



UNIVERSITÀ
DEGLI STUDI
DEL MOLISE

Sistemi operativi e programmazione concorrente

Docente: Antonio Vito Ciliberto

A.A. 2023/2024

Informazioni sul corso

- ◆ Home page del corso: <https://learn.unimol.it/course/view.php?id=7209>
- ◆ Docente: Antonio Vito Ciliberto
- ◆ Periodo: II semestre marzo 2024 – luglio 2024
 - ◆ Martedì dalle 14:00 alle 17:00 aula Bird (secondo piano)
 - ◆ Giovedì dalle 14:00 alle 17:00 aula Bird (secondo piano)
 - ◆ Venerdì dalle 10:00 alle 13:00 aula Bird (secondo piano)

Ricevimento

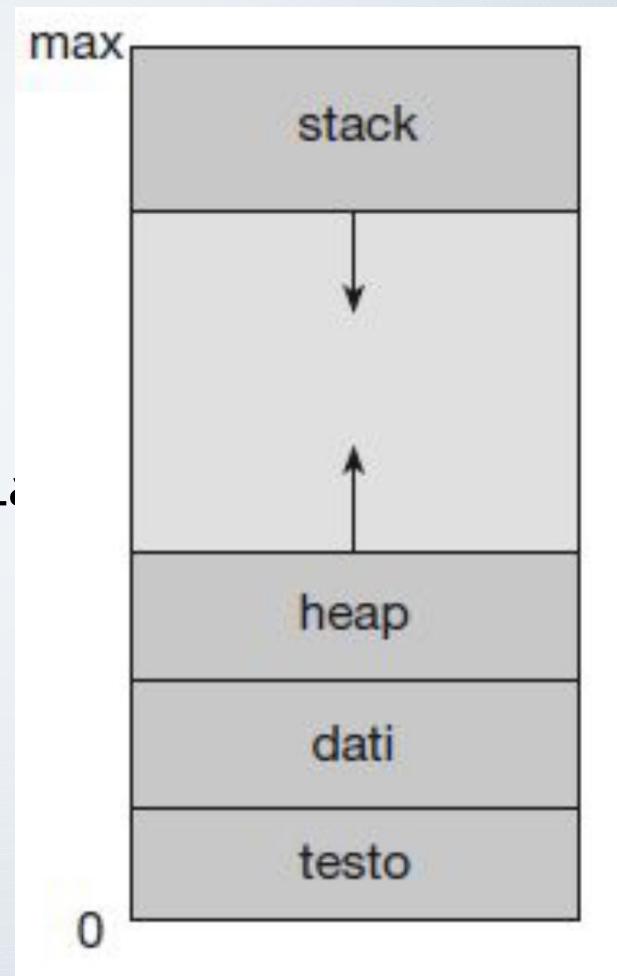
- ◆ In presenza, durante il periodo delle lezioni: da concordare con il docente tramite mail
- ◆ Tramite google meet : da concordare con il docente tramite mail
- ◆ Per prenotare un appuntamento inviare una email a
 - ◆ antonio.ciliberto@unimol.it

Concetto di processo

...dalle puntate pr

Un sistema batch (lotti) esegue job (lavori), mentre un sistema time-sharing esegue programmi utente o task; queste attività sono simili per molti aspetti, perciò sono chiamate **processi**.

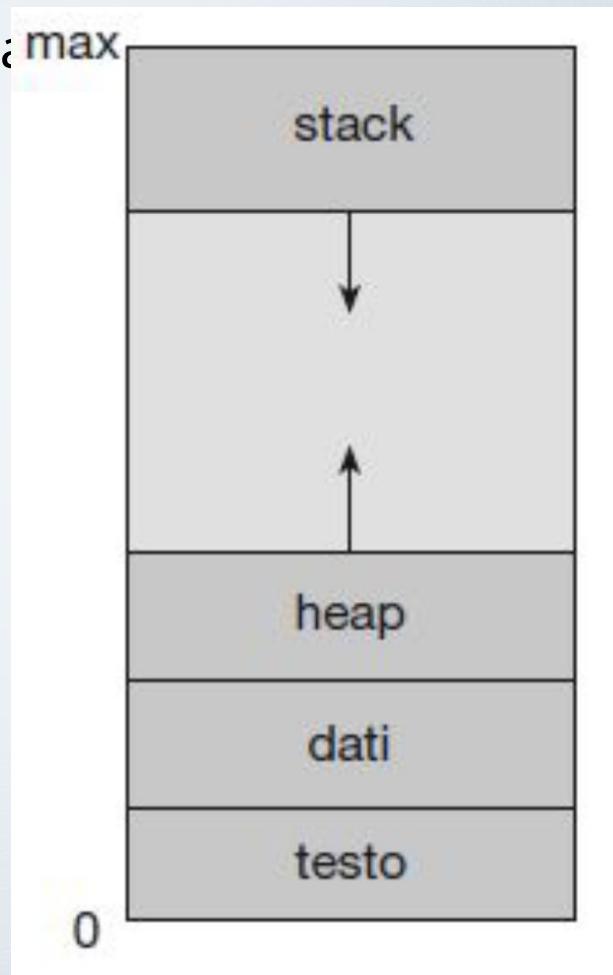
Un **processo** è un programma in esecuzione. La struttura di un processo in memoria è generalmente suddivisa in più sezioni.



Sezioni di processo

...dalle punte pr

- ◆ **Stack:** memoria temporaneamente utilizzata durante le chiamate di funzioni.
- ◆ **Heap:** memoria allocata dinamicamente durante l'esecuzione del programma
- ◆ **Dati:** contenente le variabili globali
- ◆ **Testo:** contenente il codice eseguibile



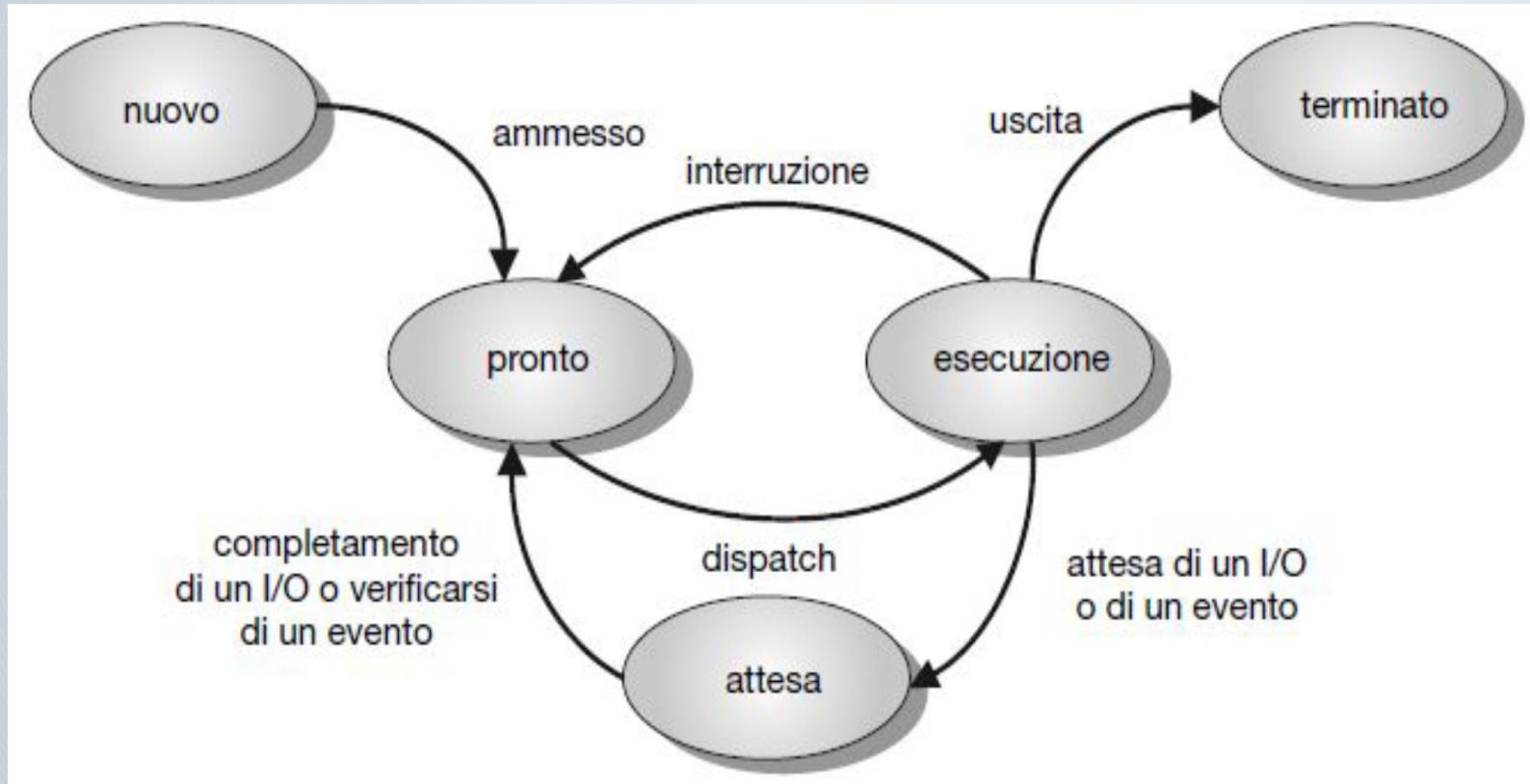
Stato di un processo

...dalle puntate pr

- ◆ **Nuovo:** Si crea il processo
- ◆ **Esecuzione (running):** Le sue istruzioni vengono eseguite
- ◆ **Attesa (waiting):** Il processo attende che si verifichi qualche evento
- ◆ **Pronto (ready):** Il processo attende di essere assegnato a un'unità di elaborazione
- ◆ **Terminato:** Il processo termina l'esecuzione

Stato di un processo

...dalle puntate pr



PCB – Process Control Block

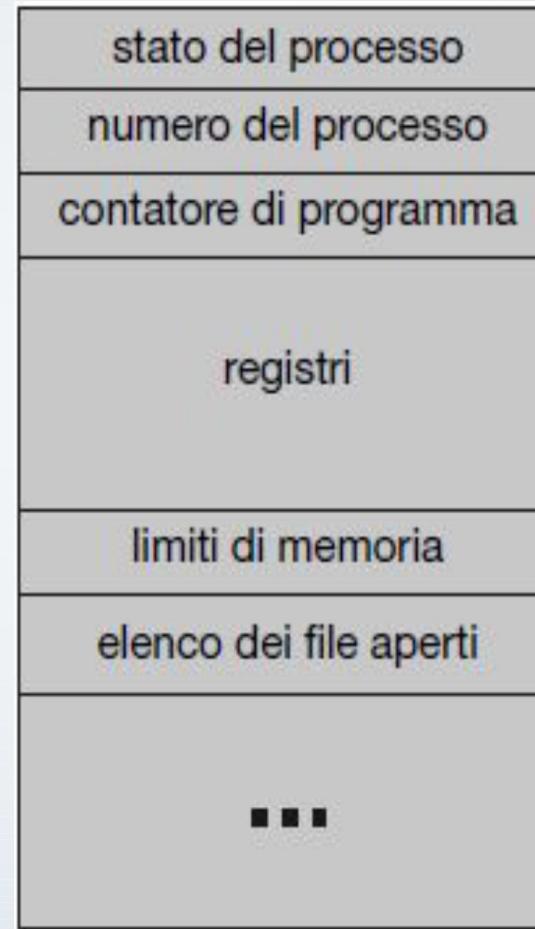
...dalle punte pr

- ◆ Ogni processo è rappresentato nel sistema operativo da un **blocco di controllo** (*process control block, PCB, o task control block, TCB*)
- ◆ Il PCB contiene un insieme di informazioni connesse a un processo specifico.

