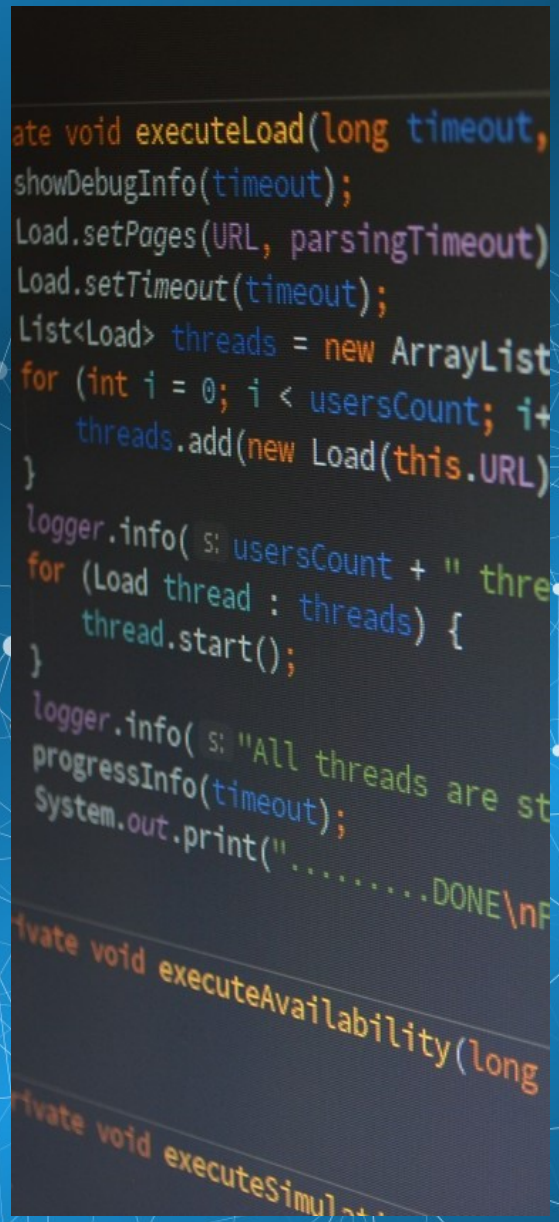
ISLab Intelligent Systems Laboratory

UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

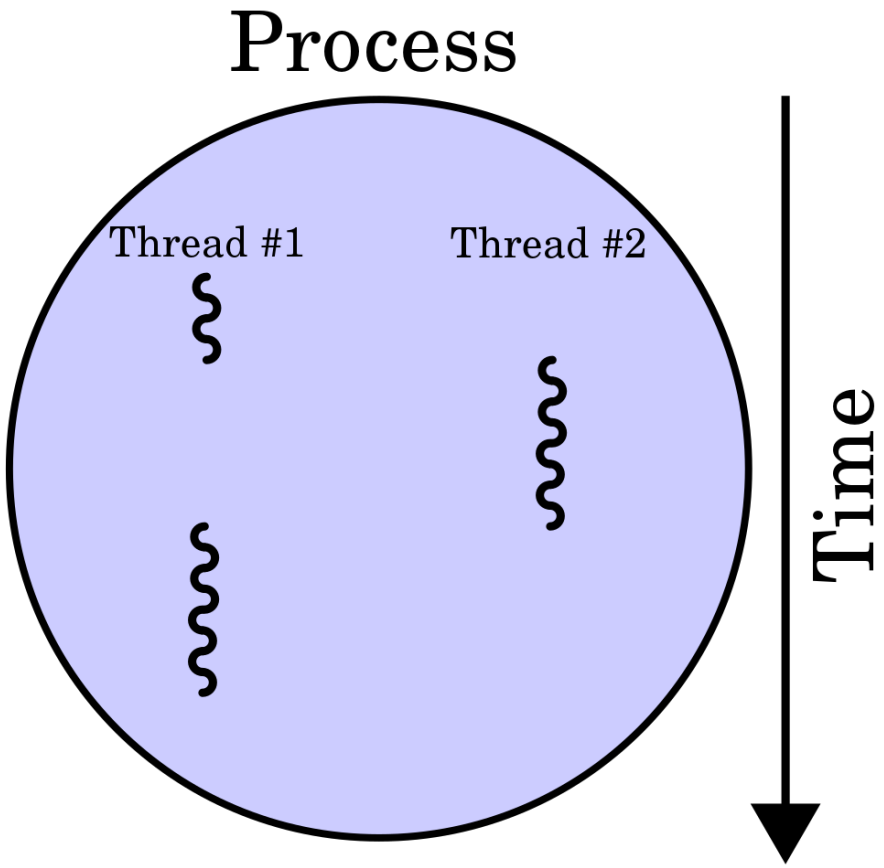
# Thread SMP Microkernel

Prof. Giuseppe Pirlo | Prof. Donato Impedovo | Dott. Stefano Galantucci

Architettura degli Elaboratori e Sistemi Operativi @ Corso di Laurea in Informatica



# Thread




Un thread è un **flusso di esecuzione indipendente (traccia)** all'interno di un processo.

Un processo può essere diviso in più thread per:

- Ottenere un parallelismo dei flussi di esecuzione all'interno del processo
- Gestire chiamate bloccanti o situazioni di risposta asincrona

# Thread

Esempio:

Dobbiamo implementare un web server. Se fosse implementato come un processo monothread potremmo gestire solo un client alla volta, e le altre richieste verrebbero ignorate (neanche poste in coda).

