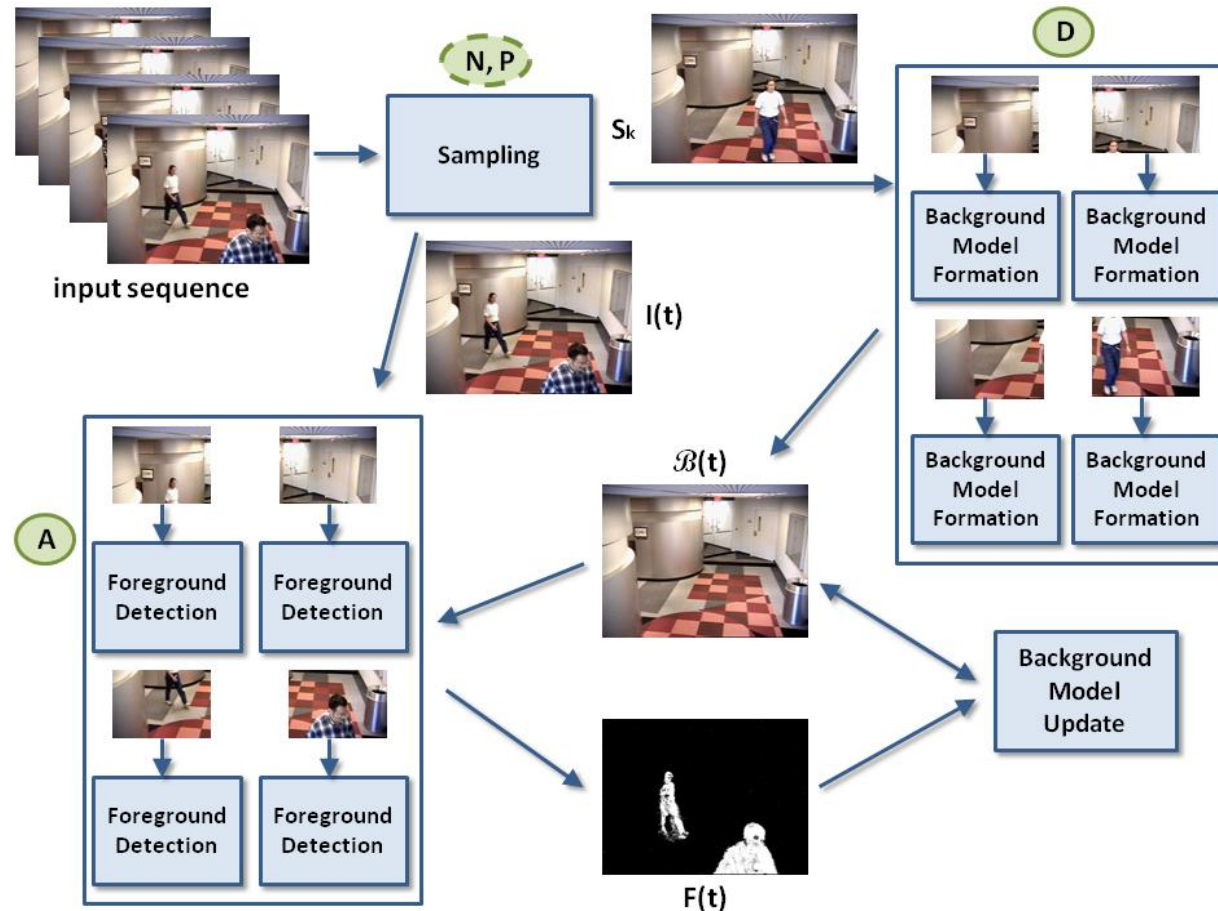
Esecuzione con **singolo processo**





# Esempio background subtraction

Esecuzione **multicore**



# Esempio background subtraction

---

**Table 3. Computational load in terms of FPS on different computer display standards for KNN (with TBB), MOG2 (with TBB), IMBS (mono-thread), and IMBS-MT (multi-thread).**

<b>Video Standard</b>	<b>Frame Size</b>	<b>KNN</b>	<b>MOG2</b>	<b>IMBS</b>	<b>IMBS-MT</b>
Video CD	352×240	144.702	80.0249	35.13	150.32
HD	1360×768	18.5798	6.2748	12.49	25.31
HD+	1600×900	13.7526	3.8475	8.42	20.72
Full HD	1920×1080	9.7816	2.9693	3.45	13.52

