

---

# Introduzione al Multithreading

Claudia Calidonna

*Istituto di Cibernetica C.N.R.*

# Argomenti principali

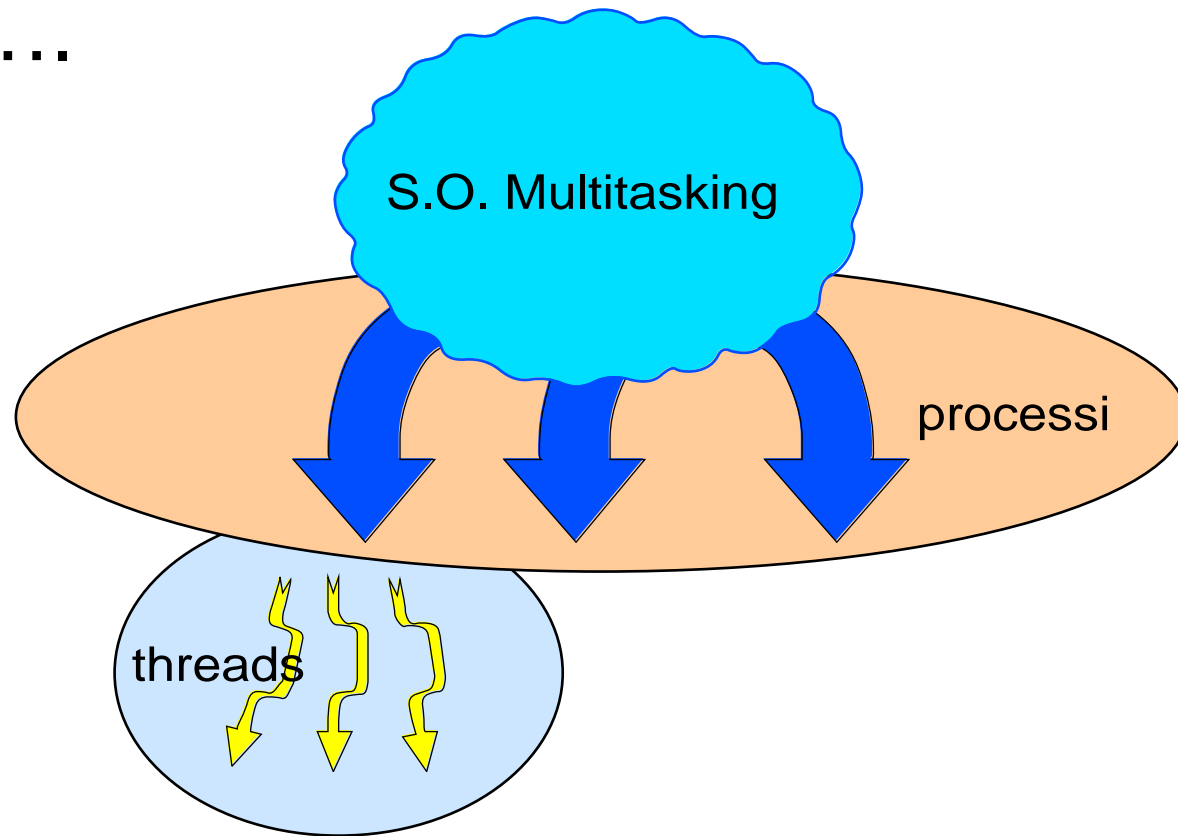
---

Parleremo di :

- Processi & Threads
- Operazioni sui threads ed eventuali confronti tra operazioni sui processi
- Threads a livello Utente e a livello Kernel

# Multitasking

Ad un istante ....