Possiamo risolvere la situazione in due maniere differenti:

Implementare il server come un processo che resta in attesa delle richieste e:

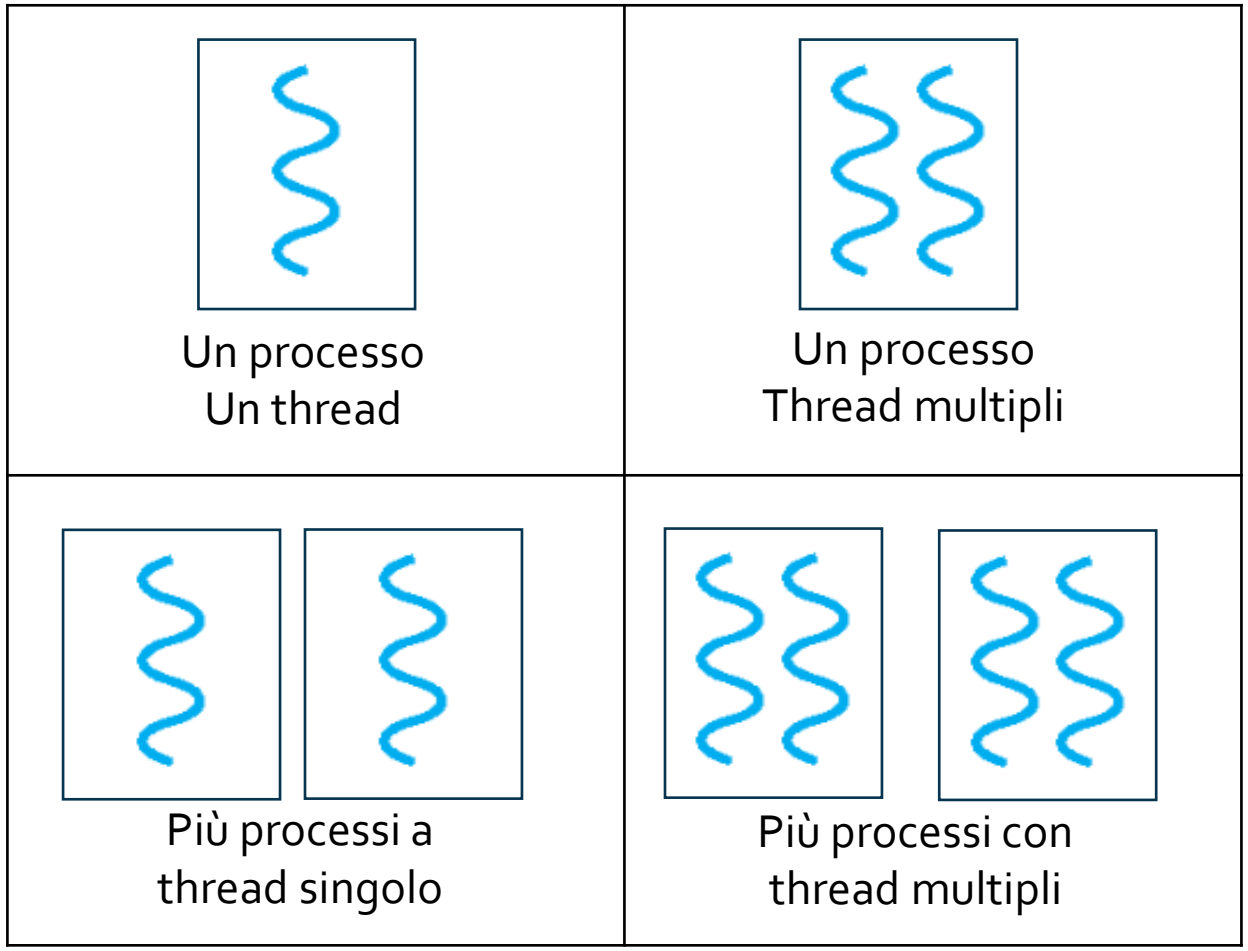
- Avviare un nuovo processo che processi la singola richiesta
- Avviare un nuovo thread che processi la singola richiesta

Avviare un nuovo processo è molto più oneroso che avviare un nuovo thread

Effettuare il context switch di un processo è molto più oneroso che quello di un thread

# Multithreading

Il multithreading è la capacità di un Sistema Operativo di supportare più thread per ogni processo




**Singolo processo a singolo thread**

- MS-DOS (il thread esiste nel processo)

**Processi multipli a singolo thread**

- UNIX

**Multithread**

- Windows
- Solaris
- MAC
- OS/2

# Thread VS Processi

## Processo

Possiede delle risorse:

- Spazio di indirizzamento virtuale che contiene l'immagine del processo
- Può chiedere ulteriore memoria
- Può chiedere il controllo di canali di I/O
- Può chiedere il controllo di dispositivi
- Può chiedere files

Unità di esecuzione:

- Ha uno stato (ready, running, blocked, suspended…)
- Ha una priorità
- Deve essere schedulato

Informazioni contenute nel Process Control Block (PCB)