# Gestione dei thread

---

**thread a livello utente**



gestiti sopra il livello del kernel e senza il suo supporto

**thread a livello kernel**



gestiti direttamente dal sistema operativo

# Modelli di supporto al mutithreading

---

Modello da  
molti a uno

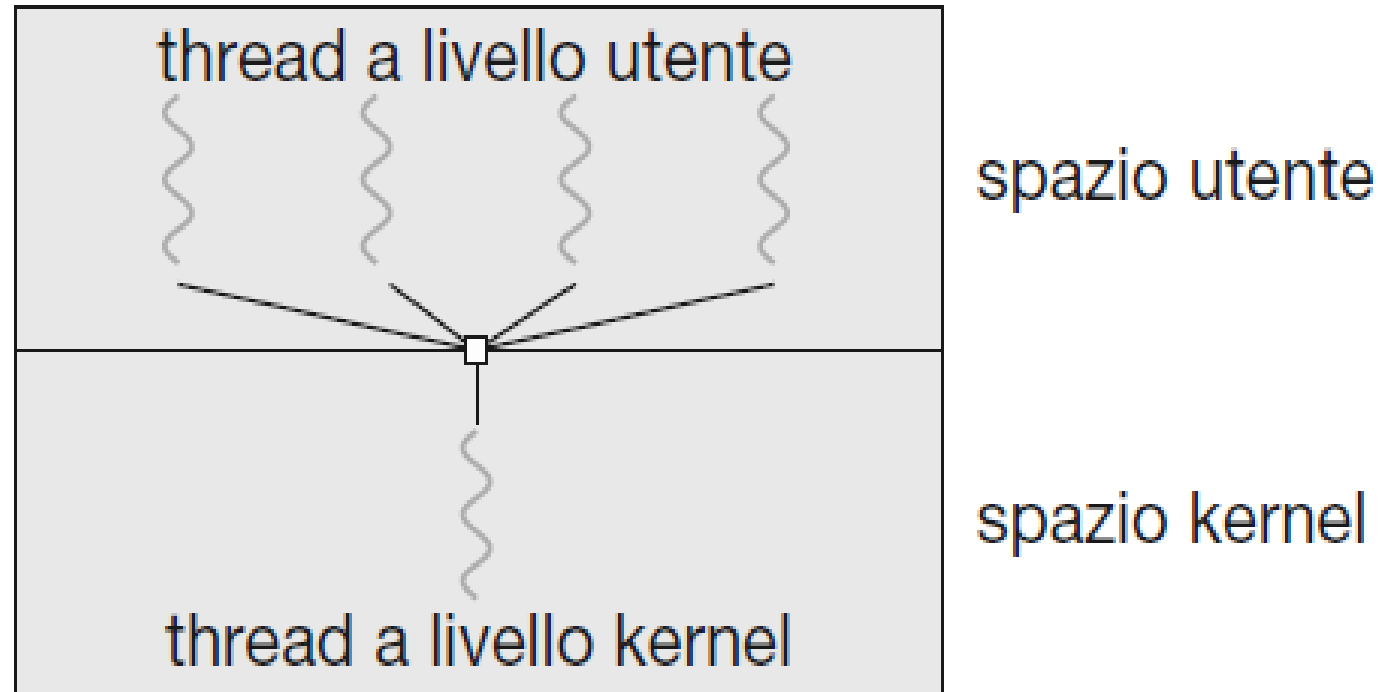
Modello da  
uno a uno

Modello da  
molti a molti

*Modello a  
due livelli*

# Modello da molti a uno

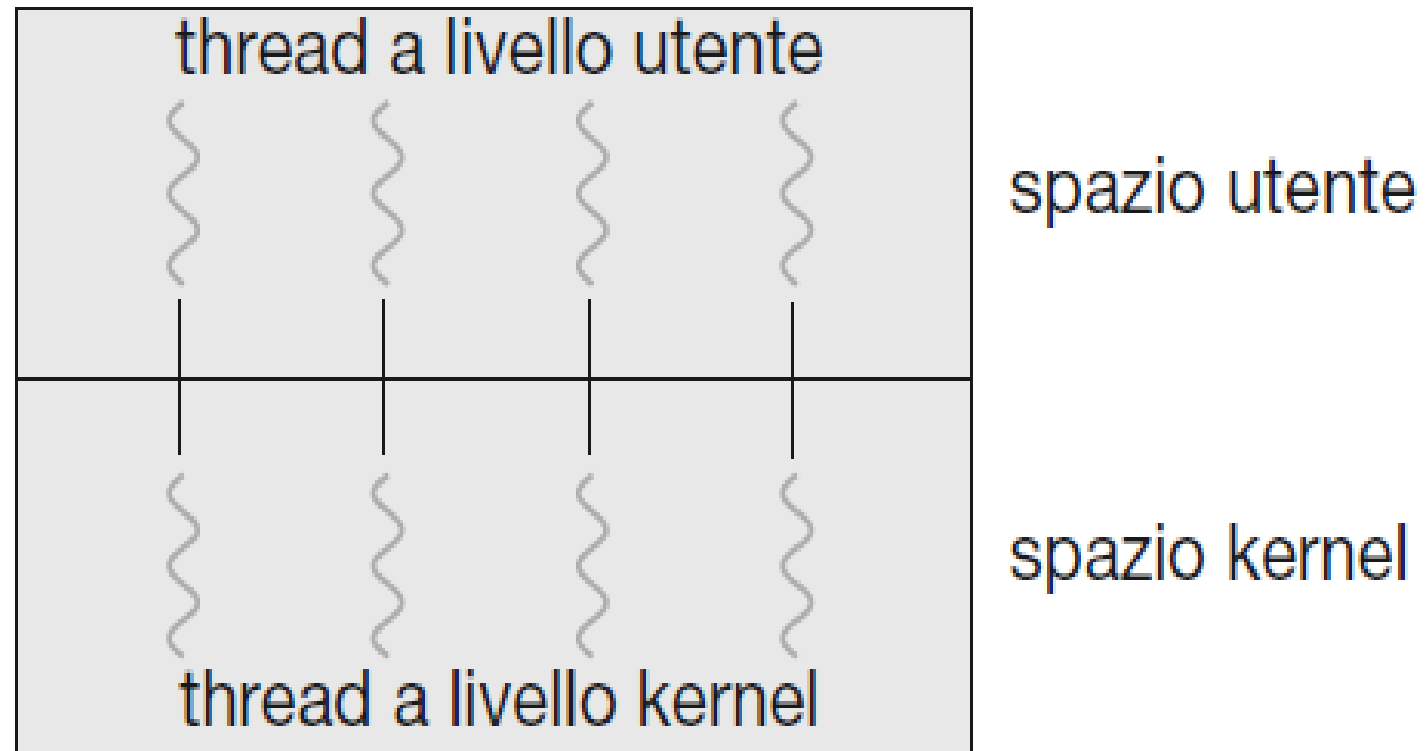
---



**Figura 4.7** Modello da molti a uno.

# Modello da uno a uno

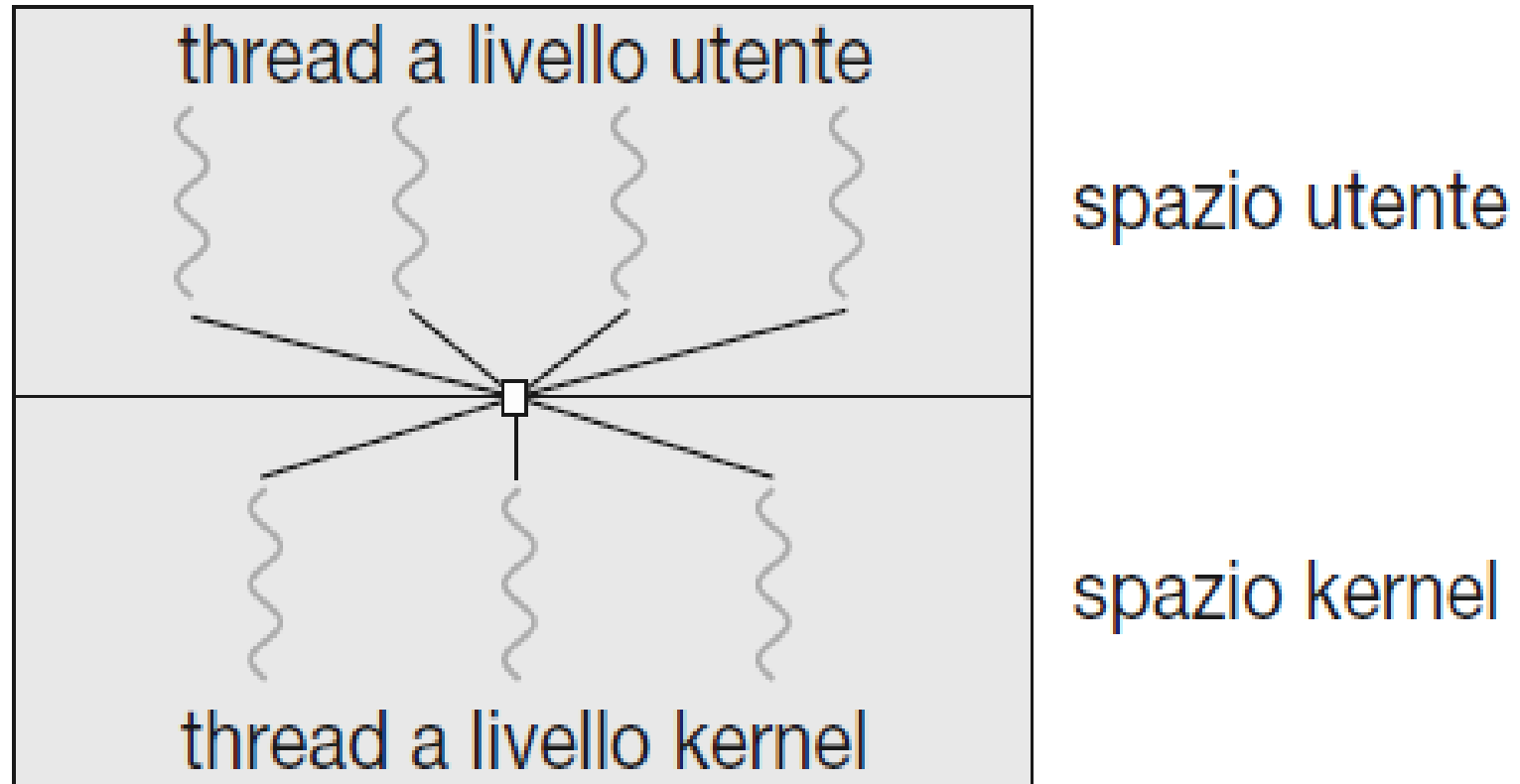
---



**Figura 4.8** Modello da uno a uno.

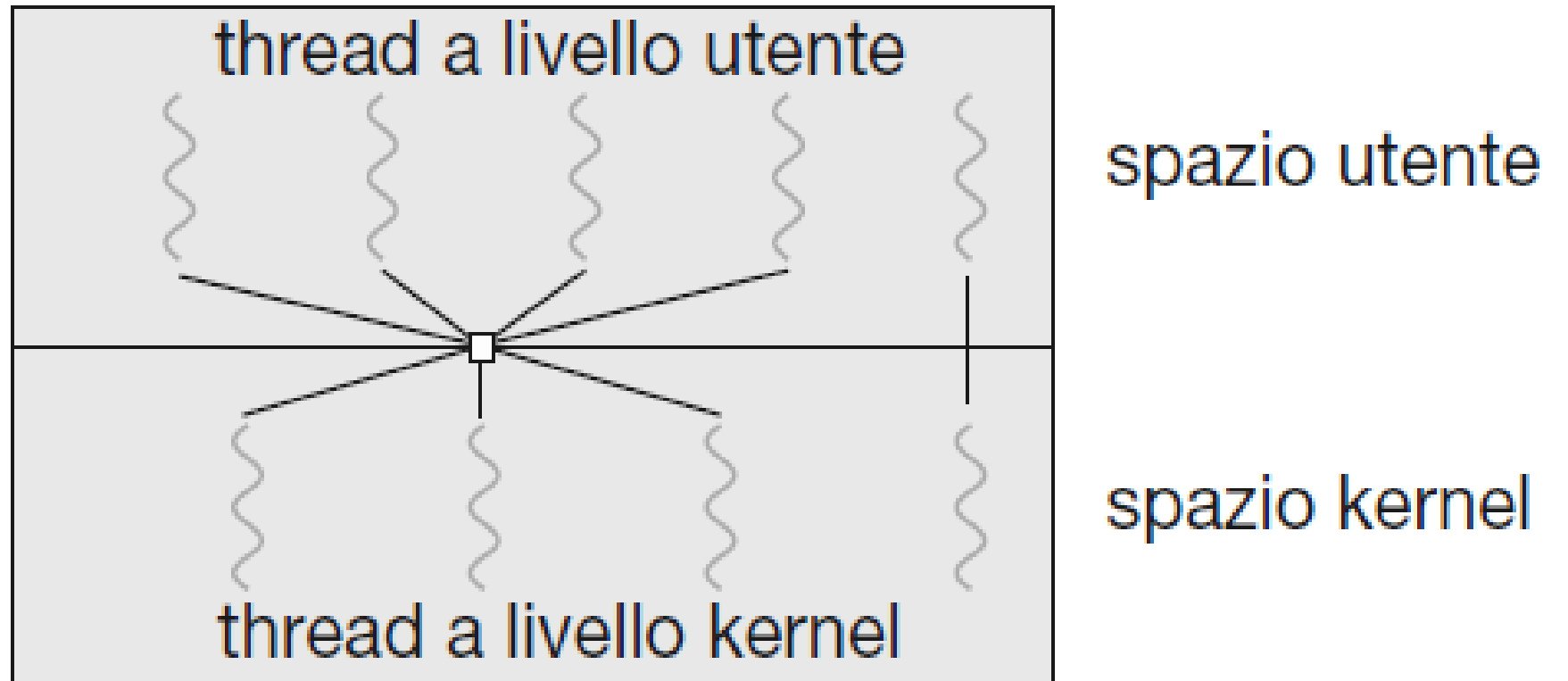
# Modello da molti a molti

---



**Figura 4.9** Modello da molti a molti.

# Modello a due livelli



**Figura 4.10** Modello a due livelli.



# Librerie dei thread

---

