8 - Thread, SMP e Microkernel

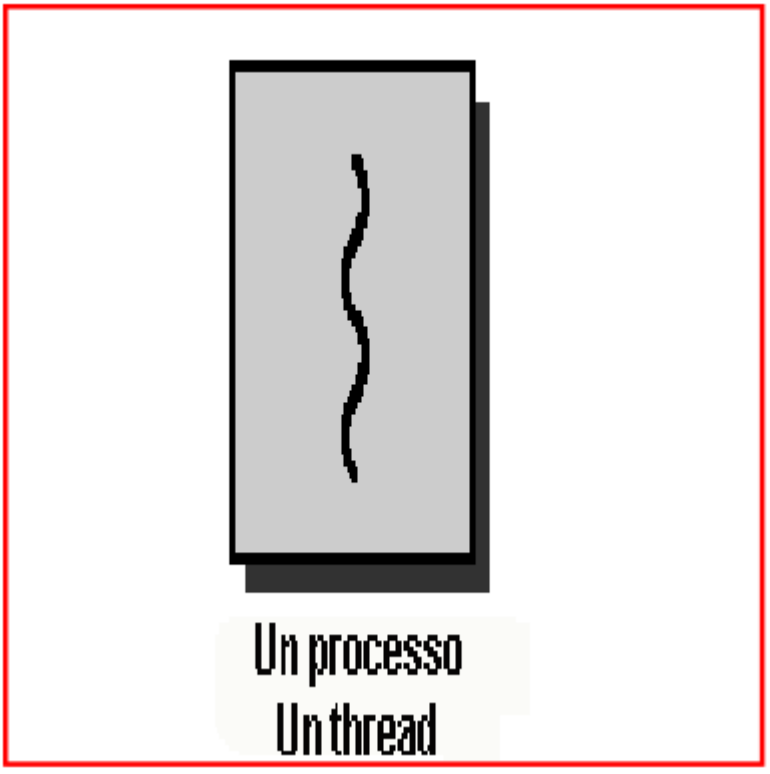
Thread

Un thread è una traccia di esecuzione di un programma rappresentata come una sequenza di valori del **Program Counter (PC)** che indica quali istruzioni devono essere eseguite.
Può essere visto come una versione "leggera" di un processo, chiamata anche **LWP (Light Weight Process)** e, a differenza di un processo completo,**non possiede tutte le risorse hardware proprie** (ad esempio uno spazio di memoria completamente separato) ma condividono le risorse dello stesso processo (come la memoria principale)
Pur avendo bisogno della CPU per essere eseguito, il thread **non "possiede" direttamente la CPU**, è lo stesso SO che assegna i thread alla CPU gestendo quando e quale thread può utilizzare il processore.