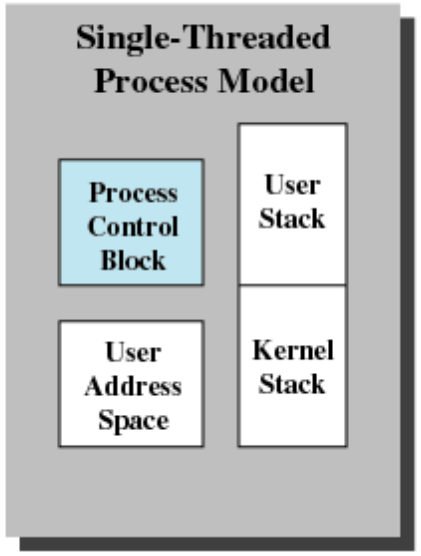
## Thread

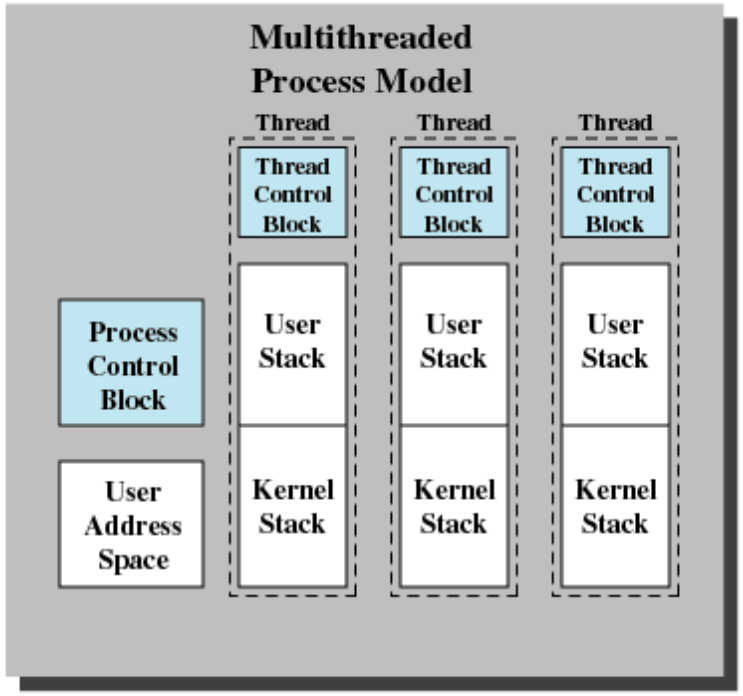
- Non possiede risorse, in quanto utilizza quelle del processo
- Viene anche chiamato **Light Weight Process (LWP)**

- Ha un proprio stato, una propria priorità e deve essere schedulato, ma tra i Thread e sottostando alla schedulazione del processo

Informazioni contenute nel Thread Control Block (TCB)

# Multithreading




I thread

- Condividono lo stato e le risorse del processo a cui appartengono
- Risiedono nello stesso spazio di indirizzamento
- Hanno accesso agli stessi dati

Dunque è più facile condividere le informazioni

# Vantaggi dei Thread

- Tempo di creazione di un nuovo thread < tempo di creazione di un nuovo processo
- Tempo di terminazione di un thread < tempo di terminazione di un processo
- Tempo necessario allo switch tra threads all'interno dello stesso processo < tempo di switch tra processi
- I threads all'interno di uno stesso processo condividono memoria e files: scambio dati senza richiedere l'intervento del kernel. Necessità di sincronizzare le atività dei threads

Alcuni esempi:

- Esecuzione in foreground e in backgroud: foglio di calcolo: un thread gestisce il menu e legge i comandi, un altro thread esegue i comandi e aggiorna il foglio
- Elaborazione asincrona: elaboratore di testo - un thread di scarico su disco ad ogni minuto evita perdite per cadute di tensione
- Velocità di esecuzione: lettura e calcolo fatti da threads diversi aumentano la velocità

# Qualche difficoltà

- La sospensione di un processo richiede che tutti i thread siano sospesi contemporaneamente perché si deve liberare spazio in memoria e tutti i thread utilizzano lo stesso spazio di memoria condivisa
- La terminazione di un processo richiede che tutti i thread siano terminati

# Stati dei Thread

- Ready
- Running
- Blocked

Il suspended non è presente in quanto non ha senso per un thread (è presente a livello di processo): se un processo viene swappato, lo stesso avviene per tutti i suoi thread

# Operazioni di base

Creazione

- creazione di un processo -> creazione di un thread
- Un thread può creare altri thread

Blocco (attesa di un evento)

- salvataggio del contesto per il thread: PC, Stack pointer, registri CPU

Sblocco

- lo stato nel TCB viene modificato (Blocked -> Ready)
- il thread viene accodato a quelli in attesa di processore

Terminazione

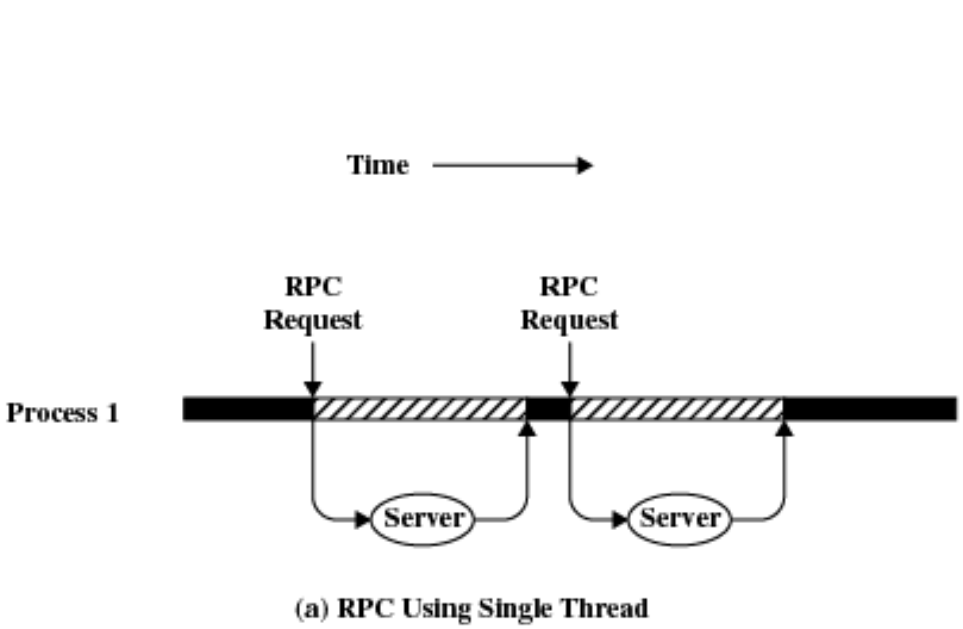
- deallocazione del contesto registri, deallocazione stack

Il blocco di un thread blocca l'intero processo?