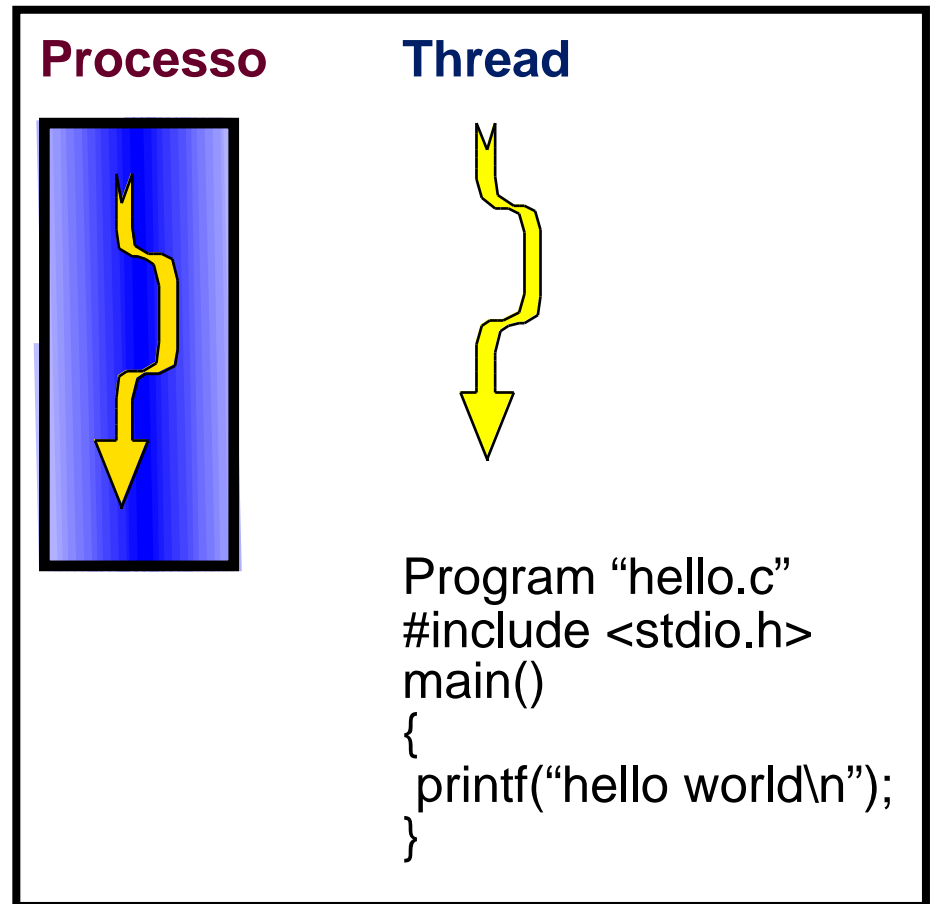
# **Che cosa è un thread**

---

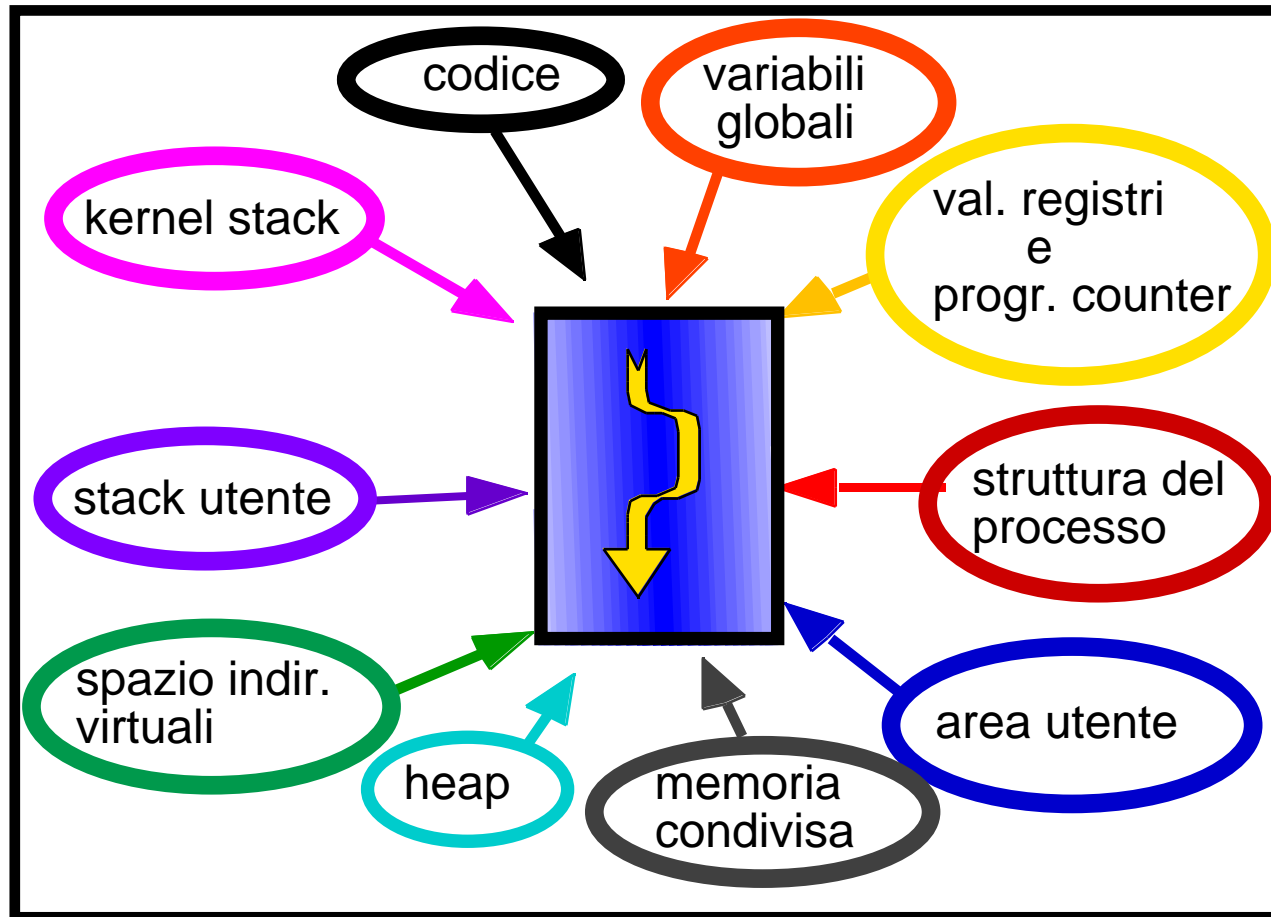
**è un singolo flusso di istruzioni eseguibili  
all'interno di un contesto di un programma**