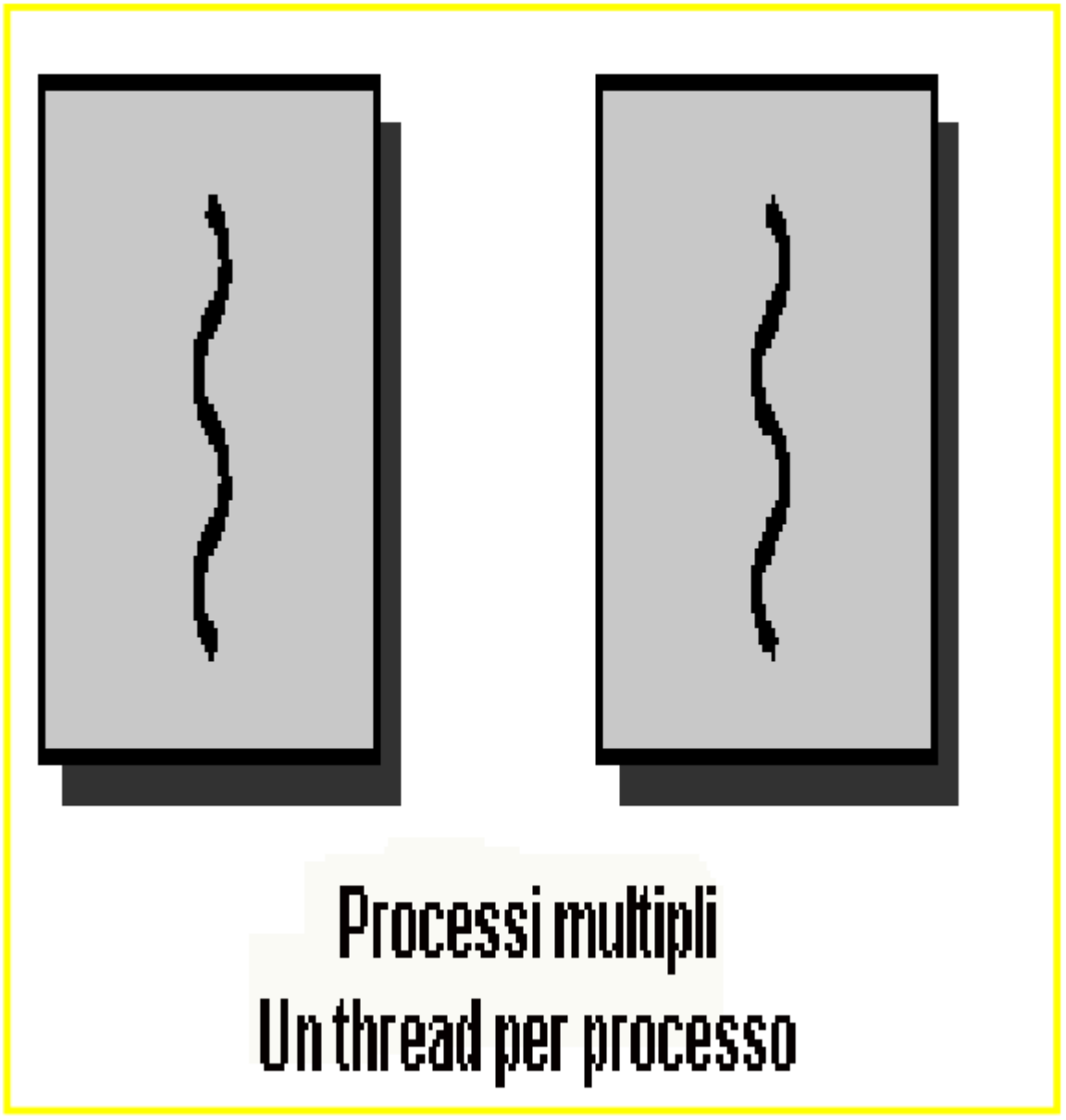
Multi-Threading

Il multi-threading è la capacità di un SO di supportare più thread (o flussi d'esecuzione) per ogni processo, eliminando di fatto la limitazione che i programmi avevano di poter unicamente eseguire sequenzialmente tutte le istruzioni.
Esistono diverse categorie di sistemi che supportano diverse configurazioni:

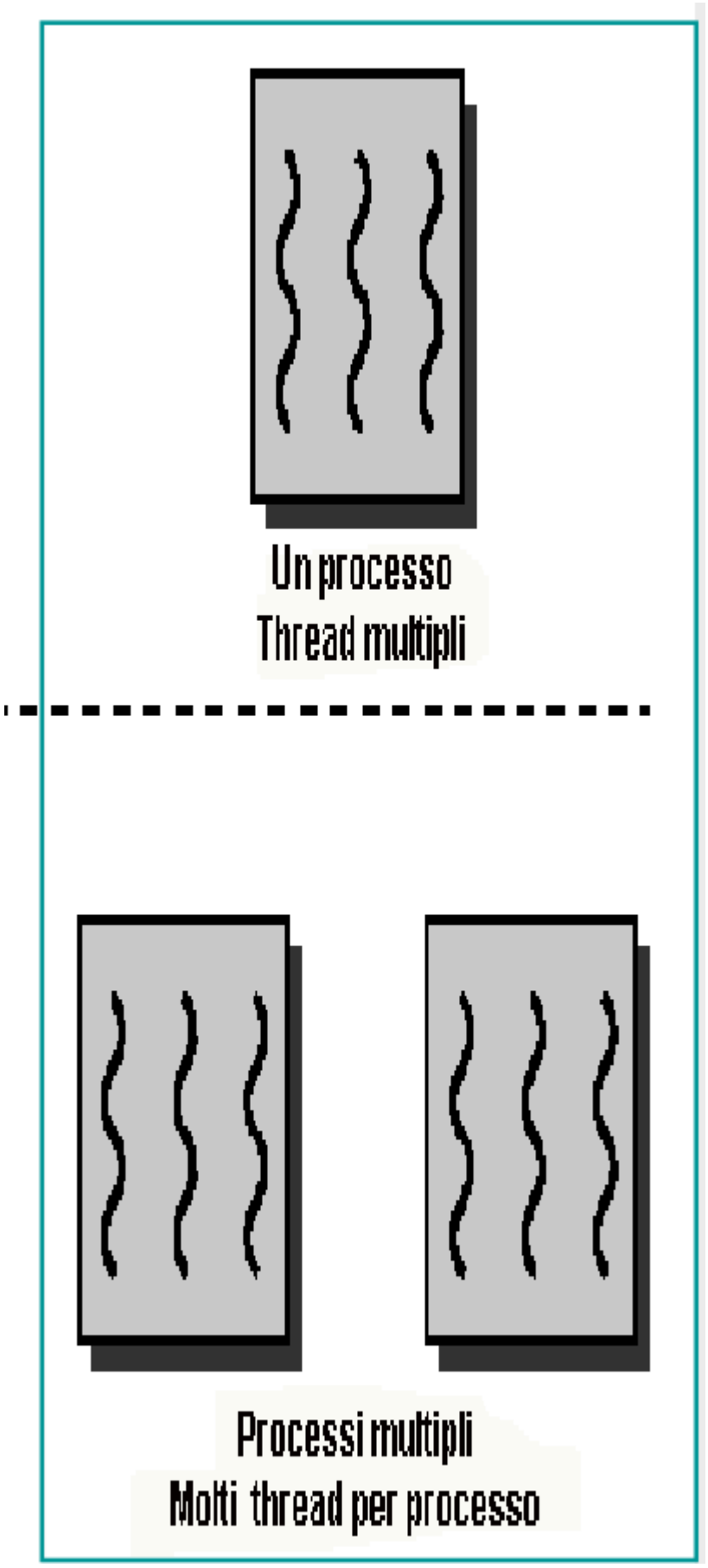
1. Legacy (MS-DOS), singolo processo con singolo thread