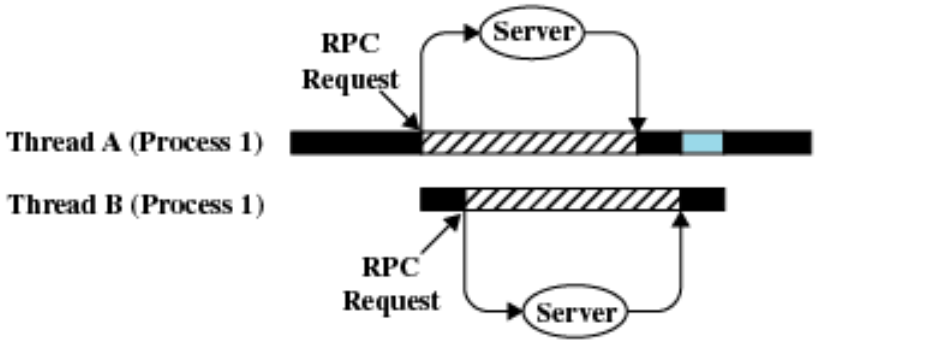
# Esempi di multithreading

Il **Remote Procedure Call (RPC)** è la chiamata da parte di un processo a una procedura attiva su un elaboratore diverso dal chiamante

Vediamo il caso di due chiamate RPC diverse a diversi host




(a) RPC Using Single Thread




(b) RPC Using One Thread per Server (on a uniprocessor)

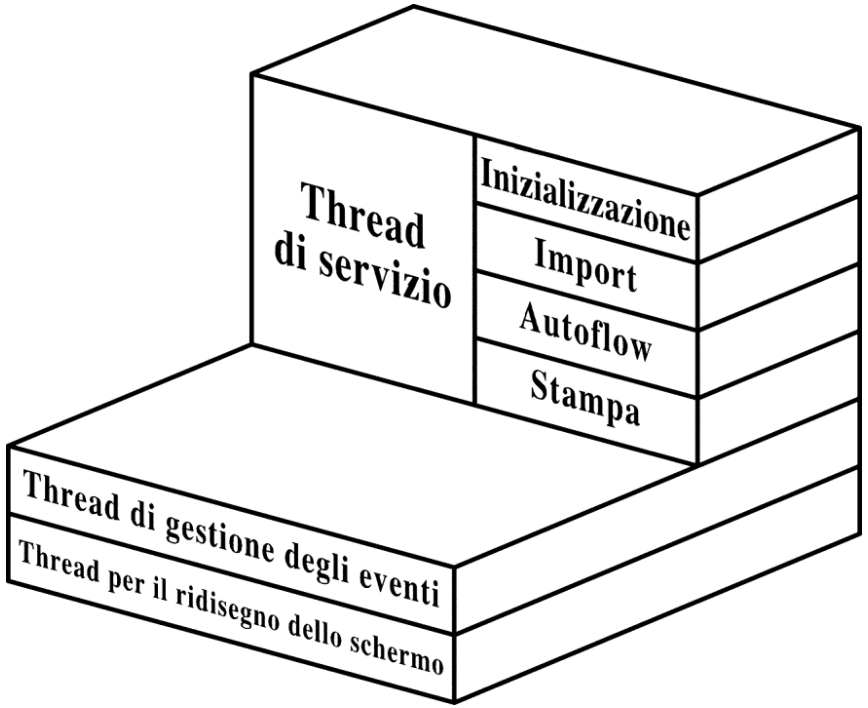
Blocked, waiting for response to RPC

Blocked, waiting for processor, which is in use by Thread B

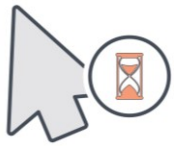
Running

**Figure 4.3 Remote Procedure Call (RPC) Using Threads**

# Esempi di multithreading




NB: esistono attività bloccanti per tutti i threads: compare il cursore «busy»



Programma di video-scrittura, gestione e pubblicazione di pagine su desktop.

Tre thread sempre attivi:

- Gestione degli eventi
- Gestione dei servizi (stampa, lettura dati, disposizione testo, attivazione altri Thread)
- Disegno dello schermo

Esempio sorrimento pagina con barra laterale:

- Il thread eventi controlla la barra di scorrimento
- Il thread di ridisegno dello schermo ridisegna la pagina in base allo spostamento