# Tradizionale Processo UNIX

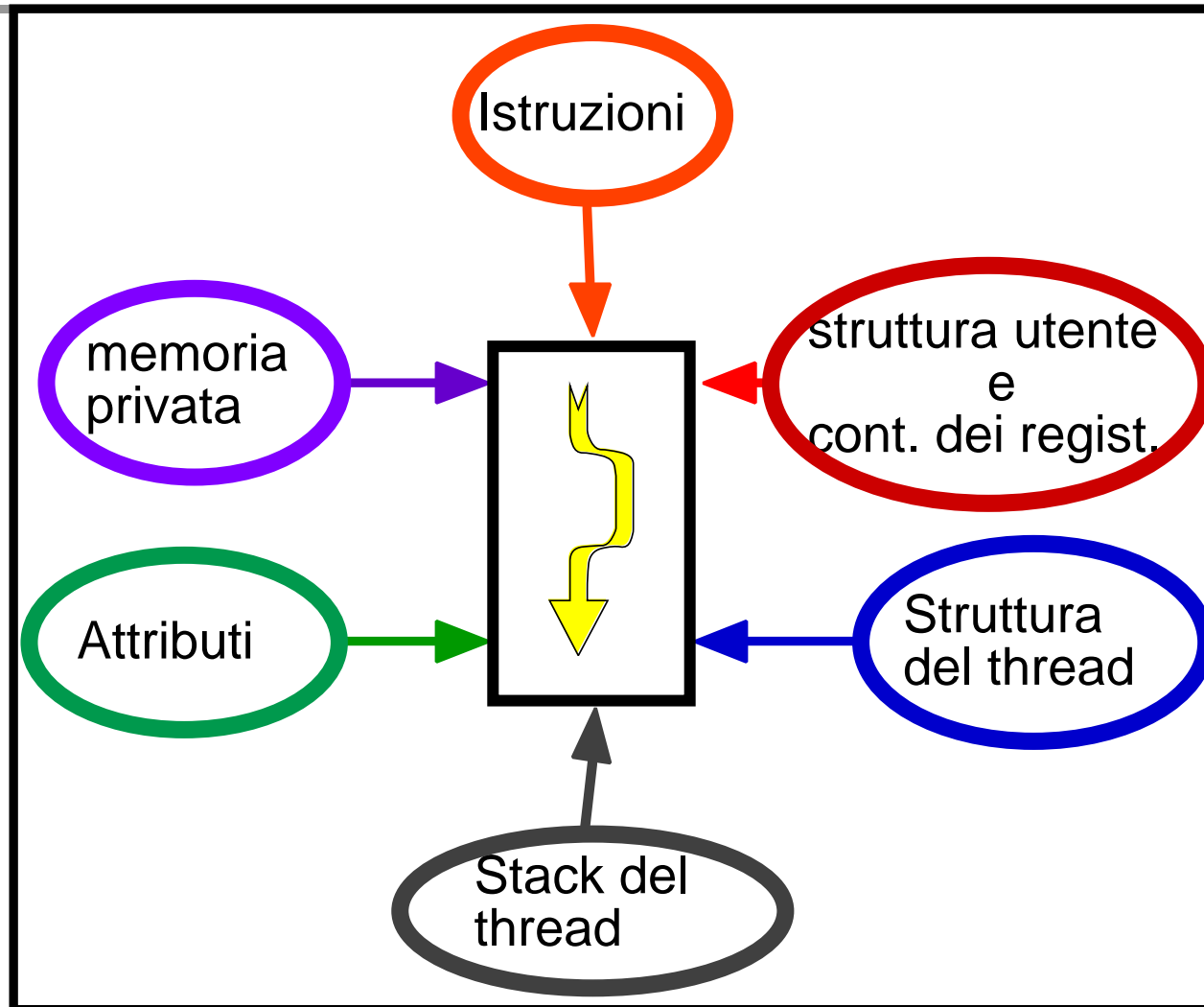
- possiede uno spazio degli indirizzi unico
- consente di isolare le esecuzioni dei programmi



# Processo



# Thread



# Relazione tra processo e thread

## Processo

- Dati
- Codice
- Stato del kernel
- Insieme dei registri della CPU



Include:  
Mappa memoria virt.  
Descritt. File  
Id utente  
.....

## Thread

- Program counter
- Registri generali
- Stack pointer
- .....

# Proprietà di threads di uno stesso processo

---

## threads in un processo condividono

- Stato del processo
- Spazio della memoria
- Codice
- Area dati

- Condivisione delle istruzioni
- Se un thread altera una variabile non locale al thread la modifica é visibile a tutti gli altri threads
- Se un thread apre un file, allora gli altri threads possono operare su questo file

# threads e processi

---

## *Processo*

- può comunicare con un altro processo
- può condividere variabili
- può condividere risorse
- può essere schedulato

## *Thread*

- può comunicare con un altro thread
- può condividere variabili
- può condividere risorse
- può essere schedulato

# Operazioni sui threads

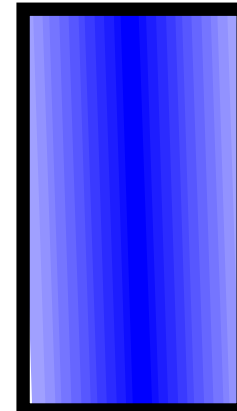
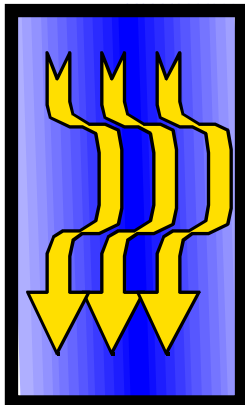
---

- Create
- Join
- Detach
- Binding
- Exit

# Threads-processi : funzioni

---

create( )	→	fork( )
detach( )	→	none
join( )	→	wait( )
exit( )	→	exit( )



# Attributi di un thread

---

- Scope
- Detach state
- Stack size
- Scheduling policy

# Esempio di creazione di un thread

---

```
int pthread_create (  
    pthread_t          *thread_id,  
    pthread_attr_t    *attr,  
    void              *(*start_routine)(void *),  
    void              *arg  
);
```

# Stati di un processo (1)

---

- Creation: stato di creazione, il kernel crea il contesto per il child. Parent é nel running state e child é nel creation state
- Runnable: stato in cui il processo é pronto per partire
- Running: il processo é nello stato di esecuzione. Può essere eseguito in modalità utente o kernel