Una **libreria dei thread** fornisce al programmatore una API per la creazione e la gestione dei thread.

Esempi:

- **Pthreads** POSIX
- Windows
- Java

# Pthreads

---

Col termine **Pthreads** ci si riferisce allo standard POSIX (IEEE 1003.1c) che definisce una API per la creazione e la sincronizzazione dei thread.

# Esempio API Pthreads

---

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

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

```
#include <stdlib.h>
```

```
int sum; /* questo dato è condiviso dai thread */
```

```
void *runner(void *param); /* i thread chiamano questa funzione */
```

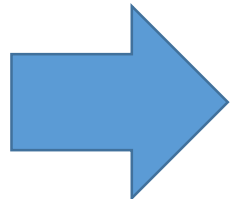
```
int main(int argc, char *argv[])
```

```
{
```

```
    pthread_t tid; /* identificatore del thread */
```

```
    pthread_attr_t attr; /* insieme di attributi del thread */
```

$$sum = \sum_{i=1}^N i$$



# Esempio API pthread

---

```
/* imposta gli attributi predefiniti del thread */
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 viene eseguito in 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 4.11** Programma multithread in linguaggio C che impiega la API Pthreads.

# Esempio API pthread

---

```
#define NUM_THREADS 10

/* array di thread da unire */
pthread_t workers[NUM_THREADS];

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

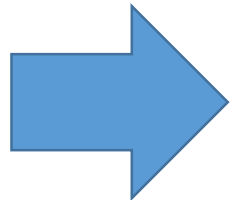
**Figura 4.12** Codice Pthread per effettuare il join di 10 thread.