Necessità di sincronizzazione tra i due threads

# Categorie di Thread

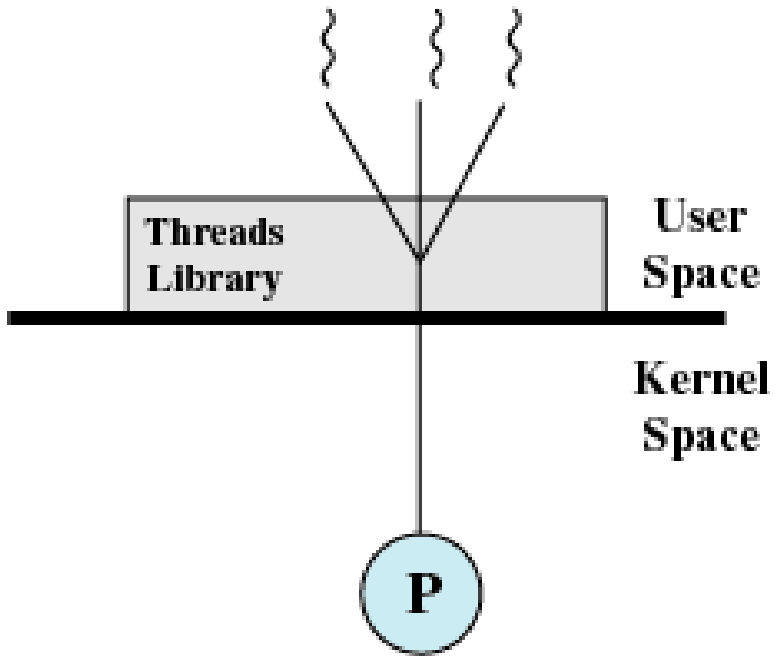
## User Level Thread (ULT)

- **Realizzati tramite librerie senza l'intervento del kernel**
  - Es. di librerie: Posix Pthread, Mach C-threads, UI-threads Solaris2
- **Trasparenti al kernel**
- Svantaggio: se il kernel è a singolo thread il blocco del thread di livello utente blocca l'intero processo (NB: il SO continua a schedulare processi)

## Kernel Level Thread (KLT)

- **Il kernel si occupa della creazione, scheduling e gestione**
- **Possono essere eseguiti su diversi processori**
- Gestione più lenta degli ULT

# User Level Thread (ULT)




Il lavoro di gestione dei threads è svolto dalla libreria utente

- Il kernel ignora l'esistenza dei threads
- Modello Molti a Uno

La libreria permette:

- Creazione e distruzione dei threads
- Scambio messaggi tra threads
- Schedulazione
- Salvataggio e caricamento dei contesti dei thread

Tali attività sono **svolte all'interno del processo utente**, pertanto **il kernel continua a schedulare i processi come unità a se stanti**

# User Level Thread (ULT)

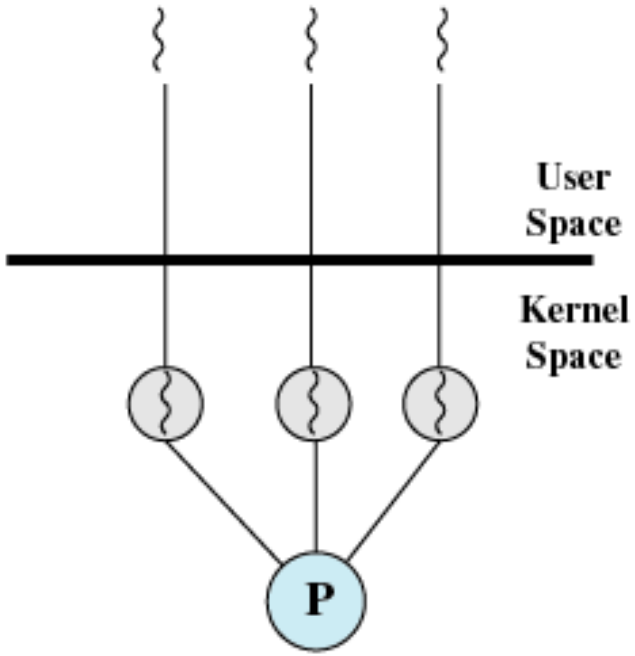
**VANTAGGI:**

- Risparmio di sovraccarico:
  - il cambio di Thread avviene all'interno dello spazio di indirazzamento utente
  - Non viene richiesto l'intervento del Kernel
- Schedulzione diversa per ogni applicazione:
  - Ottimizzazione in base al tipo di applicazione
- ULT eseguito da qualsiasi sistema operativo:
  - Libreria a livello utente condivisa dalle applicazioni

**SVANTAGGI:**

- La chiamata a sistema da parte di un thread blocca tutti i thread del processo
- Il kernel assegna un processo ad un singolo processore quindi non si può avere multiprocessing a livello di thread (thread dello stesso processo su più processori)
- Soluzioni Parziali:
  - Sviluppo dell'applicazione a livello di processi (addio vantaggi dei thread)
  - jacketing: conversione di una chiamata bloccante in una non bloccante. Es.: Nel caso di I/O si invoca una procedura di jacketing che verifica se il dispositivo è occupato, in caso affermativo il thread passa in ready e un altro thread va in run.