PCB – Process Control Block

...dalle puntate pr

- ◆ **Stato del processo:** nuovo, pronto, esecuzione, attesa, arresto
- ◆ **Contatore di programma:** che contiene l'indirizzo della successiva istruzione da eseguire per tale processo.
- ◆ **Registri della CPU:** accumulatori, registri indice, puntatori alla cima dello stack (*stack pointer*), registri di uso generale e registri contenenti i codici di condizione (*condition codes*)



Come avviene la comunicazione

...dalle puntate pr

Due modelli fondamentali:

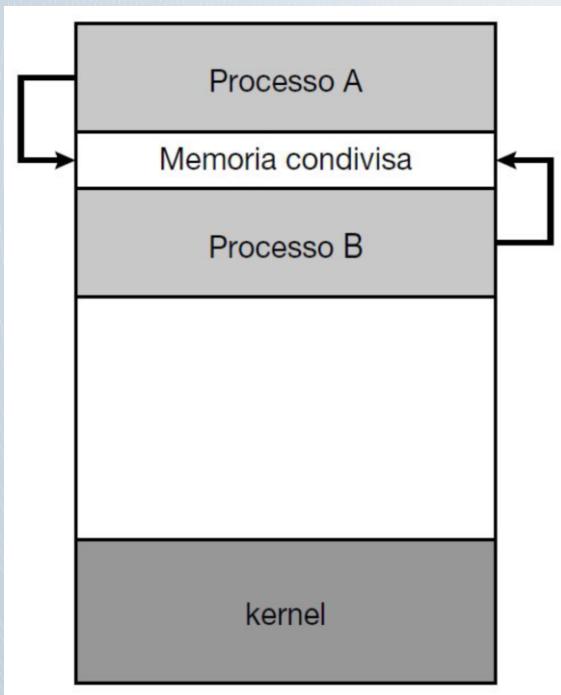
- ◆ **A memoria condivisa:** si stabilisce una zona di memoria condivisa dai processi cooperanti, che possono così comunicare scrivendo e leggendo da tale zona.
- ◆ **A scambio di messaggi:** in questo modello la comunicazione avviene per mezzo di messaggi. Utile per trasmettere piccole quantità di dati, ed è molto facile da realizzare.

Nei sistemi operativi sono diffusi entrambi i modelli, spesso coesistono in un unico sistema.

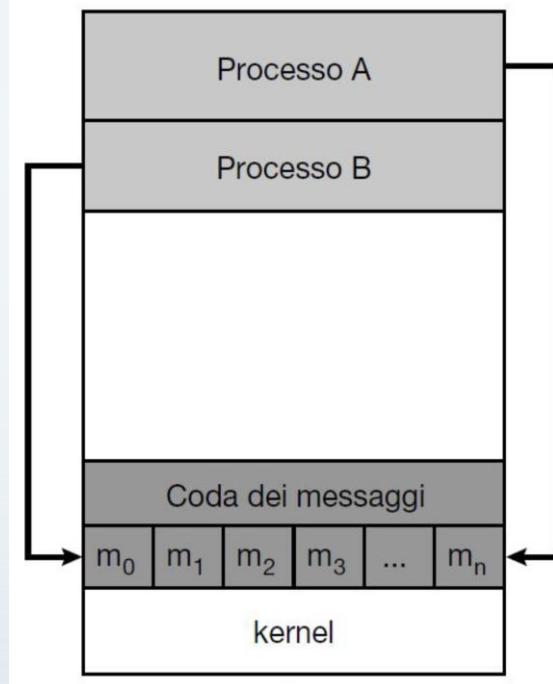
Come avviene la comunicazione

...dalle puntate pr

Memoria condivisa



Scambio di messaggi



IPC a memoria condivisa

...dalle puntate pr

Un processo chiamato **producer** produce informazioni che sono consumate da un processo chiamato **consumer**.

Es. compilatore produce codice assembly, consumato dall'assemblatore per generare oggetti che a loro volta saranno consumati dal loader.

- ◆ Problema del producer/consumer: l'esecuzione concorrente dei due processi richiede la presenza di un buffer che possa essere riempito dal producer e svuotato dal consumer.
 - ◆ Buffer illimitato
 - ◆ Buffer limitato

IPC a scambio di messaggi

...dalle puntate pr

- ◆ Lo **scambio di messaggi** è un meccanismo che permette a due o più processi di comunicare e di sincronizzarsi senza bisogno di condividere lo stesso spazio di indirizzi.
- ◆ È una tecnica particolarmente utile negli ambienti distribuiti, dove i processi possono risiedere su macchine diverse connesse da un **webchat**.
- ◆ La tipologia di canale di comunicazione scelta permette di ottenere differenti implementazioni per il modello a scambio di messaggi:
 1. Comunicazione diretta o indiretta
 2. Comunicazione sincrona o asincrona
 3. Gestione automatica o esplicita del buffer

Esempi di sistemi IPC

...dalle puntate pr

1. API POSIX
basata su memoria
condivisa

2. Scambio di
messaggi nel
sistema Mach

3. Comunicazione
fra processi
windows

4. Le pipe, canali di
comunicazione tra
processi

Producer - parte 1/2

...dalle puntate pr

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<string.h>
4 #include<fcntl.h>
5 #include<sys/shm.h>
6 #include<sys/stat.h>
7 #include<sys/mman.h>
8 #include<unistd.h>
9 #include<sys/types.h>
10
11 int main() {
12     /*dimensione in byte, dell'oggetto di memoria condivisa*/
13     const int SIZE = 4096;
14     /*nome dell'oggetto in memoria condivisa*/
15     const char *name ="OS";
16     /*stringa scritta nella memoria condivisa*/
17     const char *message_0 = "La Ferrari ";
18     const char *message_1 = "Vince il mondiale ";
19     const char *message_2 = "di Formula 1!";
20
21     /*descrittore del file di memoria condivisa*/
22     int shm_fd;
23     /*puntatore all'oggetto memoria condivisa*/
24     void *ptr;
```

Producer – parte 2/2

...dalle puntate pr

```
26     /*crea l'oggetto memoria condivisa*/
27     shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
28
29     /*configura la dimensione dell'oggetto memoria condivisa*/
30     ftruncate(shm_fd, SIZE);
31
32     /* mappa in memoria l'oggetto memoria condivisa*/
33     ptr = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
34     if(ptr == MAP_FAILED){
35         printf("Map Failed\n");
36         return -1;
37     }
38
39     /*scrivere sull'oggetto memoria condivisa*/
40     sprintf(ptr, "%s", message_0);
41     ptr += strlen(message_0);
42     sprintf(ptr, "%s", message_1);
43     ptr += strlen(message_1);
44     sprintf(ptr, "%s", message_2);
45     ptr += strlen(message_2);
46
47
48     return 0;
49
50 }
```

Consumer – parte 1/2

...dalle puntate pr

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<fcntl.h>
4 #include<sys/shm.h>
5 #include<sys/stat.h>
6 #include<sys/mman.h>
7 #include<unistd.h>
8
9 int main(){
10     /*dimensione in byte, dell'oggetto di memoria condivisa*/
11     const int SIZE = 4096;
12     /*nome dell'oggetto in memoria condivisa*/
13     const char *name ="OS";
14     /*descrittore del file di memoria condivisa*/
15     int shm_fd;
16     /*puntatore all'oggetto memoria condivisa*/
17     void *ptr;
18     int i;
19
20     /*apre l'oggetto memoria condivisa*/
21     shm_fd = shm_open(name, O_RDONLY, 0666);
22     if (shm_fd == -1) {
23         printf("shared memory failed\n");
24         exit(-1);
25     }
```

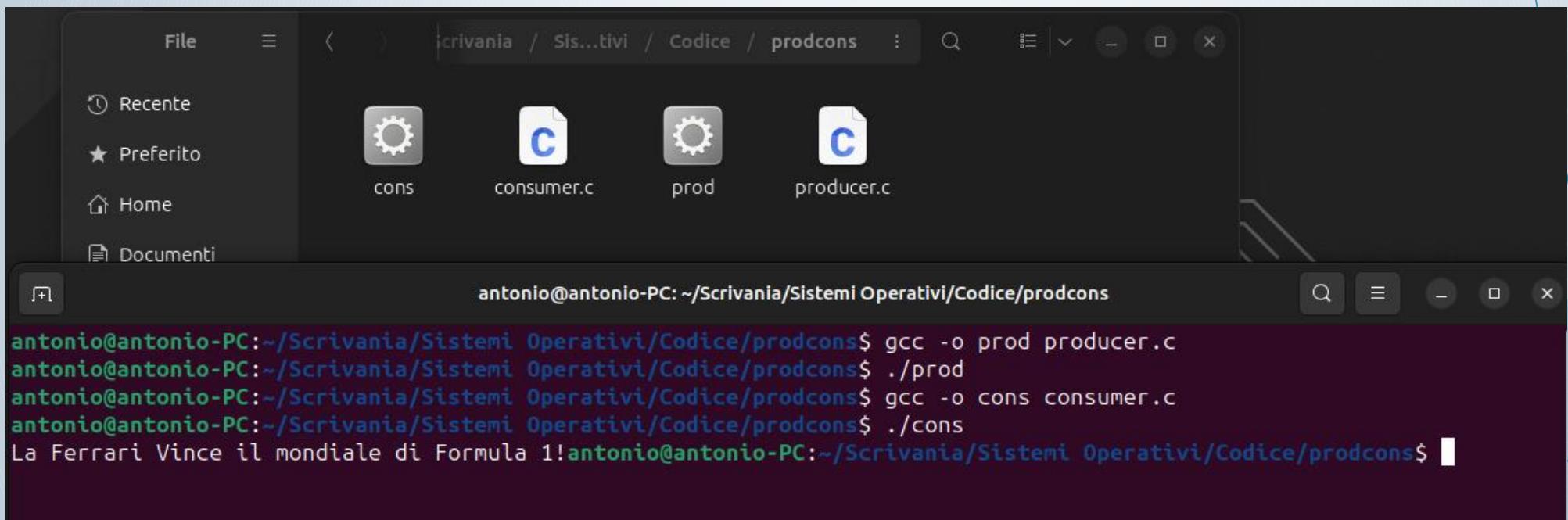
Consumer – parte 2/2

...dalle puntate pr

```
26  
27     /* mappa in memoria l'oggetto memoria condivisa*/  
28     ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);  
29     if (ptr == MAP_FAILED) {  
30         printf("Map failed\n");  
31         exit(-1);  
32     }  
33  
34     /*legge dall'oggetto memoria condivisa*/  
35     printf("%s", (char *) ptr);  
36  
37     /*rimuove l'oggetto memoria condivisa*/  
38     if (shm_unlink(name) == -1) {  
39         printf("Error removing %s\n", name);  
40         exit(-1);  
41     }  
42  
43     return 0;  
44  
45 }
```