# Stati di un processo (2)

---

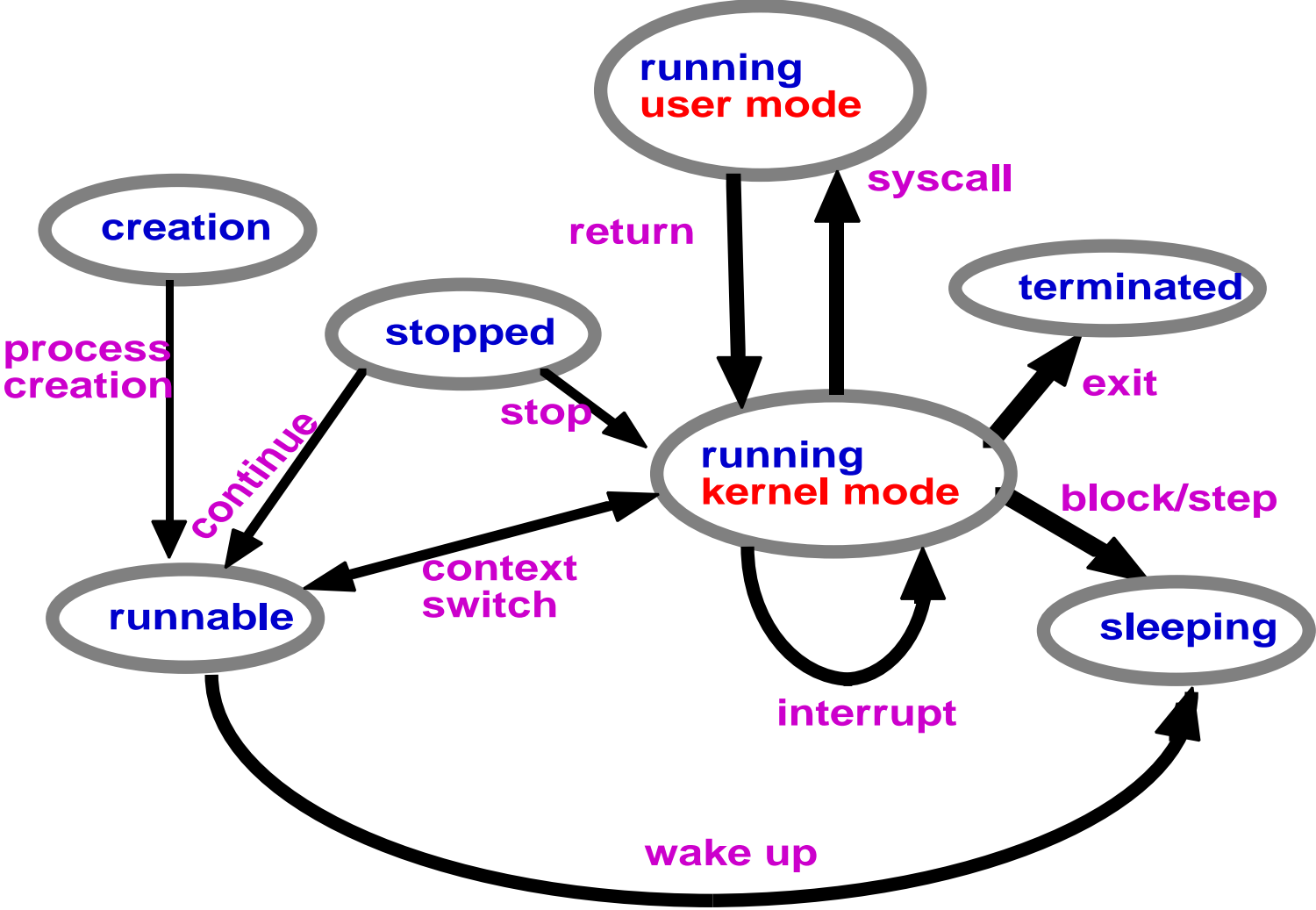
- Sleeping: il processo é in attesa, sta attendendo oppure é bloccato
- stopped: il processo é nello stato di stop, attende un signal per poter continuare
- terminated: il processo é terminato, a causa di una chiamata a `exit()`

# Stato di un thread

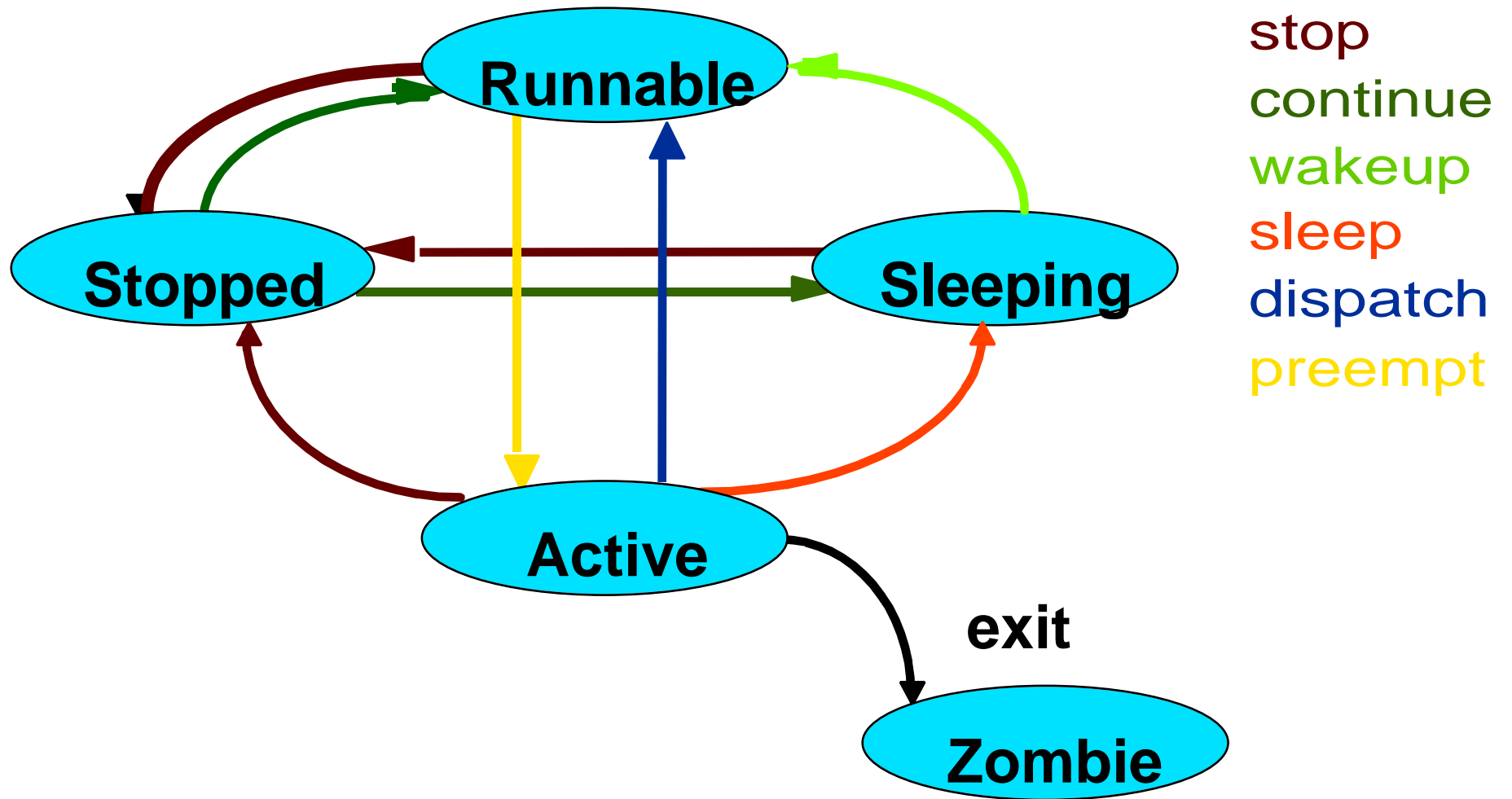
---

- Active:
- Runnable:
- Sleeping:
- Stopped:
- Zombie:

# Stati di transizione dei processi



# Stati di transizione di un thread



# Threads a livello utente

---

- Che **non** eseguono chiamate a sistema, vengono eseguiti più velocemente
- Per la loro gestione non vengono effettuate chiamate a sistema.
- Il kernel ignora l'esistenza degli user level threads, dunque ogni operazione su di essi è trasparente al kernel

# Riepilogo

---

abbiamo parlato di :

- che cosa é un thread
- in che cosa si differenzia da un processo
- principali operazioni sui threads e sui processi
- differenza tra threads a livello utente e a livello kernel

# Vantaggi di utilizzo dei threads

---

- Concorrenza
- Aumento del throughput
- Struttura di un programma
- Sfruttamento migliore delle risorse di sistema
- Sorgente unico per diverse piattaforme

# Concorrenza

---

- Possibilità di suddividere più flussi di controllo in più threads, e dunque esistenza di flussi di controllo multipli e contemporanei.
- Possibilità di eseguire più threads sulle molteplici CPU esistenti nel sistema.

# Aumento del throughput

---

- Un programma che richiede un I/O sincrono, utilizzando un unico thread, deve sempre aspettare la richiesta del servizio dal sistema operativo.
- Su un sistema monoprocesore, per un I/O sincrono, il thread permette la sovrapposizione di un medesimo processo mentre sta utilizzando un servizio di livello più alto.
- Mentre si sta effettuando la richiesta si può eseguire un altro thread

# Struttura di un Programma

---

I programmi possono essere strutturati in modo più efficiente.

# Risorse di Sistema

---

Programmi che usano due o più processi che condividono dati hanno bisogno di più spazio e tempo rispetto ai threads :

- ogni processo deve mantenere sia lo spazio degli indirizzi che lo stato del sistema
- I threads usano parte delle risorse utilizzate dai processi

# sorgente multipiattaforme

---

- I threads a livello utente, in particolare la versione POSIX, consentono di scrivere dei codici portabili su varie architetture.
- La libreria threads Solaris, permette di scrivere codice per diverse architetture SUN che supportano questo modello, indipendentemente se la macchina ha uno o più processori

# Quando utilizzare i threads?

---

- Livello utente: programmi che richiedono individuano istruzioni da effettuare su una enorme quantità di dati in modo indipendente
- Livello kernel:
  - Tasks indipendenti
  - Operazioni da Server
  - Tasks che debbono essere ripetuti

# Tipi di scheduling

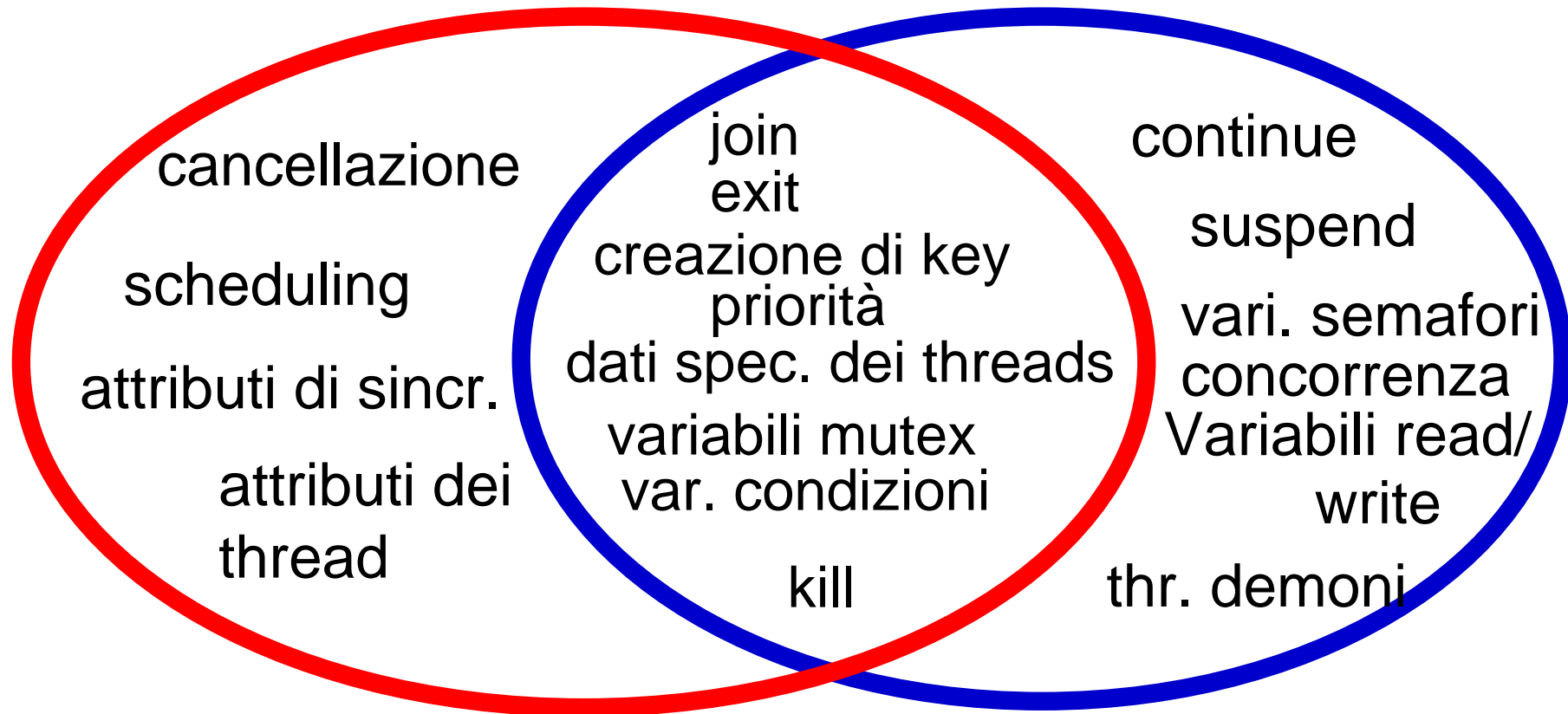
---

- Globale al sistema (kernel)
- Locale al livello di processo

# Libreria e API

---

---



# Bound e Unbound user level threads

---

- ❖ Bound thread : thread utente legato direttamente ad un kernel thread
- ❖ Unbound thread : thread utente eseguito sulla base di ogni kernel thread disponibile nei processi.