# Thread in Windows

---

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* il dato è condiviso tra i thread */

/* il thread viene eseguito in questa funzione separata */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}
```



# Thread in Windows

---

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

    Param = atoi (argv [1])

    /* crea il thread */
    ThreadHandle = CreateThread(
        NULL, /* attributi di sicurezza di default */
        0, /* dimensione di default dello stack */
        Summation, /* funzione del thread */
        &Param, /* parametri alla funzione del thread */
        0, /* flag di creazione di default */
        &ThreadId); /* restituisce l'identificatore del thread */

    /* adesso aspetta la fine del thread */
    WaitForSingleObject(ThreadHandle,INFINITE);

    /* chiude l'handle del thread */
    CloseHandle(ThreadHandle);

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

**Figura 4.13** Programma multithread in C con l'utilizzo dell'API Windows.

# Thread in Java – Java Executor

---

```
import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
    private int upper;
    public Summation(int upper) {
        this.upper = upper;
    }

    /* Il thread viene eseguito in questo metodo */
    public Integer call() {
        int sum = 0;
        for (int i = 1; i <= upper; i++)
            sum += i;

        return new Integer(sum);
    }
}
```



# Thread in Java – Java Executor

---

```
public class Driver
{
    public static void main(String[] args) {
        int upper = Integer.parseInt(args[0]);

        ExecutorService pool = Executors.newSingleThreadExecutor();
        Future<Integer> result = pool.submit(new Summation(upper));

        try {
            System.out.println("sum = " + result.get());
        } catch (InterruptedException | ExecutionException ie) { }
    }
}
```

**Figura 4.14** Esempio di utilizzo dell'API Java Executor.

# Threading implicito

---

**threading implicito**




trasferimento della  
creazione e della  
gestione del threading  
*dagli sviluppatori di  
applicazioni ai  
compilatori e alle  
librerie di runtime*

# Threading implicito

---

Approcci alternativi per la progettazione di programmi multithread in grado di sfruttare i processori multicore attraverso il **threading implicito**

**gruppi di thread**  
*(thread pool)*  creare un certo numero di thread alla creazione del processo e organizzarlo in un **gruppo** (*pool*) in cui attenda di eseguire il lavoro che gli sarà richiesto

# Thread pool in Java

---

```
import java.util.concurrent.*;

public class ThreadPoolExample
{
    public static void main(String[] args) {
        int numTasks = Integer.parseInt(args[0].trim());

        /* Crea il gruppo di thread */
        ExecutorService pool = Executors.newCachedThreadPool();

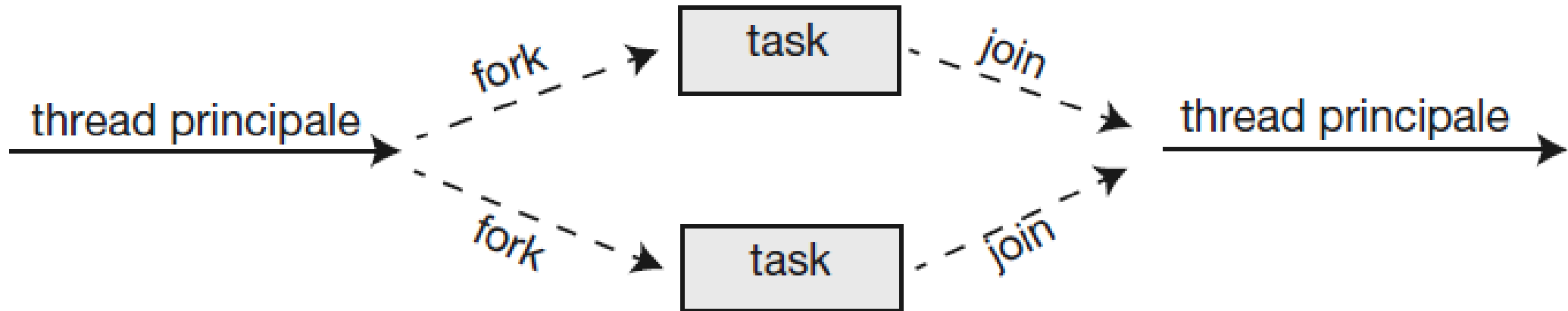
        /* Esegue ogni attività con un thread distinto del gruppo */
        for (int i = 0; i < numTasks; i++)
            pool.execute(new Task());