Output esecuzione

...dalle puntate pr



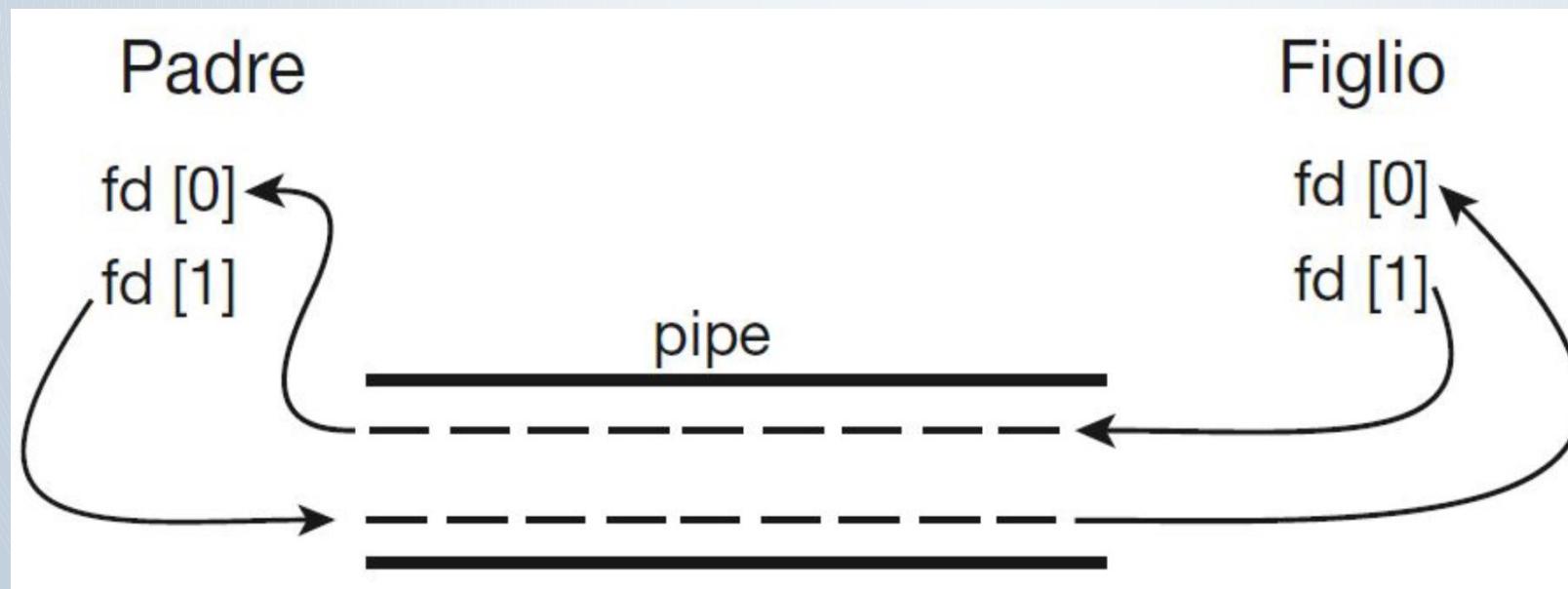
The screenshot shows a dark-themed file manager window and a terminal window. The file manager has a sidebar with 'Recente', 'Preferito', 'Home', and 'Documenti'. The main area shows four files: 'cons' (gear icon), 'consumer.c' (file icon with a 'C'), 'prod' (gear icon), and 'producer.c' (file icon with a 'C'). The terminal window below has a dark background and shows the following session:

```
antonio@antonio-PC:~/Scrivania/Sistemi Operativi/Codice/prodcons$ gcc -o prod producer.c
antonio@antonio-PC:~/Scrivania/Sistemi Operativi/Codice/prodcons$ ./prod
antonio@antonio-PC:~/Scrivania/Sistemi Operativi/Codice/prodcons$ gcc -o cons consumer.c
antonio@antonio-PC:~/Scrivania/Sistemi Operativi/Codice/prodcons$ ./cons
La Ferrari Vince il mondiale di Formula 1!antonio@antonio-PC:~/Scrivania/Sistemi Operativi/Codice/prodcons$
```

Pipe convenzionali

...dalle punte pr

Le **pipe convenzionali** permettono a due processi di comunicare secondo una modalità standard chiamata del **produttore-consumatore**. Il produttore scrive a una estremità del canale «**l'estremità dedicata alla scrittura o write-end**» mentre il consumatore legge dalla'altra estremità «**l'estremità dedicata alla lettura o read-end**». Ne consegue che le pipe convenzionali sono unidirezionali e che per abilitare una comunicazione bidirezionale è necessario implementarne due.



Pipe convenzionali – 1/2

...dalle puntate pr

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4 #include <string.h>
5
6 #define BUFFER_SIZE 50
7 #define READ_END    0
8 #define WRITE_END   1
9
10 int main(void)
11 {
12     char write_msg[BUFFER_SIZE] = "Questo è il msg letto dalla coda";
13     char read_msg[BUFFER_SIZE];
14     pid_t pid;
15     int fd[2];
16
17     /* create the pipe */
18     if (pipe(fd) == -1) {
19         fprintf(stderr,"Pipe failed");
20         return 1;
21     }
22
23     /* now fork a child process */
24     pid = fork();
25
26     if (pid < 0) {
27         fprintf(stderr, "Fork failed");
28         return 1;
29     }
```

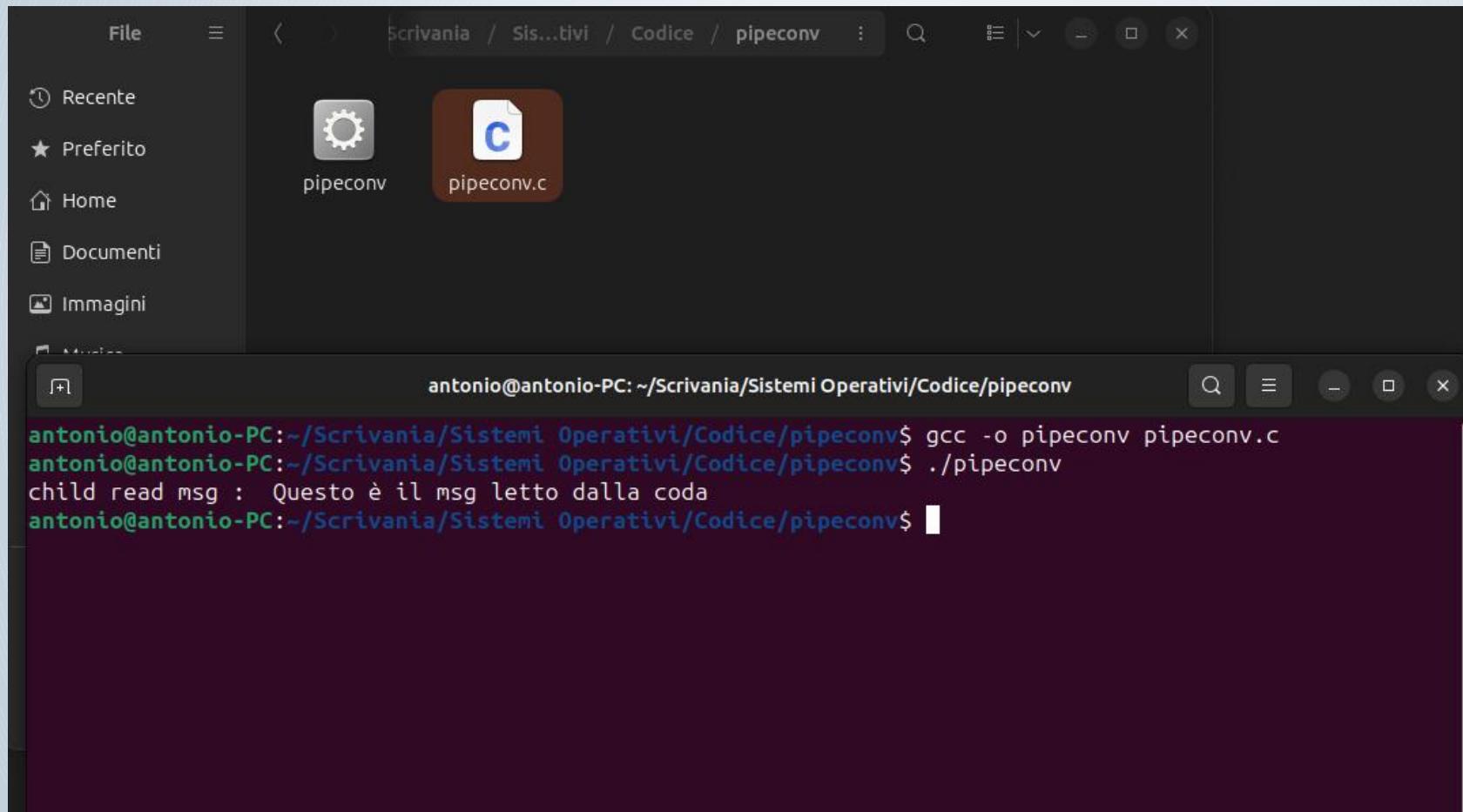
Pipe convenzionali – 2/2

...dalle puntate pr