Uno degli attributi del thread permette il binding di threads e livello utente a threads a livello kernel

# Sincronizzazione

---

Esistono tre meccanismi fondamentali:

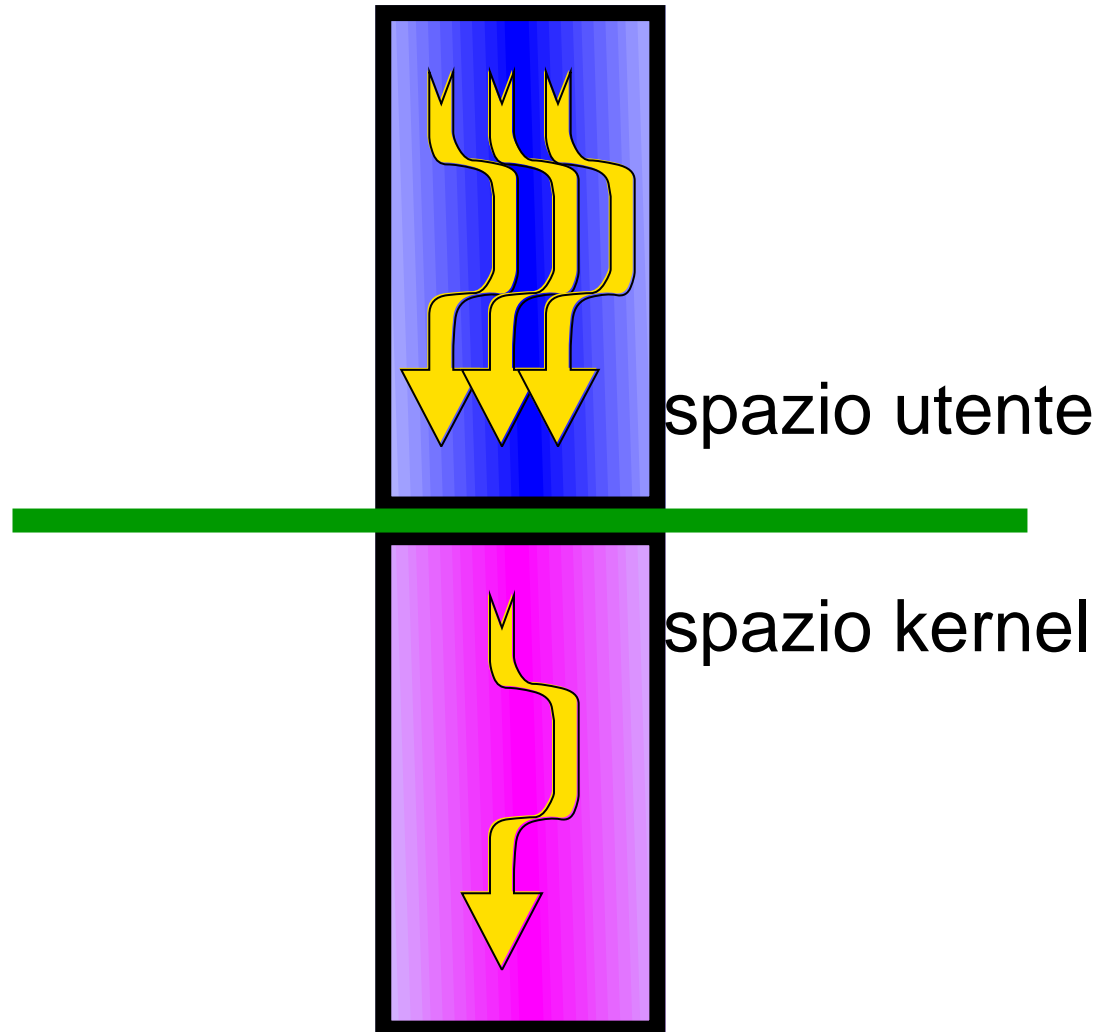
- Mutex (*mutual exclusion*): sincronizzazione di oggetti condivisi - ad es. variabili globali
- Variabili di condizioni: sincronizzazione su verificarsi di una condizione (tra threads)
- Semafori: sincronizzazione ad un oggetto o a piu' oggetti

# Kernel threads scheduling

---

- **molti** threads -a- **un** processo
- **un** thread -a- **uno** processi
- **molti** threads -a- **molti** processi
- doppio livello : **molti -a- uno** e **uno - a- uno**

# Molti a uno



# Vantaggi e svantaggi Mx1

---

## *Vantaggi*

- ☺ Concorrenza
- ☺ Veloce gestione threads
- ☺ Programmazione strutturata
- ☺ Codice pronto sia per MxM che 1x1