# Kernel Level Thread (KLT) puro




(b) Pure kernel-level

Il lavoro di gestione dei threads è svolto dal Kernel: modello uno a uno

A livello utente una API consente l'accesso alla parte del Kernel che gestisce I thread

Il kernel mantiene info su:

- Contesto del processo
- Contesto dei threads
- Scambio messaggi tra threads

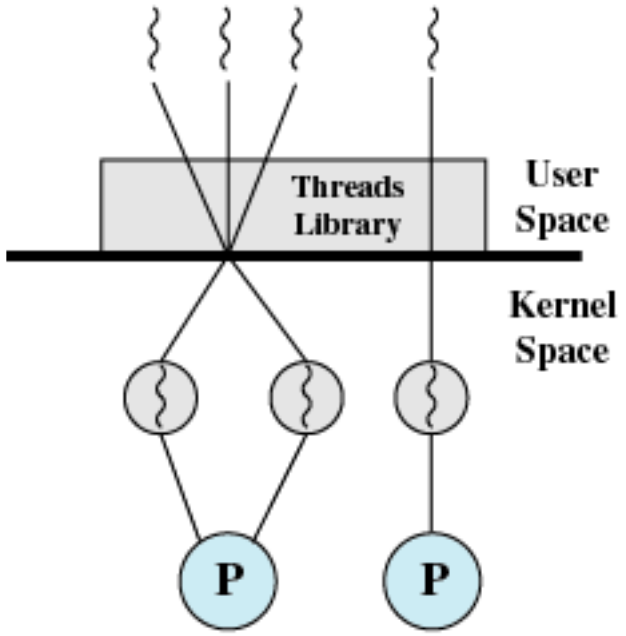
**Schedulazione effettuata a livello di thread**

- se un thread di un P è bloccato, un altro thread dello stesso processo può essere eseguito
- Thread di uno stesso P possono essere schedulati su diversi processori

**SVANTAGGI:**

Overhead: trasferimento del controllo da un thread ad un altro richiede l'intervento del kernel

# Approcci misti




(c) Combined

Modello molti a molti: più thread di livello utente sono in corrispondenza con più thread di livello kernel

I Thread sono creati nello spazio utente

Vari thread di uno stesso processo possono essere eseguiti contemporaneamente su più processori

Una chiamata bloccante non blocca necessariamente l'intero processo

Necessità di comunicazione fra kernel e libreria di thread per mantenere un appropriato numero di kernel thread allocati all'applicazione.

Light Weight Process (LWP) – struttura intermedia appare alla libreria dei thread utente come un processore virtuale sul quale schedulare l'esecuzione.

Es. Una applicazione CPU-bound su un sistema monoprocessore implica che un solo thread per volta possa essere eseguito, quindi per essa sarà sufficiente un unico LWP per thread.

Una applicazione I/O-bound tipicamente richiede un LWP per ciascuna chiamata di sistema bloccante

# Relazione tra Thread e Processi

| Thread : Processi | Descrizione | Sistemi |
|---|---|---|
| 1:1 | Ogni thread di esecuzione è un processo unico con il proprio spazio di indirizzamento e le proprie risorse | Molte implementazioni di UNIX |
| M:1 | Ogni processo ha associato un proprio spazio di indirizzamento e delle risorse. In ogni processo si possono creare ed eseguire molti thread | Windows NT<br>Solaris<br>OS/2<br>OS/390<br>MACH |
| 1:M | Un thread può spostarsi da un processo all'altro: ciò permette di spostare facilmente i thread tra sistemi diversi | Ra(Clouds)<br>Emerald |
| M:M | Combina le proprietà degli approcci M:1 e 1:M | TRIX |

Ambienti distribuiti: thread possono spostarsi tra più calcolatori

# Symmetric Multi Processing (SMP)

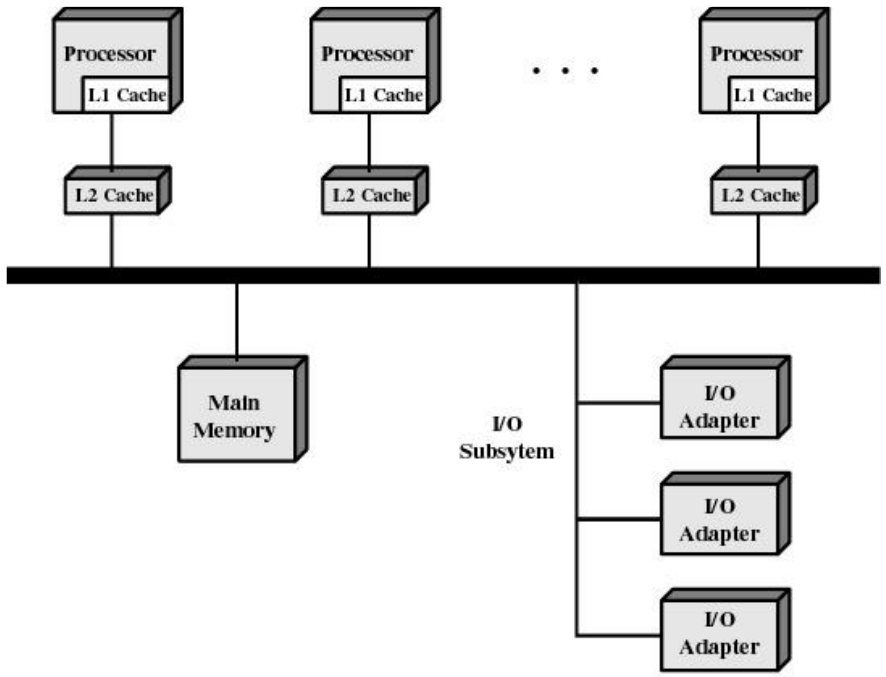



Figure 4.9 Symmetric Multiprocessor Organization

Un calcolatore con molti processori

- I processori condividono le stesse risorse
- Tutti i processori possono effettuare le stesse funzioni
- Ogni processore esegue una stessa copia del SO
- Ogni processore gestisce la schedulazione dei processi o thread disponibili

Difficoltà:

- I processori non devono schedulare lo stesso processo
- Comunicazione tra processori: memoria condivisa (possibilità di effettuare accessi simultanei alla memoria – memoria multiporta)
- Coerenza della $: RAW, WAR, RAR, WAW (risolti a livello hardware)