```
30
31     if (pid > 0) { /* parent process */
32         /* close the unused end of the pipe */
33         close(fd[READ_END]);
34
35         /* write to the pipe */
36         write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
37
38         /* close the write end of the pipe */
39         close(fd[WRITE_END]);
40     }
41     else { /* child process */
42         /* close the unused end of the pipe */
43         close(fd[WRITE_END]);
44
45         /* read from the pipe */
46         read(fd[READ_END], read_msg, BUFFER_SIZE);
47         printf("child read %s\n", read_msg);
48
49         /* close the write end of the pipe */
50         close(fd[READ_END]);
51     }
52
53     return 0;
54 }
```

Output esecuzione pipeconv.c

...dalle puntate pr



The screenshot shows a Linux desktop environment with a dark theme. A file manager window is open, showing a directory structure under 'Scrivania / Sistemi Operativi / Codice / pipeconv'. Inside this directory are two files: 'pipeconv' (represented by a gear icon) and 'pipeconv.c' (represented by a C language icon). The terminal window below has a dark background and displays the following command-line session:

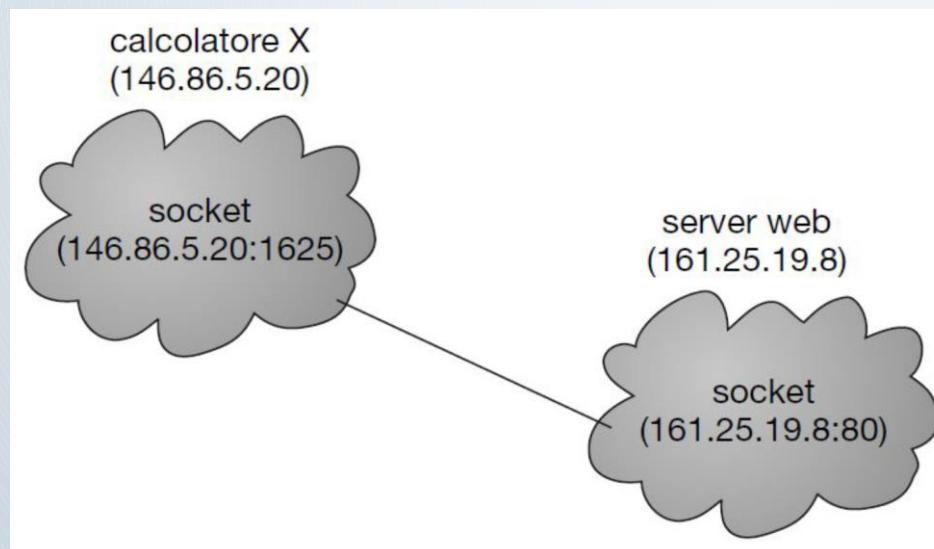
```
antonio@antonio-PC:~/Scrivania/Sistemi Operativi/Codice/pipeconv$ gcc -o pipeconv pipeconv.c
antonio@antonio-PC:~/Scrivania/Sistemi Operativi/Codice/pipeconv$ ./pipeconv
child read msg : Questo è il msg letto dalla coda
antonio@antonio-PC:~/Scrivania/Sistemi Operativi/Codice/pipeconv$
```

Socket

...dalle puntate pr

Una **socket** è definita come l'estremità di un canale di comunicazione. Impiegano generalmente un'architettura **client-server**.

- ◆ Una coppia di processi che comunicano attraverso una rete usa una coppia di socket, una per ogni processo
- ◆ Ogni socket è identificata da un indirizzo IP concatenato a un numero di porta

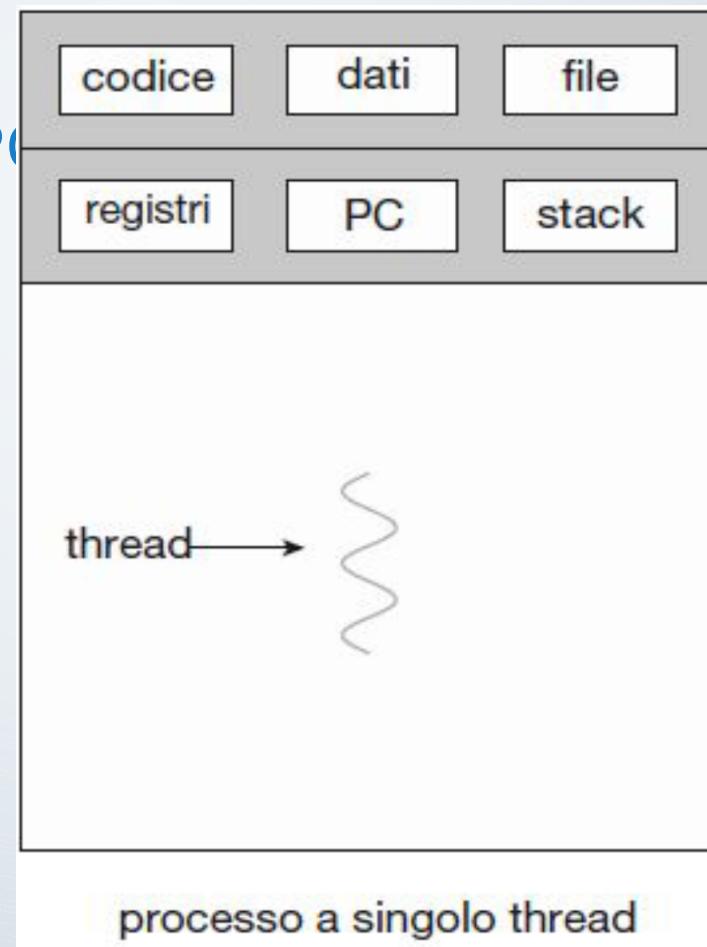


Programma del corso

- ◆ Introduzione ai sistemi operativi. Attività e struttura di un sistema operativo.
- ◆ I sistemi a processi. Comunicazione: condivisione di memoria, scambio di messaggi. Thread e concorrenza. Scheduling della CPU.
- ◆ I semafori. Cooperazione e sincronizzazione. Deadlock.
- ◆ Gestione dell'unità centrale.
- ◆ Gestione della memoria di massa.
- ◆ Il file system. Attributi, operazioni e metodi di accesso.
- ◆ Sicurezza. Protezione.

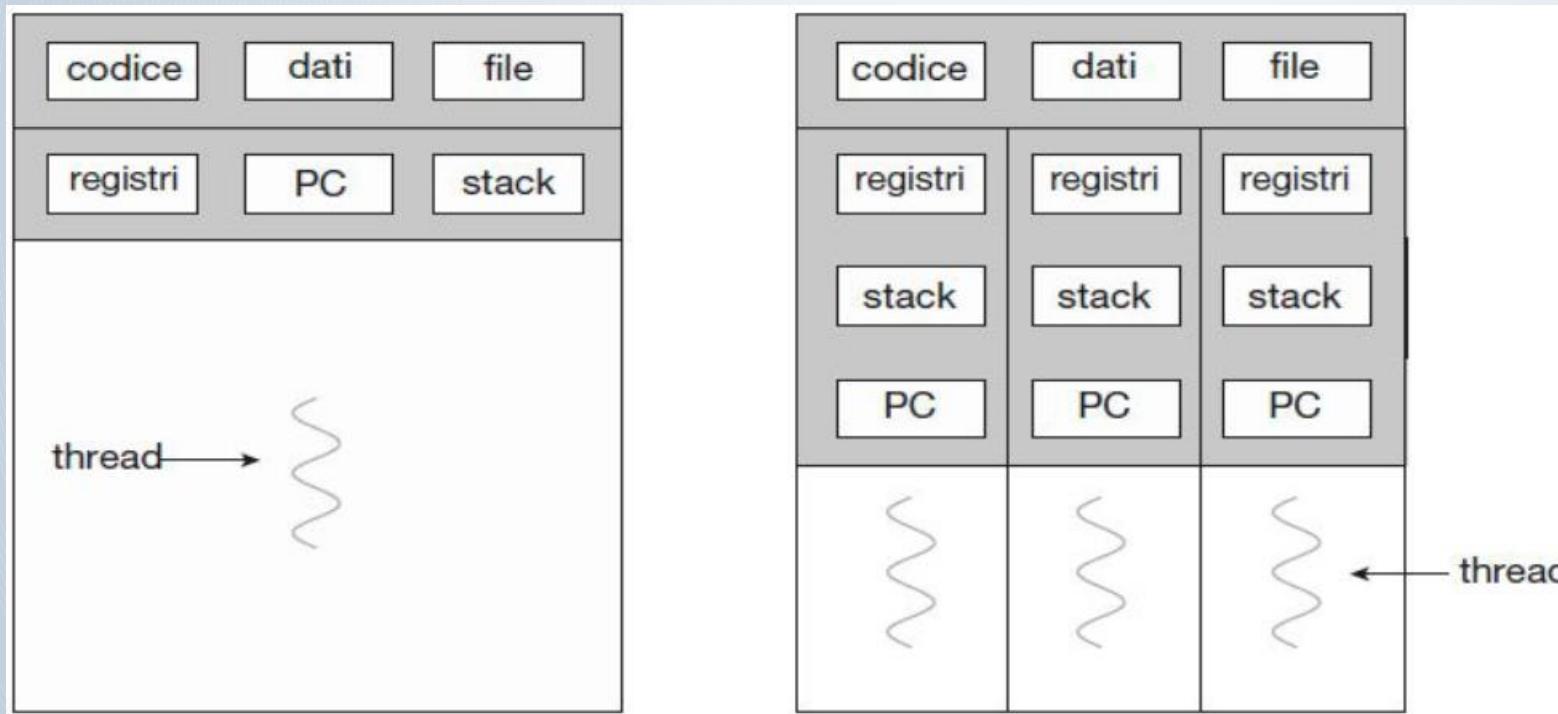
Definizione di thread

- ◆ 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)**



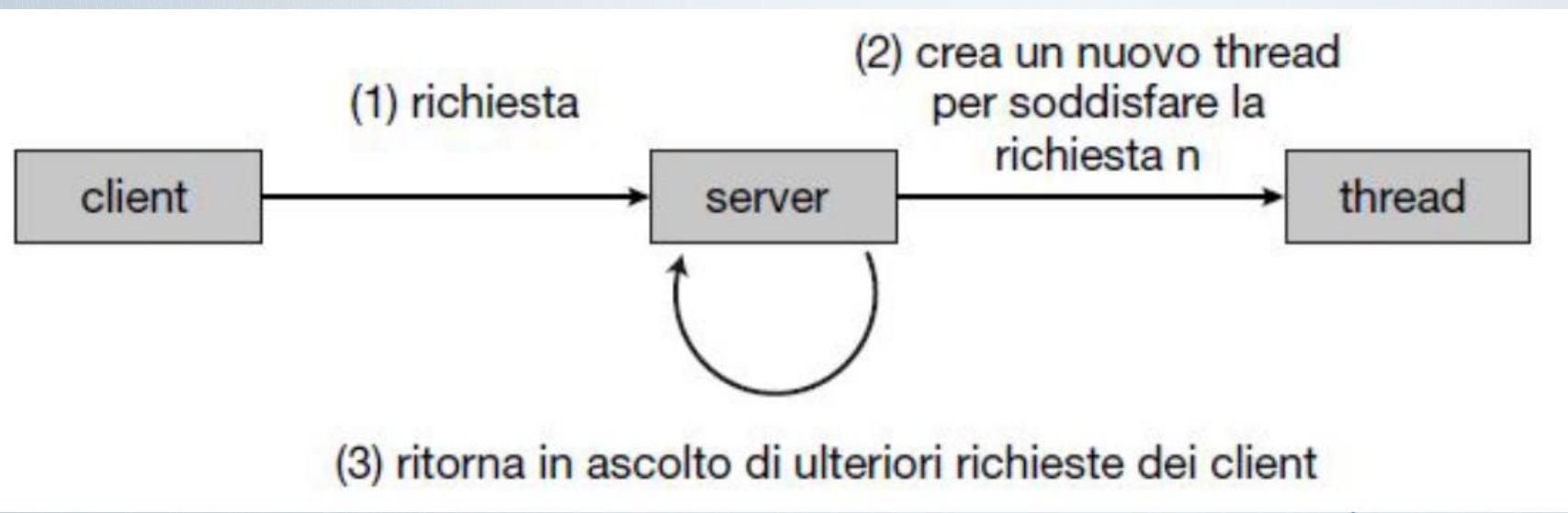
Singlethread e multithread

- ◆ La maggior parte delle applicazioni per i moderni computer è multithread.



Multithread process

- ◆ Un word processor può avere un thread per la rappresentazione grafica, uno per la risposta all'input da tastiera e uno per la correzione ortografica eseguito in background