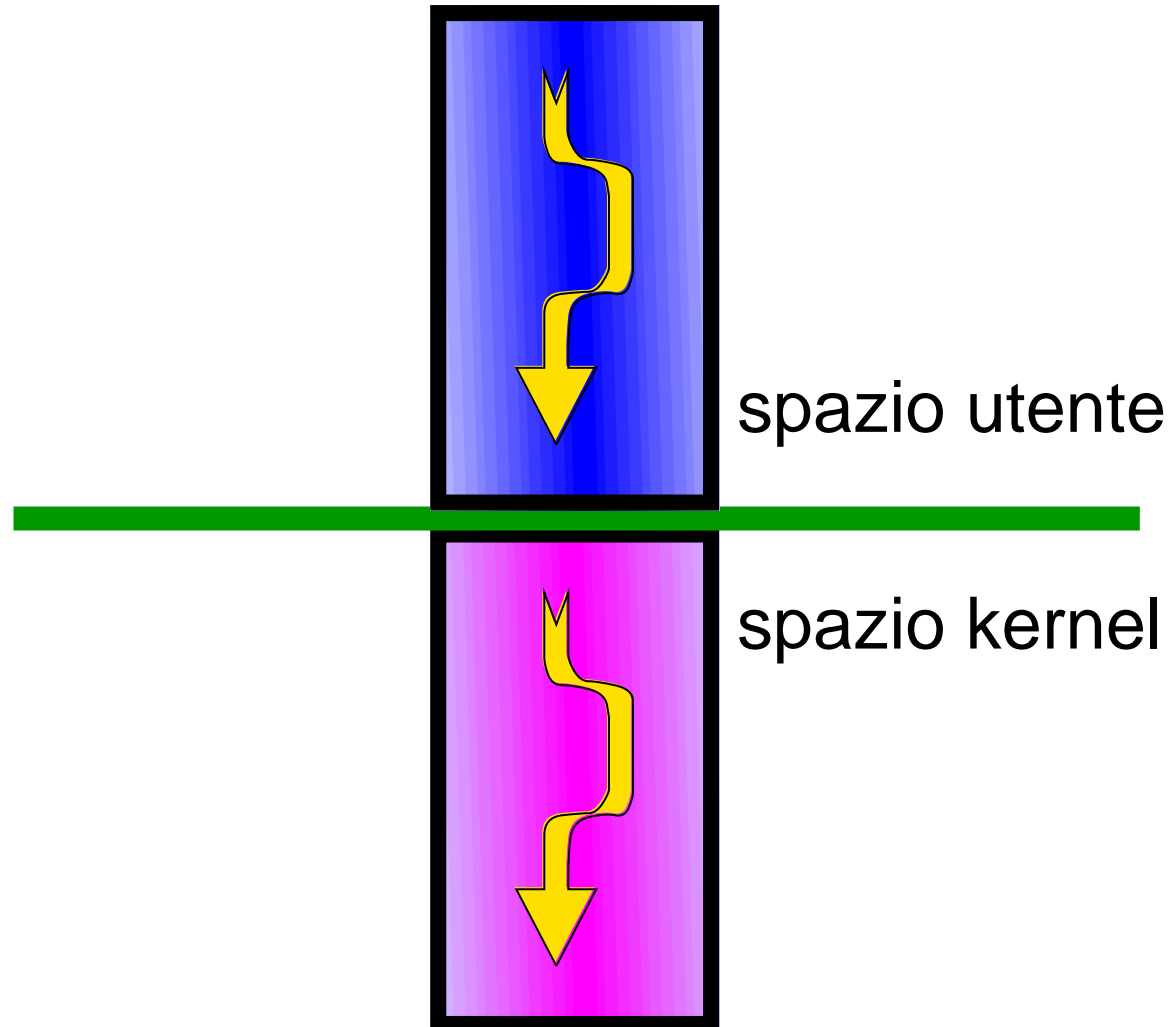
## *Svantaggi*

- ☹ No parallelismo fisico
- ☹ Blocking :
  - ☺ Sleep per un thr. → selezione di un altro thr.
  - ☺ thr. esegue read() su dispositivo nonblocking wrapper trasforma blocking in noblocking e select() prende i dati
- ☹ Profiling