        /* Termina il gruppo quando tutti i thread hanno completato
           l'attività */
        pool.shutdown();
    }
}
```

**Figura 4.15** Creazione di un gruppo di thread in Java.

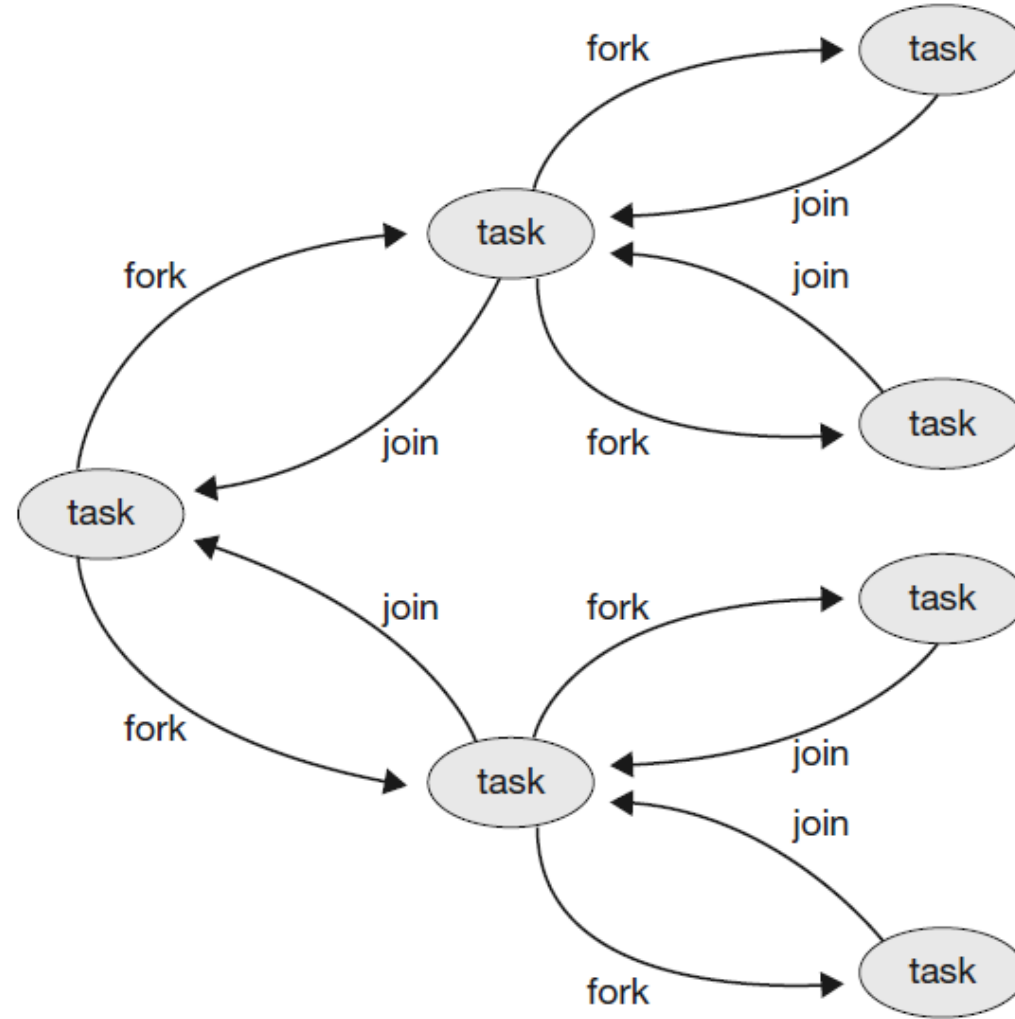
# fork – join

Nel metodo fork-join il *thread padre* crea (*fork*) uno o più *thread figli*, attende che i figli terminino e si uniscano a esso (*join*) e a quel punto può recuperare e combinare i risultati ottenuti dai figli.



**Figura 4.16** Parallelismo fork-join.

# fork – join in Java



**Figura 4.17** Fork-join in Java.

# fork – join in Java

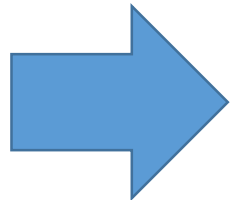
---

```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
    static final int THRESHOLD = 1000;

    private int begin;
    private int end;
    private int[] array;

    public SumTask(int begin, int end, int[] array) {
        this.begin = begin;
        this.end = end;
        this.array = array;
    }
}
```



# fork – join in Java

---

```
protected Integer compute() {
    if (end - begin < THRESHOLD) {
        int sum = 0;
        for (int i = begin; i <= end; i++)
            sum += array[i];
        return sum;
    }
    else {
        int mid = (begin + end) / 2;

        SumTask leftTask = new SumTask(begin, mid, array);
        SumTask rightTask = new SumTask(mid + 1, end, array);

        leftTask.fork();
        rightTask.fork();