- ◆ Un web browser può avere un thread per rappresentare sullo schermo immagini e testo e un altro thread per scaricare dati e informazioni dalla rete: (esempio in figura)



Multithread process - Vantaggi

- ◆ I vantaggi della programmazione multithread si possono classificare in 4 grandi categorie
 - **Tempo di risposta:** permettere ad un programma di continuare la sua esecuzione nonostante una parte di esso sia bloccata o stia eseguendo altre operazioni.
 - **Condivisione delle risorse:** Memoria condivisa o scambio di messaggi.
 - **Economia:** vista la capacità dei thread di condividere memoria e risorse non è necessario allocare altro spazio per i nuovi processi.
 - **Scalabilità:** nelle architetture multicore i thread possono essere eseguiti in parallelo.

Concorrenza vs Parallelismo

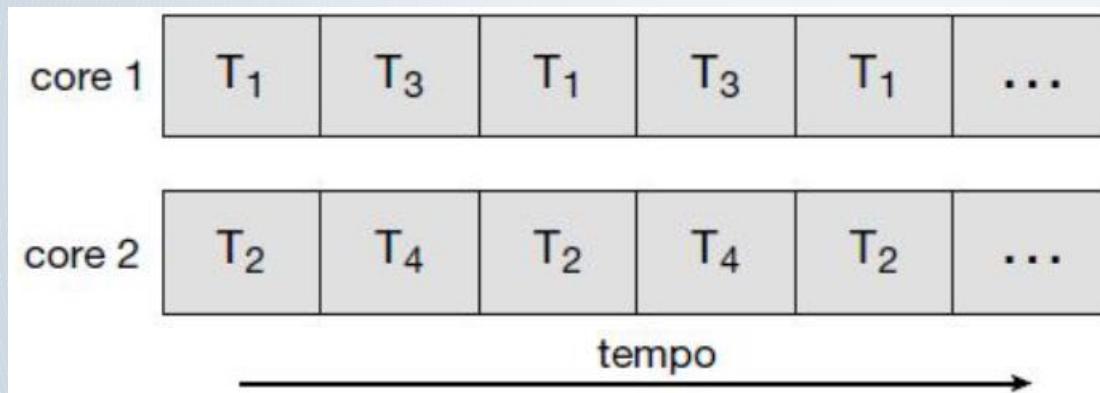
- ◆ Sistema concorrente supporta più task permettendo a ciascuno di progredire nell'esecuzione.



- ◆ N.B. Su un sistema single core Esecuzione concorrente significa solo che l'esecuzione dei thread è avvicendata nel tempo, gli algoritmi di scheduling erano progettati per dare l'illusione del parallelismo.

Concorrenza vs Parallelismo

- ◆ Sistema parallelo: può eseguire simultaneamente più task.



- ◆ N.B. su un sistema multicore per esecuzione concorrente si intende che i processi funzionano in parallelo, dal momento che il sistema può assegnare thread diversi a ciascun core.

Le sfide della programmazione

- ◆ Per gli sviluppatori di applicazioni, la sfida consiste nel modificare programmi esistenti e progettare nuovi programmi multithread.
- ◆ Legge di Amdahl: esprime una formula che permette di determinare i potenziali guadagni in termini di prestazioni ottenute dall'aggiunta di core di elaborazione, nel caso di applicazioni che consentono sia componenti seriali che componenti parallele. Si lascia l'approfondimento di questa legge al interesse personale dello studente.

Le sfide della programmazione

1. Identificazione dei task

- Esame delle app, ricerca di task separabili e concorrenti

2. Bilanciamento

- Mole di lavoro confrontabile tra task differenti

3. Suddivisione dei dati

- Separazione tra i dati alla quale i task devono accedere

4. Dipendenze dei dati

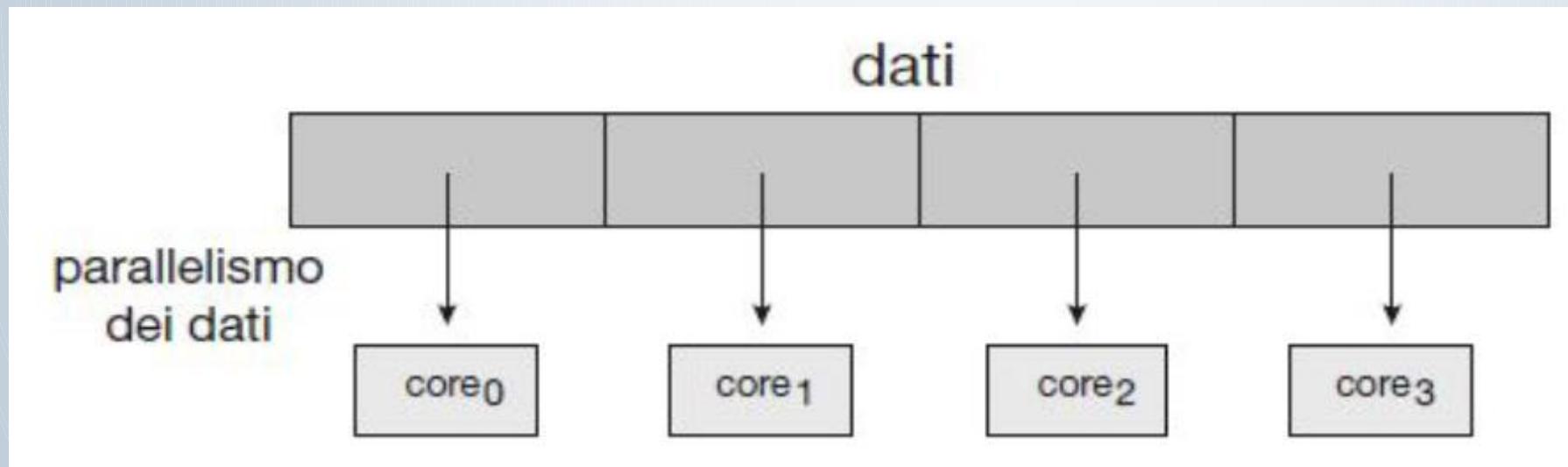
- Esaminare dati per verificare la dipendenza tra 2 o più task

5. Test e Debugging

- Diversi flussi di esecuzione test e debugging complessi

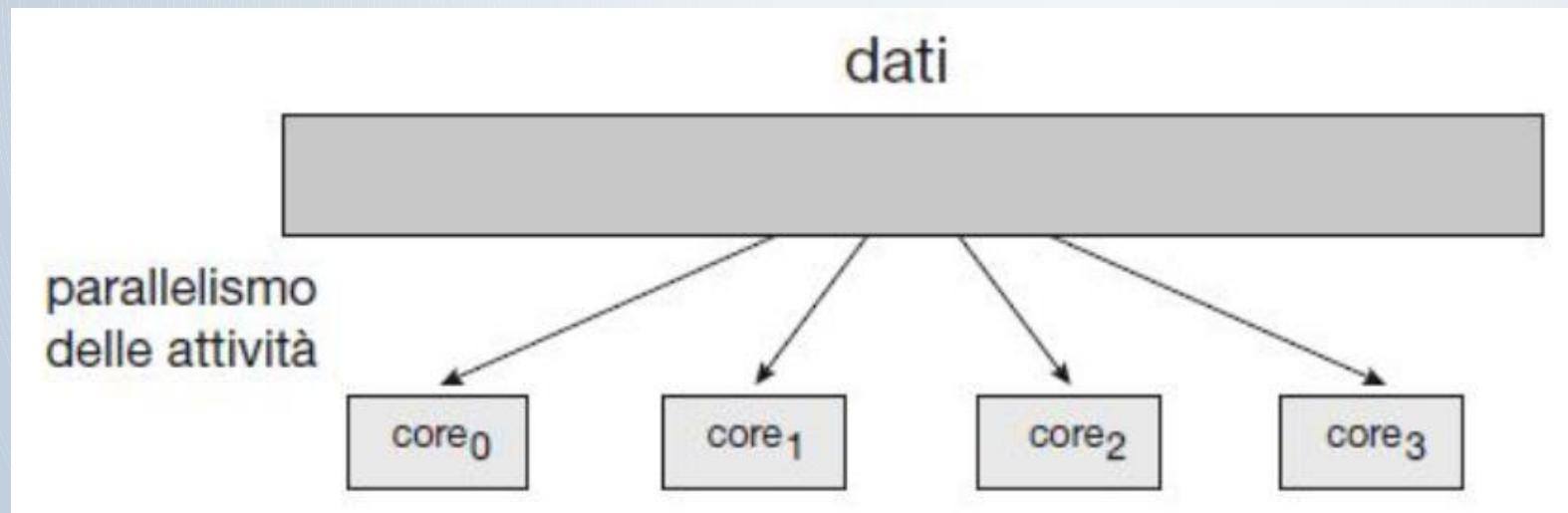
Tipi di parallelismo

- ◆ **Parallelismo dei dati:** distribuzione di sottosinsiemi dei dati su più core di elaborazione ed esecuzione della stessa operazione su ogni core:



Tipi di parallelismo

- ◆ **Parallelismo delle attività:** Distribuzione di attività(thread) su più core, ogni thread realizza un'operazione distinta e a thread differenti è concesso l'utilizzo degli stessi dati o di dati diversi:



Gestione dei processi leggeri(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