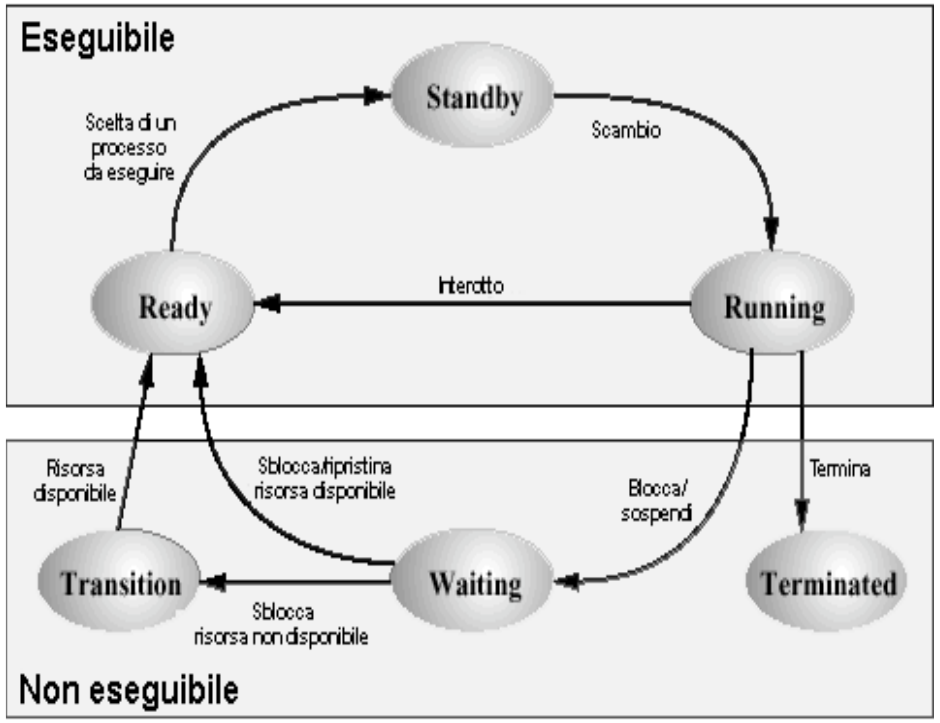
# Symmetric Multi Processing (SMP)

Il multiprocessore **deve essere trasparente all'utente**: il programmatore deve operare come se fosse in multiprogrammazione su monoprocessore

Punti critici della progettazione di un Sistema Operativo per SMP:

- Processi e Thread del Kernel concorrenti: l'esecuzione contemporanea su diversi processori non deve compromettere le strutture di gestione del SO (tabelle, ecc.)
- Schedulazione: necessità di evitare conflitti
- Sincronizzazione: mutua esclusione e ordinamento degli eventi
- Gestione della memoria condivisa
- Tolleranza ai guasti: in caso di "perdita di un processore" devono essere aggiornate le strutture di controllo del SO

# Stati dei Thread in Windows

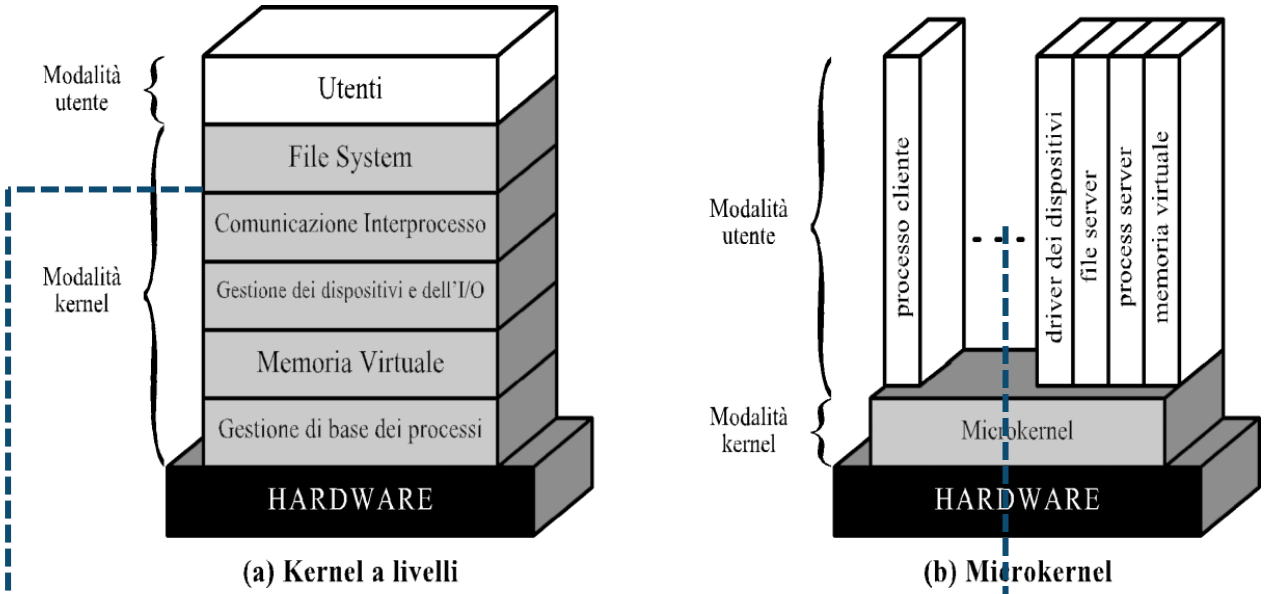



- **Ready**
- **Standby**: legato alla disponibilità del processore (SMP) richiesto per il thread. Se la priorità è sufficientemente alta il processo in running può essere interrotto.
- **Running**
- **Waiting**: I/O, attesa per sincronizzazione
- **Transition**: thread pronto per l'esecuzione ma le risorse non sono disponibili (es. lo stack può essere stato spostato su disco mentre era in waiting).
- **Terminated**

Supporto di SMP:

- I thread (inclusi quelli del kernel) possono essere eseguiti su ogni processore
- Il primo thread in ready viene assegnato al primo processore disponibile
- Thread appartenenti allo stesso processo possono essere eseguiti (contemporaneamente) su diversi processori
- Esecuzione di un thread sempre sullo stesso processore: dati ancora in cache..