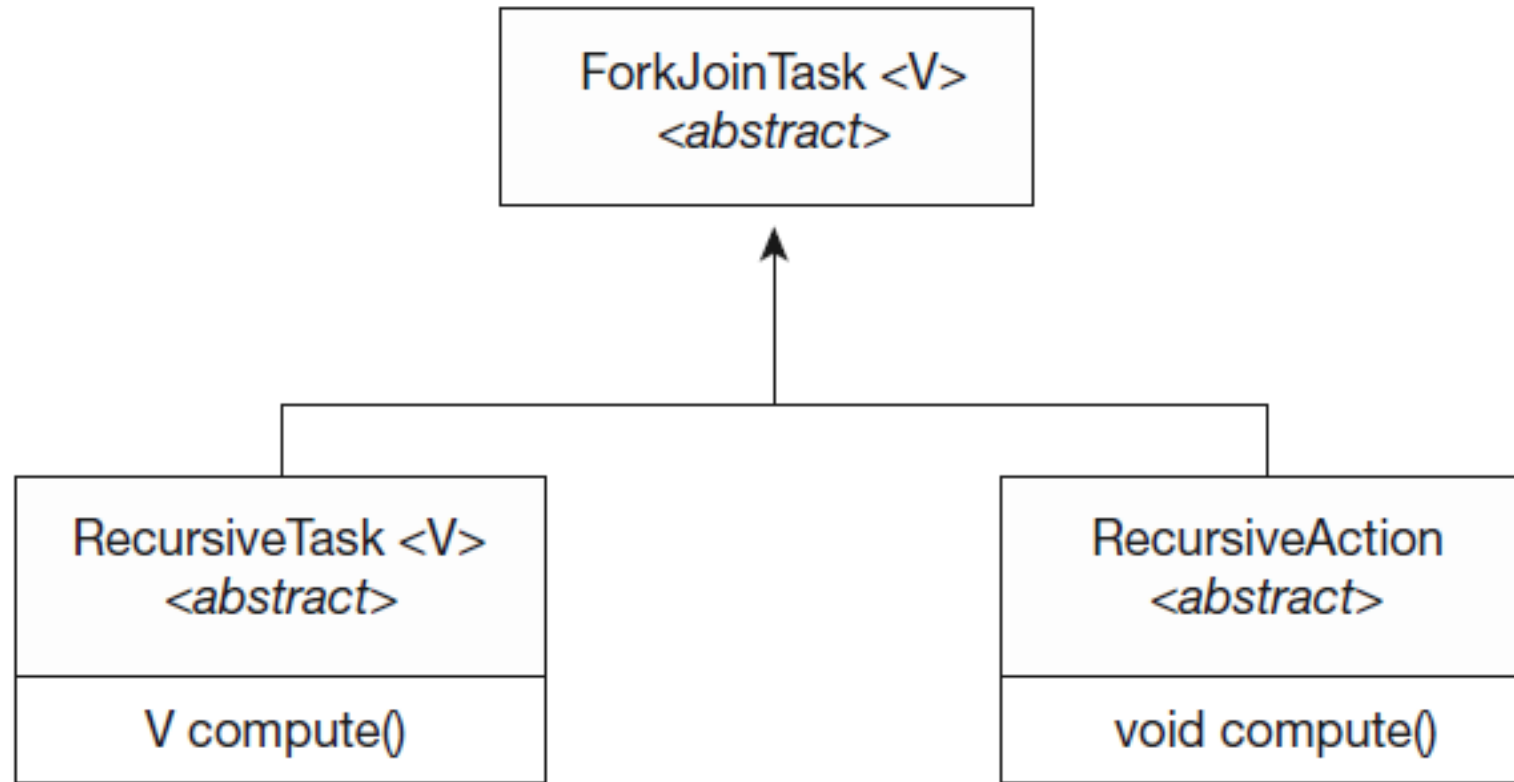
        return rightTask.join() + leftTask.join();
    }
}
```

**Figura 4.18** Esempio di computazione fork-join in Java.



# fork – join in Java

---



**Figura 4.19** Diagramma UML della classe Java per il fork-join.

# OpenMP / GCD / TBB

---

- **OpenMP** è un insieme di direttive del compilatore e una API per programmi scritti in C, C++ o FORTRAN che fornisce il supporto per la **programmazione parallela** in ambienti a memoria condivisa.
- **Grand Central Dispatch (GCD)** è una tecnologia per i sistemi operativi macOS e iOS di Apple. È una combinazione di estensioni del linguaggio C, una API e una libreria di runtime che permette agli sviluppatori di applicazioni di individuare sezioni di codice da eseguire **in parallelo**.
- **Intel Threading Building Blocks (TBB)** è una libreria di template che supporta la progettazione di **applicazioni parallele** in C++ e non richiede alcun compilatore o supporto linguistico speciale.

# Programmazione multithread

---

## PROBLEMATICHE

Chiamate di  
sistema fork()  
ed exec()

Gestione dei  
segnali

Cancellazione  
dei thread

Dati locali dei  
thread

Attivazione  
dello  
scheduler

# Cancellazione di thread

---

I thread possono essere terminati utilizzando la **cancellazione asincrona** o la **cancellazione differita**.