Esiste una relazione che lega le due tipologie di processo?

Gestione dei processi leggeri(thread)

Modello
molti a uno

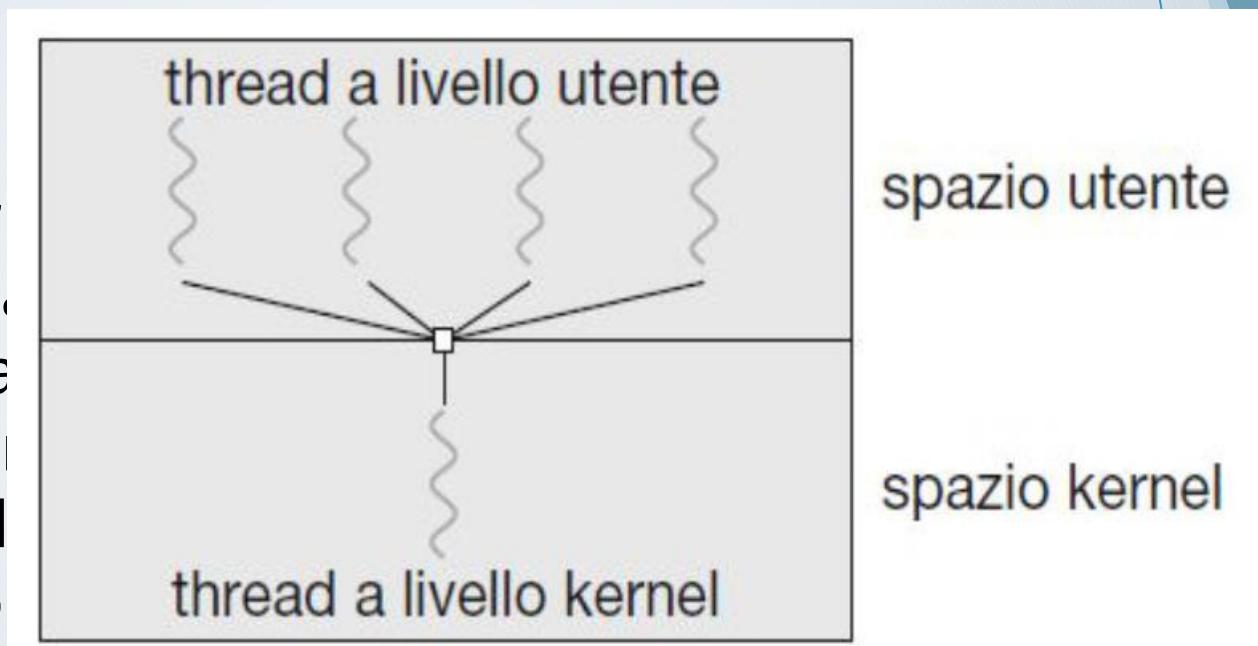
Modello uno
a uno

Modello
molti a molti

Modello a
due livelli

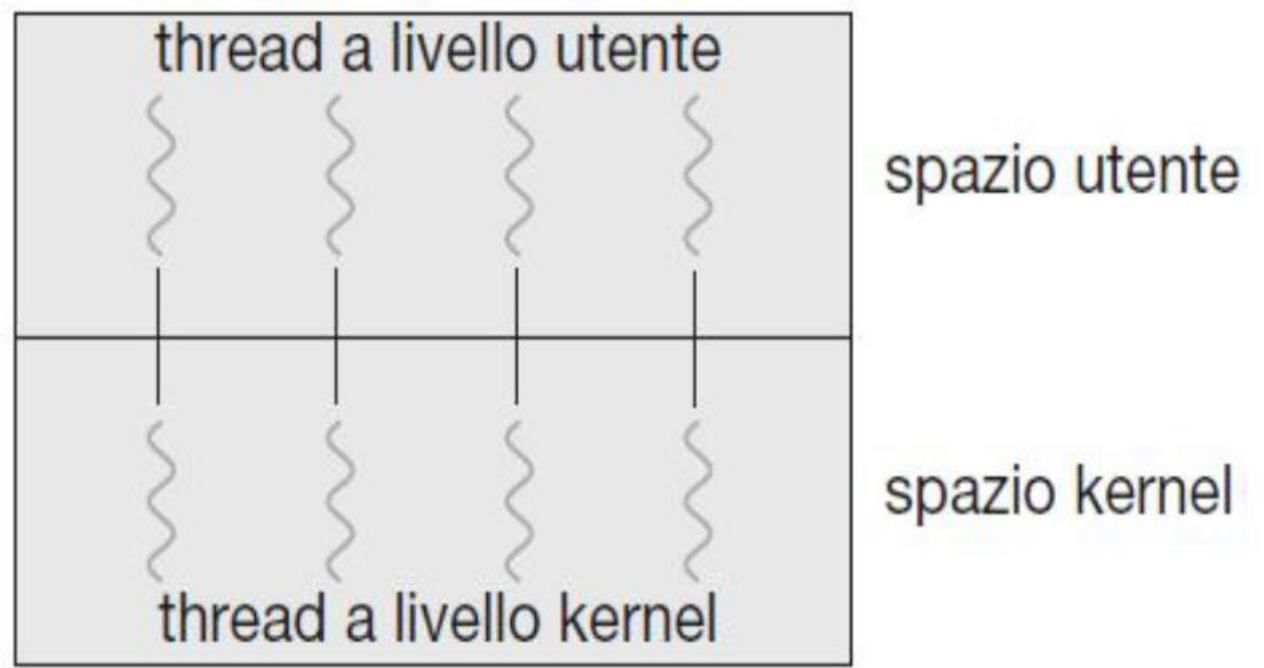
Relazioni tra thread – molti a uno

- ◆ In questo modello la gestione dei thread risulta efficiente in quanto effettuata per mezzo di una libreria a livello utente. Tuttavia se un thread invoca una chiamata di sistema di tipo bloccante l'intero processo resta bloccato.



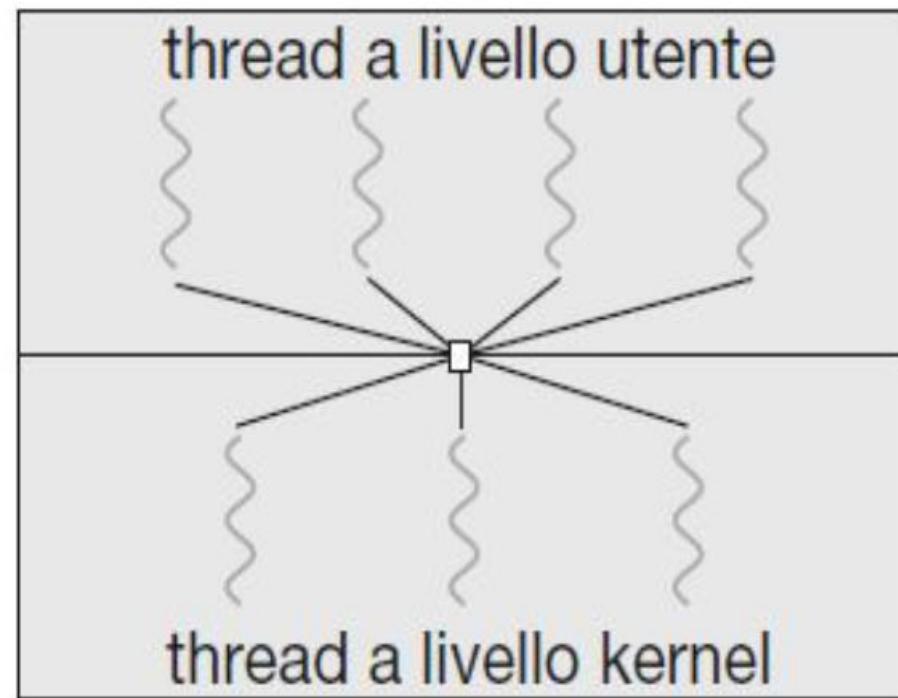
Relazioni tra thread – uno a uno

- ◆ In questo modello il grado di concorrenza è maggiore, si risolve il problema della chiamata di sistema bloccante, e nei sistemi multicore è possibile anche l'esecuzione in parallelo. Unico svantaggio la creazione di un thread kernel per ogni thread utente.



Relazioni tra thread – molti a molti

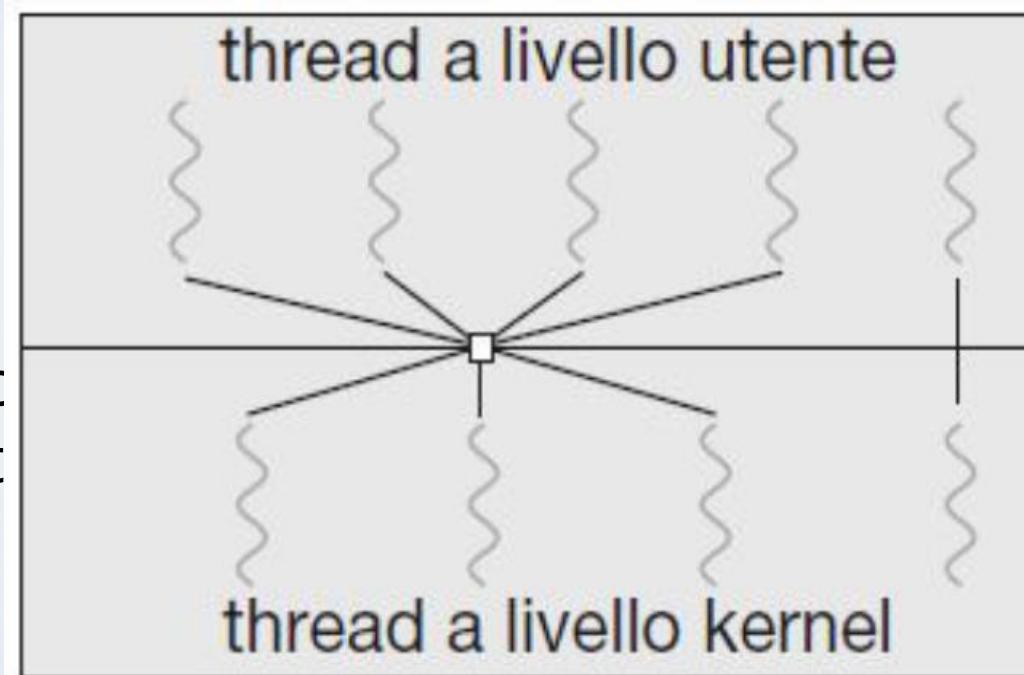
- ◆ In questo modello è permessa la creazione di più thread sia a livello utente che a livello kernel, quest'ultimo può essere specifico per una certa applicazione per un particolare calcolatore. Diversa quantità di assegnazione in base all'architettura 8 core/4 core ecc.



spazio utente
spazio kernel

Relazioni tra thread - 2 livelli

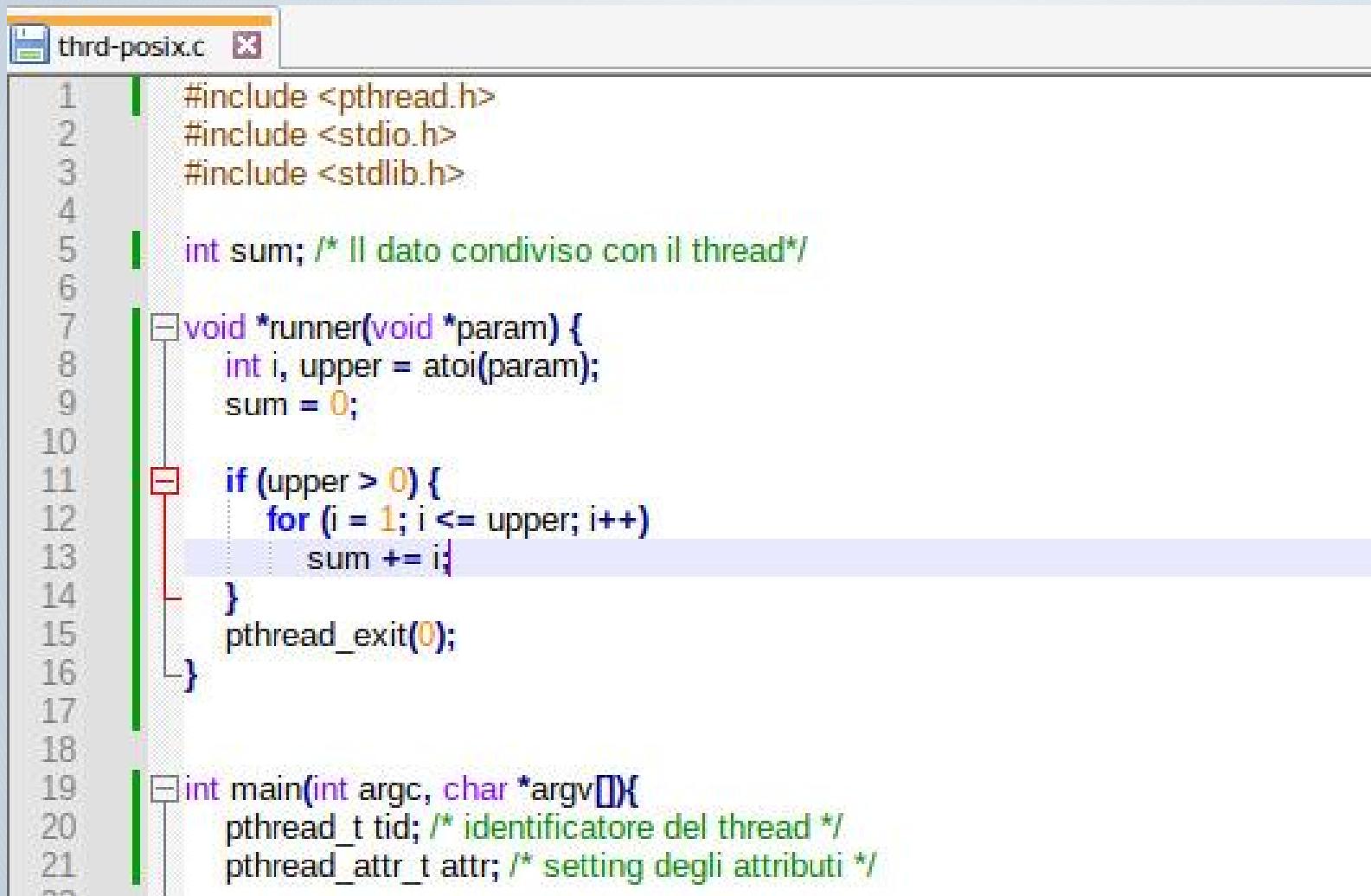
- ◆ Questa variante del modello molti a molti mantiene la corrispondenza numerica tra thread a livello utente e a livello kernel con la possibilità di vincolare un thread a livello utente con un thread a livello kernel.



API – librerie per thread

- ◆ Una libreria dei thread fornisce al programmatore una API per la creazione e la gestione dei thread. Attualmente sono tre le librerie di thread maggiormente in uso:
 - ◆ **Pthread** – estensione dello standard Posix, puo essere realizzata sia a livello utente che a livello kernel
 - ◆ **Thread Windows** – libreria a livello kernel per ambiente windows
 - ◆ **Java** – gestita direttamente dall'applicazione, la JVM gestisce questa libreria in base al SO ospitante java.

Pthread



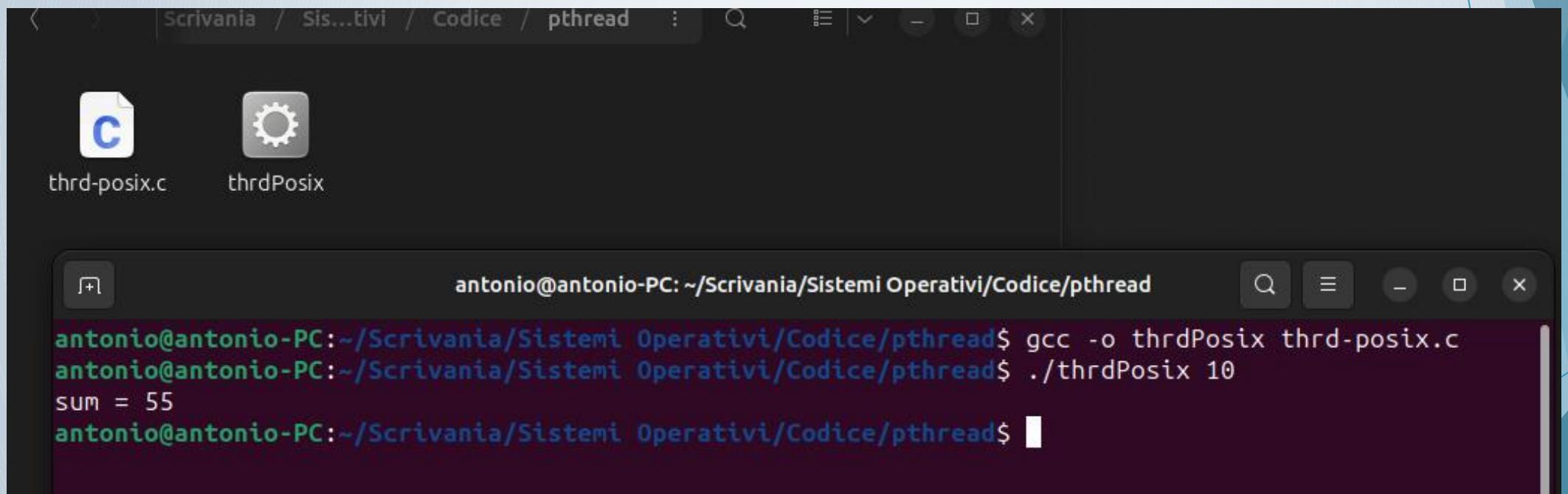
The screenshot shows a code editor window titled "third-posix.c". The code is a C program demonstrating the use of threads from the Pthreads library. It includes standard headers and defines a shared variable "sum". The "runner" function calculates the sum of integers from 1 to "upper". The "main" function creates a thread and joins it.

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int sum; /* Il dato condiviso con il thread*/
6
7 void *runner(void *param) {
8     int i, upper = atoi(param);
9     sum = 0;
10
11    if (upper > 0) {
12        for (i = 1; i <= upper; i++)
13            sum += i;
14    }
15    pthread_exit(0);
16}
17
18
19 int main(int argc, char *argv[])
20 {
21     pthread_t tid; /* identificatore del thread */
22     pthread_attr_t attr; /* setting degli attributi */
```