# MicroKernel




Interazione solo tra strati adiacenti

La comunicazione avviene attraverso il MicroKernel che ridireziona i messaggi

Piccolo Nucleo del SO

Contiene le funzioni essenziali del SO

Servizi tradizionalmente inclusi nel SO sono sottosistemi esterni al microkernel ed eseguiti in modalità utente:

- Device drivers
- File systems
- Virtual memory manager
- Windowing system
- Security services

# Vantaggi del MicroKernel

- **Interfaccia uniforme:** I moduli usano le stesse interfacce per le richieste al microKernel
- **Estensibilità:** introduzione di nuovi servizi o modifiche non richiedono modifiche del microKernel
- **Flessibilità:** a seconda delle applicazioni certe caratteristiche possono essere ridotte o potenziate per soddisfare al meglio le richieste dei clienti. Es. Windows home – professional – ultimate
- **Portabilità:** Il cambio dell'hardware comporterà unicamente la modifica del microkernel.
- **Affidabilità:** lo sviluppo di piccole porzioni di codice ne permette una migliore ottimizzazione e test.
- **Supporto ai sistemi distribuiti:** ogni servizio è identificato da un numero nel microkernel e una richiesta da client non è necessario che sappia dove si trova il server in grado di soddisfare la stessa. Messaggistica gestita dal microkernel

# Design del MicroKernel

Il microkernel deve contenere:

- Le funzioni che dipendono direttamente dall'hardware (gestione degli interrupt e I/O)
- Le funzioni per la comunicazione tra processi (IPC)
- Gestione primitiva della memoria

Problema delle prestazioni per sistemi microkernel:

Costruire, inviare, accettare, decodificare un messaggio costa più che una chiamata a SO

Possibili soluzioni:

- Aggiungendo funzionalità al microkernel (MACH OS) si riduce il numero di cambiamenti di stato (utente/kernel) - Riduzione di flessibilità, interfacce minime…
- Ulteriore riduzione del microkernel

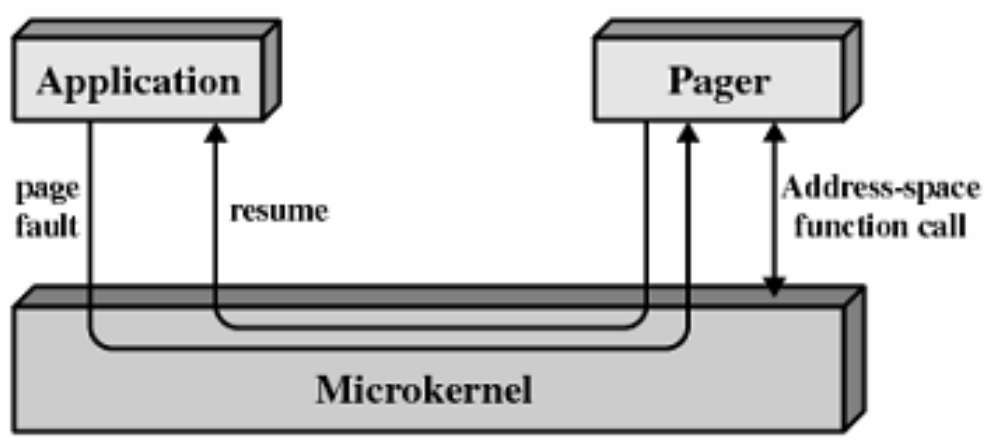
# Funzioni minime del MicroKernel

**Gestione primitiva della memoria**

Un modulo esterno al microkernel mappa pagine virtuali in pagine fisiche.

Il mapping è conservato in memoria principale.

- Una applicazione che accede ad una pagina che non si trova in memoria genera un page fault
- L'esecuzione passa al microkernel che invia un msg al paginatore comunicando la pagina richiesta
- La pagina viene caricata (paginatore e kernel collaborano per il mapping memoria reale-virtuale)
- Quando viene caricata la pagina il pager invia un msg all'applicazione



# Funzioni minime del MicroKernel

**Comunicazione tra processi**

Messaggio = (intestazione) + (corpo) + (puntatore+inform. di contr.)

- Intestazione (Mittente, Ricevente)
- Corpo (dati del messaggio)
- Puntatore (informazioni di controllo del processo, blocco dati)

Associata ad ogni processo c'è una PORTA: una capability list indica chi può inviare messaggi

Tale porta è amministrata dal Kernel

# Funzioni minime del MicroKernel

## Gestione degli Interrupt e dell'I/O

- Il microkernel riconosce gli interrupt ma non li gestisce direttamente.
- Il microkernel **trasforma l'interrupt in messaggio a livello utente**, che invia al processo che gestisce l'interrupt

```
driver thread:
do
wait(msg, mittente);
if mittente=mio_interrupt_hardware
then leggi/scrivi le porte di I/O;
azzera l'interrupt hardware
else….
Endif
enddo
```