La **cancellazione asincrona** interrompe immediatamente un thread, anche se si trova a metà di un aggiornamento.

La **cancellazione differita** informa un thread che dovrebbe terminare la sua esecuzione, ma gli consente di terminare in modo ordinato.

Nella maggior parte dei casi la cancellazione differita è preferibile alla terminazione asincrona.

# Cancellazione in pthreads

---

pthreads permette di disabilitare o abilitare la **cancellazione dei thread**

Modalità	Stato	Tipo
Off	Disabilitato	-
Differita	Abilitato	Differito
Asincrona	Abilitato	Asincrono

# Windows vs. Linux

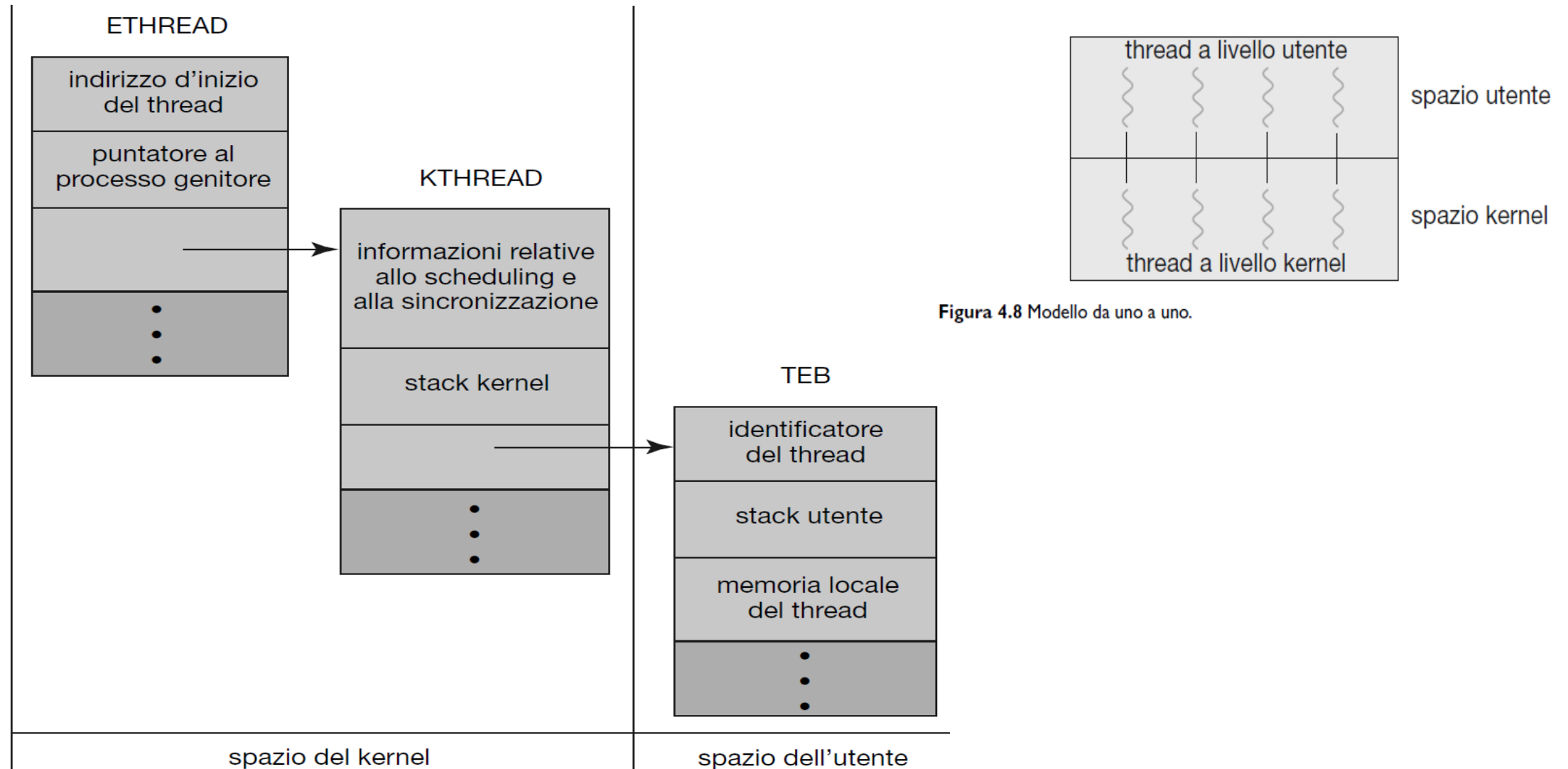


Figura 4.8 Modello da uno a uno.

**Figura 4.21** Strutture dati di un thread Windows.

# Windows vs. Linux

---

A differenza di molti altri sistemi operativi, **Linux** non distingue tra processi e thread, ma si riferisce a ciascuno come un **task**.

La chiamata di sistema **clone()** di Linux può essere utilizzata per creare task che si comportano in maniera più simile ai processi o più simile ai thread.

# clone()

---

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

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

**Figura 4.22** Alcuni dei flag passati quando viene invocata la funzione `clone()`.





**UNIVERSITÀ DEGLI STUDI  
DELLA BASILICATA**

## *Corso di Sistemi Operativi*

# Thread

Docente:  
**Domenico Daniele  
Bloisi**