Pthread

```
23 |     if (argc != 2) {  
24 |         fprintf(stderr,"usage: a.out <integer value>\n");  
25 |         return -1;  
26 |     }  
27 |  
28 |     if (atoi(argv[1]) < 0) {  
29 |         fprintf(stderr,"Argument %d must be non-negative\n",atoi(argv[1]));  
30 |         return -1;  
31 |     }  
32 |  
33 |     /* attributi di default */  
34 |     pthread_attr_init(&attr);  
35 |  
36 |     /* creazione del thread */  
37 |     pthread_create(&tid,&attr,runner,argv[1]);  
38 |  
39 |     /* attesa del completamento del thread */  
40 |     pthread_join(tid,NULL);  
41 |  
42 |     printf("sum = %d\n",sum);  
43 | }
```

Pthread



The screenshot shows a terminal window with the following details:

- File Explorer:** Shows two files: "thrd-posix.c" and "thrdPosix".
- Terminal Title:** antonio@antonio-PC: ~/Scrivania/Sistemi Operativi/Codice/pthread
- Terminal Content:**

```
antonio@antonio-PC:~/Scrivania/Sistemi Operativi/Codice/pthread$ gcc -o thrdPosix thrd-posix.c
antonio@antonio-PC:~/Scrivania/Sistemi Operativi/Codice/pthread$ ./thrdPosix 10
sum = 55
antonio@antonio-PC:~/Scrivania/Sistemi Operativi/Codice/pthread$
```

Pthread – dettagli 1/2

- ◆ Tutti i programmi che lavorano con i thread devono integrare al loro interno la libreria **pthread.h**
- ◆ La dichiarazione **pthread_t tid** specifica l'identificatore per il thread da creare
- ◆ La dichiarazione **pthread_attr_t attr** riguarda la struttura dati per gli attributi del thread, i cui valori sono assegnati attraverso la chiamata **pthread_attr_init(&attr)**
- ◆ La chiamata **Pthread_create()** crea un nuovo thread

Pthread – dettagli 2/2

A questo punto il programma ha due thread ovvero:

1. Il thread iniziale (il main()) – Genitore
2. Il thread che esegue la somma (il runner()) – Figlio

Il programma a questo punto sfrutta una fork-join:

- ◆ Dopo aver creato il figlio, il genitore ne attende il completamento chiamando la funzione **`pthread_join()`**.
- ◆ Il thread figlio termina quando chiama la funzione **`pthread_exit()`**.
- ◆ Quando il figlio termina il thread genitore produce in output il valore condiviso «**sum**».

Pthread – fork_join()

- ◆ Il codice di esempio visto in precedenza genera al massimo due thread provocando l'attesa sul thread figlio.
- ◆ Per i sistemi multicore l'attesa potrebbe essere caratterizzata dalla presenza di più thread.
- ◆ Quale può essere un modo semplice di gestire l'attesa di più thread usando la funzione **pthread_join()**?

Pthread - fork_join() soluzione

```
#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);
```

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 crezione 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);
}
```

Thread in Windows

- ◆ La tecnica usata dalla libreria di Windows per la creazione dei thread non si discosta molto da quella vista in precedenza.
- ◆ Tutti i programmi devono includere la libreria **windows.h**
- ◆ Anche in questo caso come si affronta la problematica legata all'attesa di più thread?

Thread in Windows - multithread

- ◆ In situazioni che richiedono l'attesa della terminazione di più thread viene utilizzata la funzione :
 - WaitForMultipleObjects()
- ◆ alla quale vengono passati quattro parametri:
 1. Il numero di oggetti da attendere
 2. Un puntatore al vettore di oggetti
 3. Un flag che indica se tutti gli oggetti sono stati segnalati
 4. La durata del timeout

Thread in Java - Java executor

- ◆ I thread in java sono disponibili su qualunque sistema includa una JVM. In java vi sono due tecniche per la generazione dei thread:
 - ◆ Creare una nuova classe derivata dalla classe **Thread** che sovrascrive (override) il suo metodo run().
 - ◆ Definire una classe che implementa l'interfaccia **Runnable**.
- ◆ In entrambi i casi è necessario invocare il metodo start() del nuovo oggetto Thread, questo provoca un duplice effetto:
 1. Allocare memoria e inizializzare un nuovo thread nella JVM
 2. Chiamare il metodo run(), ovvero rendere il thread eseguibile all'interno della JVM.

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) { }
    }
}
```

Threading implicito

Trasferimento della creazione e della gestione del threading dagli sviluppatori di applicazioni ai compilatori e alla librerie runtime.