# Uno a Uno

---

---



# Vantaggi e svantaggi 1x1

---

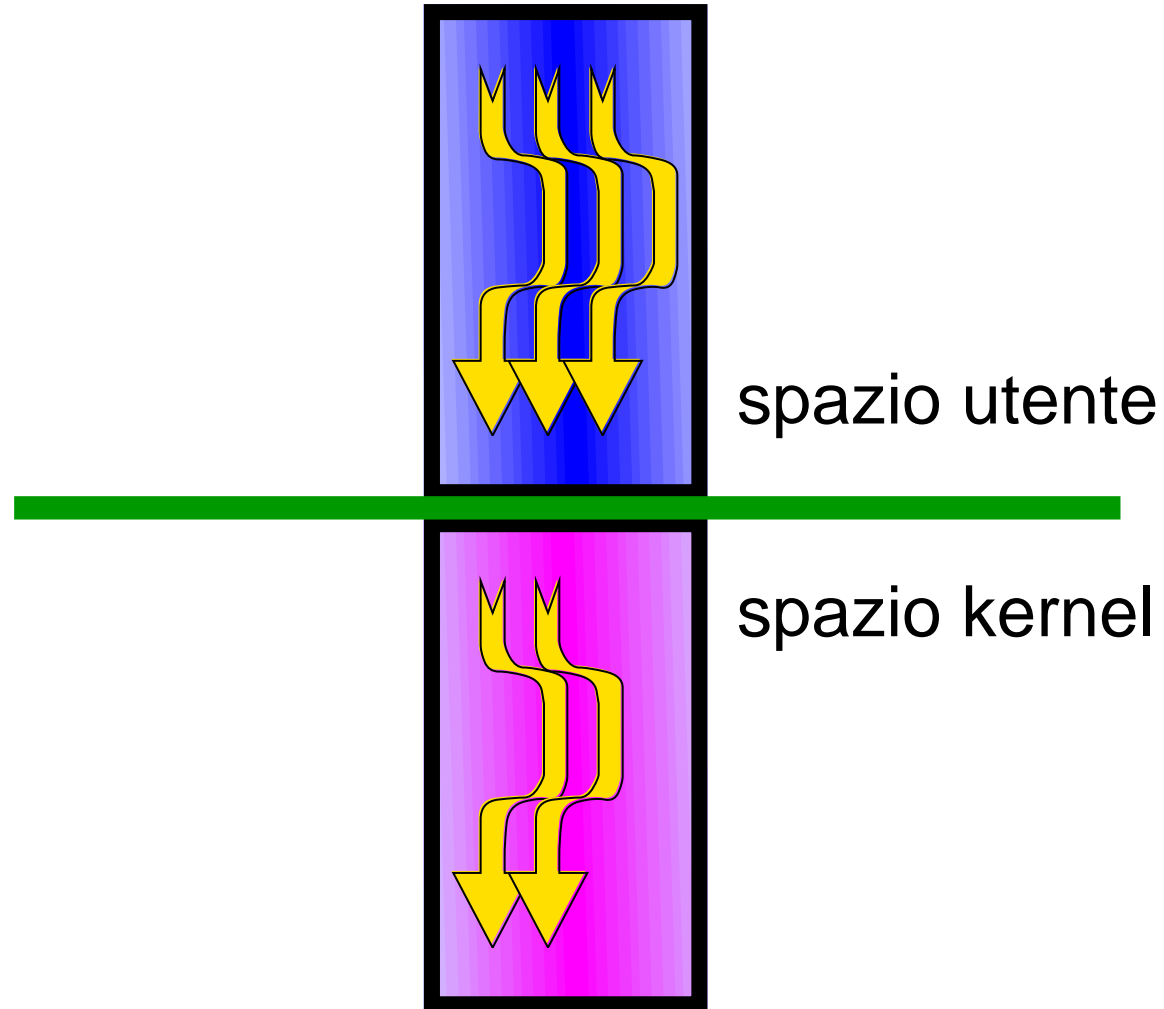
## *Vantaggi*

- ☺ Esecuzione parallela
- ☺ Profiling
- ☺ Vantaggi di Mx1

## *Svantaggi*

- ☹ Overhead di gestione maggiore (+ contesto da gestire)
- ☹ Utilizzo più risorse

# Molti a Molti



# Vantaggi e svantaggi MxM

---

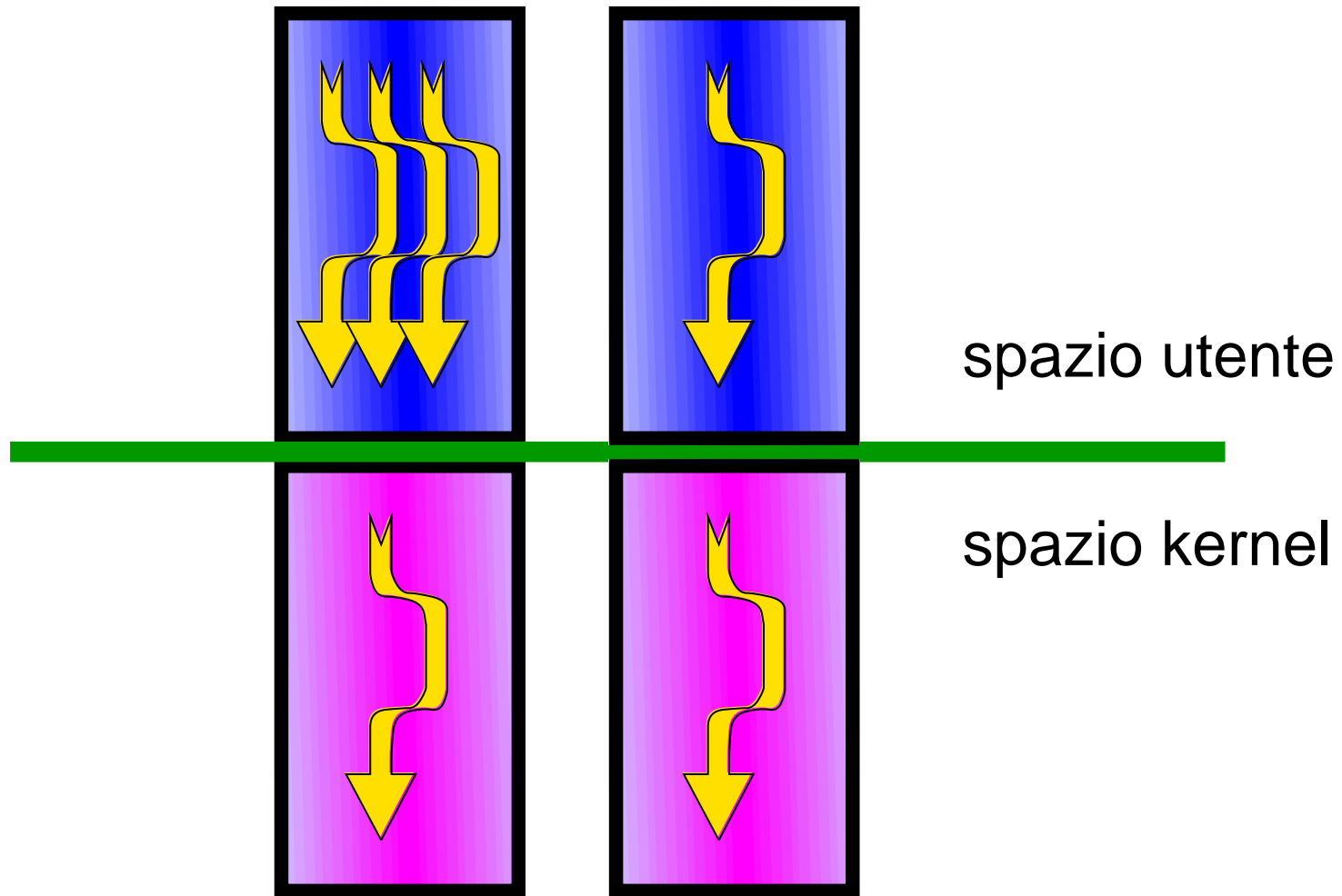
## *Vantaggi*

- ☺ Vantaggi del modello Mx1 e 1x1
- ☺ Uso di bound e unbound threads

## *Svantaggi*

- ☹ Interazione di scheduling di livello kernel e utente
- ☹ Svantaggi di Mx1 e 1x1

# Doppio livello



# Vantaggi e svantaggi

## Mx1 & MxM

---

### *Vantaggi*

☺ Vantaggi di tutti i  
modelli visti finora

### *Svantaggi*

☹ Svantaggi di tutti i  
modelli visti finora

# Riepilogo

---

Oggi abbiamo parlato di:

- Vantaggi di utilizzare sistemi multithreading
- Caratteristiche e proprietà dei threads a livello kernel ed a livello utente
- Sincronizzazione
- Binding
- Modelli di scheduling di threads a livello utente e threads a livello kernel