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

- ◆ Gruppi di thread: creare un certo numero di thread alla creazione del processo e organizzarli 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());

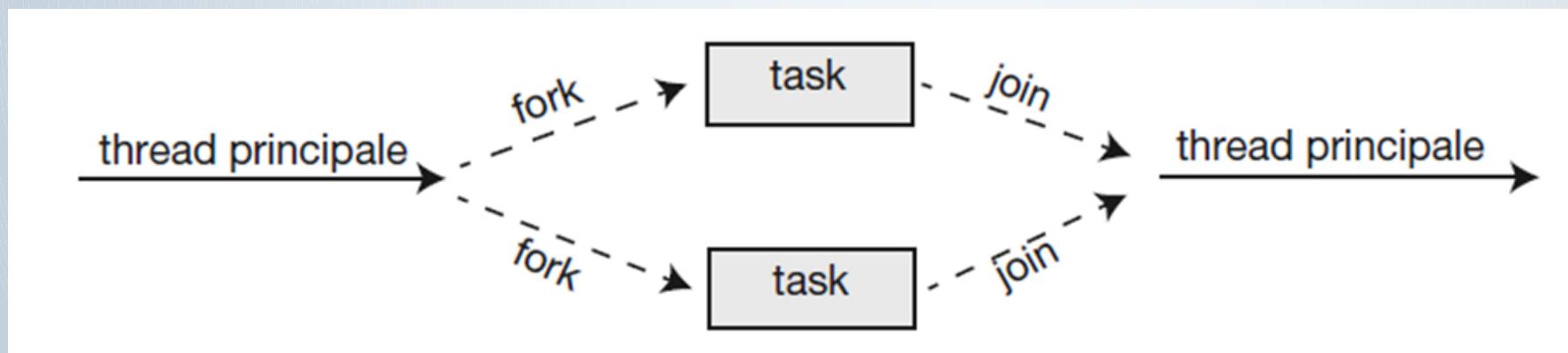
        /* Crea il gruppo di thread */
        ExecutorService pool = Executors.newCachedThreadPool();

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

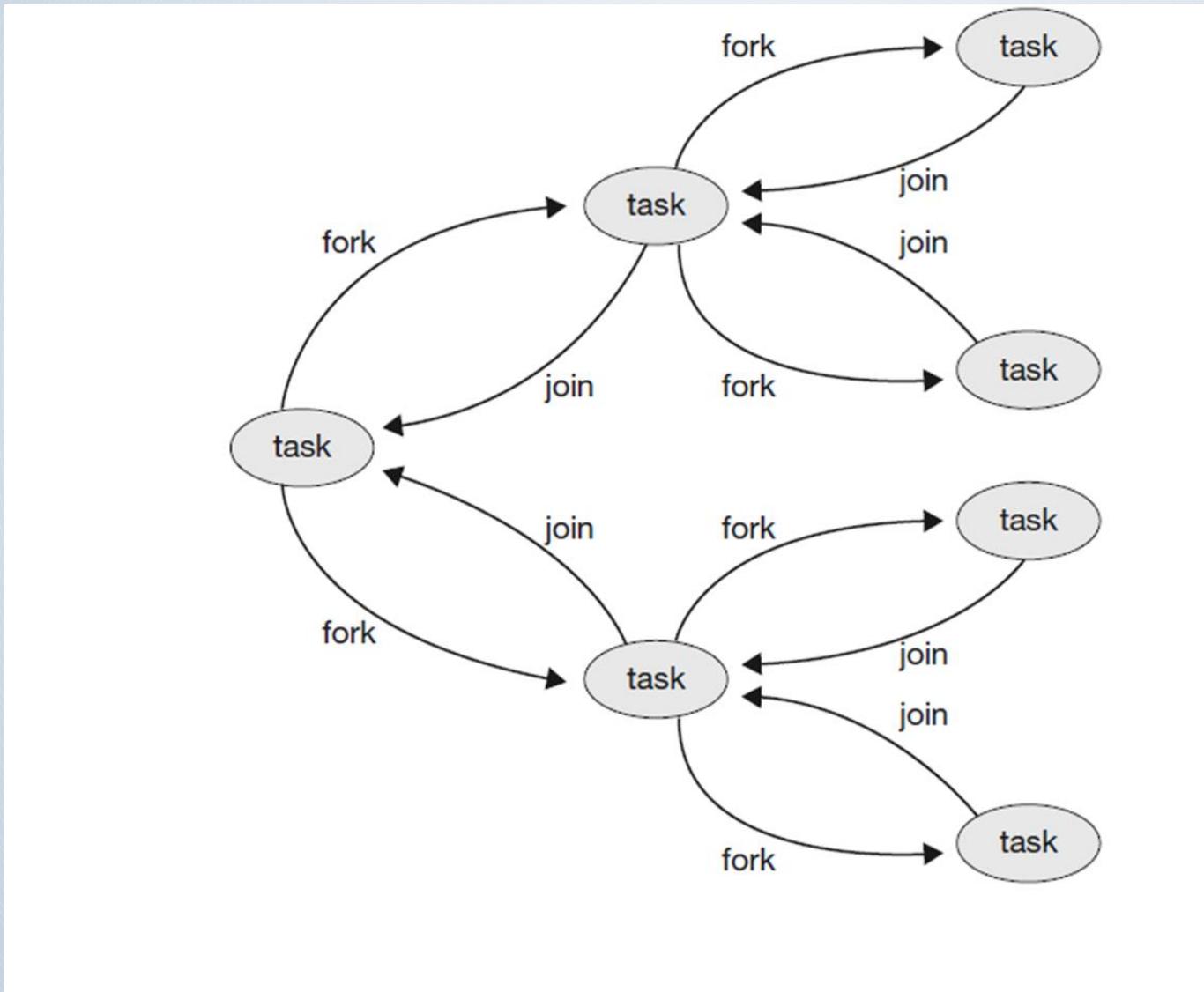
        /* Termina il gruppo quando tutti i thread hanno completato
           l'attività */
        pool.shutdown();
    }
}
```

Metodo fork-join

Secondo questo metodo 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.



Metodo fork-join



Open MP / 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.

Problematiche del multithreading

- ◆ Chiamate di sistema fork() ed exec(): Se la chiamata exec() non esegue immediatamente una fork(), in un sistema multithreading si potrebbe verificare la duplicazione di tutti i thread del processo genitore.
- ◆ Gestione dei segnali: Generazione di segnali asincroni.
- ◆ Cancellazione dei thread: se ad un thread sono state assegnate delle risorse o sta aggiornando dei file l'operazione di cancellazione potrebbe essere più complessa.

Cancellazione dei 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.

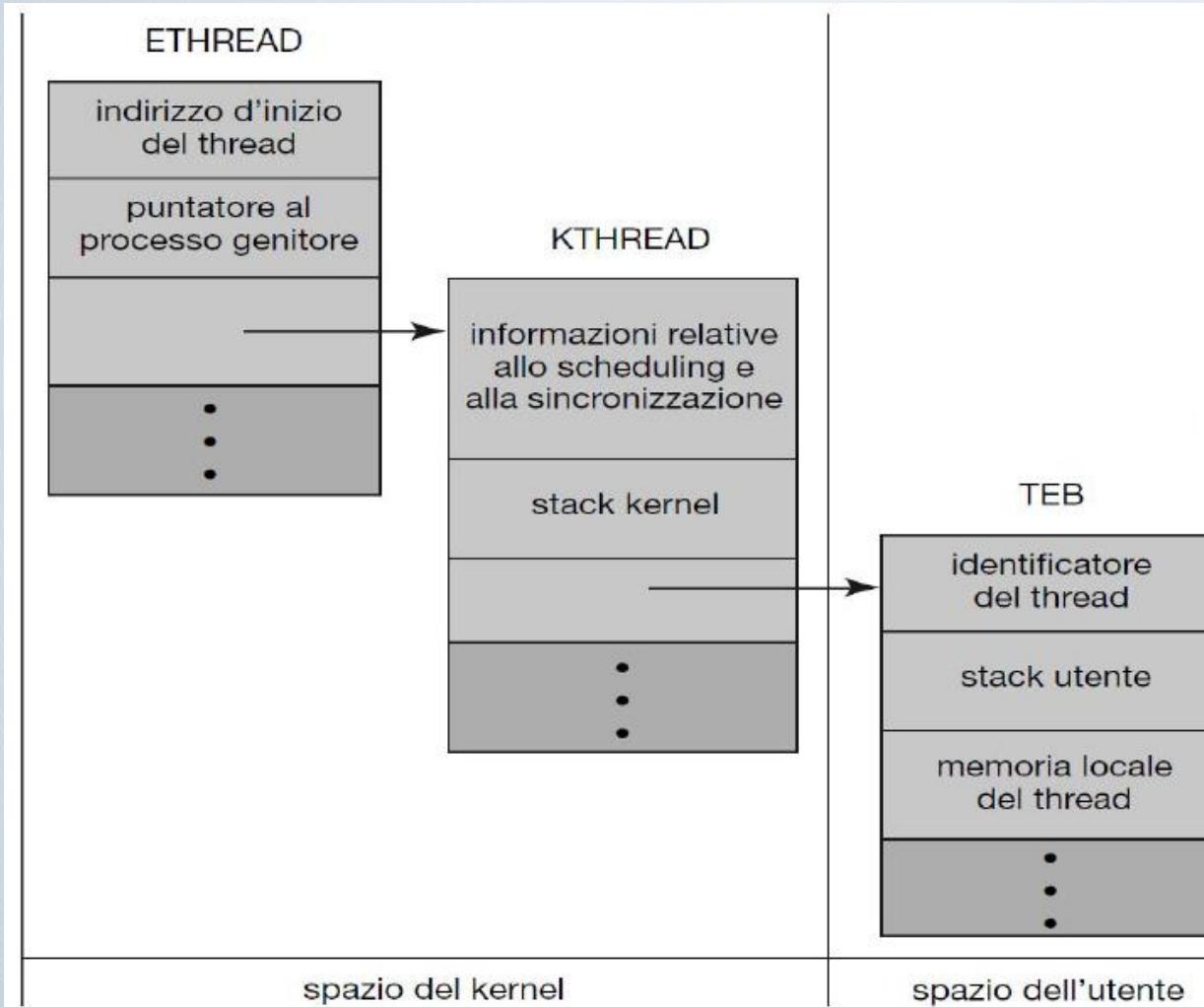
Thread in Windows

- ◆ Il sistema operativo windows impiega il modello uno a uno, i componenti di un thread includono: 5 caratteristiche
- ◆ Identificatore di thread univoco
- ◆ Un insieme di registri per lo stato
- ◆ Contatore di programma
- ◆ Stack utente e stack kernel
- ◆ Area di memoria privata

Thread in Windows

- ◆ L'insieme dei registri, gli stack e la memoria privata sono detti **contesto** dei thread le strutture principali sono:
 - ◆ ETHREAD (Executive thread block) – puntatore al processo a cui il thread appartiene e l'indirizzo di inizio.
 - ◆ KTHREAD (Kernel thread block) – include tutte le informazioni sullo scheduling del thread, e lo stack kernel.
 - ◆ TEB (Thread environment block) – contiene l'identificatore al thread, uno stack per la modalità utente e un vettore dati specifici del thread.

Thread in 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.

Windows vs. Linux

- ◆ 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
<code>CLONE_FS</code>	Condivisione delle informazioni sul file system
<code>CLONE_VM</code>	Condivisione dello stesso spazio di memoria
<code>CLONE_SIGHAND</code>	Condivisione dei gestori dei segnali
<code>CLONE_FILES</code>	Condivisione dei file aperti

Riferimenti - fonti

1. © Pearson Italia S.p.A. – Silberschatz, Galvin, Gagne, *Sistemi operativi*