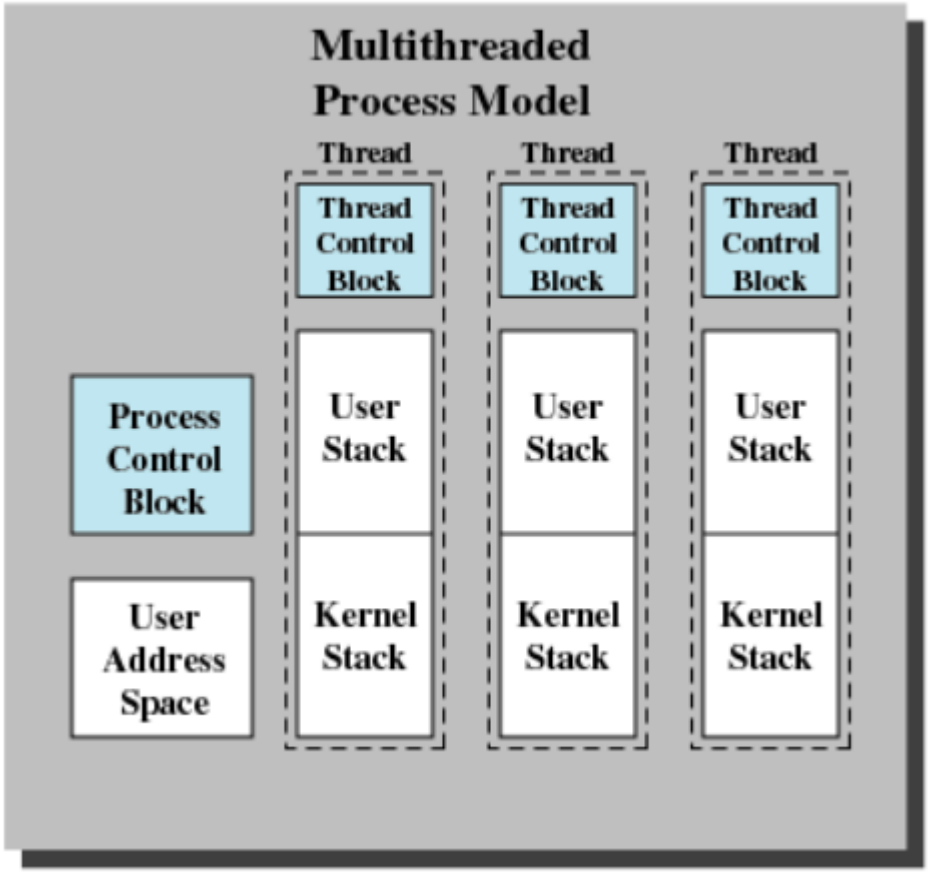
2. Processi multipli, singolo thread (Unix) :



3. Un processo, thread multipli e/o processi multipli, molti thread per processo (in sostanza più flussi d'esecuzione in contemporanea nello stesso processo)



Memoria e spazio del processo multi-thread



I thread sono entità che condividono stato e risorse del processo (genitore) a cui appartengono, risiedono nello stesso spazio di indirizzamento(User Address Space) ed accedono agli stessi dati (facilitando così la condivisione di informazioni).
Il thread è composto dal Thread Control Block, un blocco simile al PCB di un normale processo che comprende:

- **Identificatore del thread:** un ID unico (tid) assegnato a ogni nuovo thread
- **Puntatore allo stack** (kernel o utente)
- **Contatore del programma (PC):** punta all'istruzione corrente del programma del thread
- **Stato del thread:**
- **Valori dei registri del thread**
- **Puntatore al Process Control Block:** collegamento al PCB del processo a cui appartiene il thread

Oltre al TCB ogni thread possiede anche gli stack utente e kernel quando dovrà fare le chiamate a procedura.
Ma le istruzioni (o il codice) dove è presente effettivamente fisicamente? Nello spazio di indirizzamento (User Address Space), siccome il thread non può detenere questa risorsa per definizione

Vantaggi e difficoltà del thread

I thread portano, rispetto ai processi, una serie dei vantaggi:

1. Il tempo di creazione di un nuovo thread è **minore** del tempo di creazione di un nuovo processo

    - Un nuovo thread ha già la memoria allocata vivendo all'interno del genitore
    - Il processo richiede un nuovo feed, una struttura PCB, memoria necessaria con il controllo della disponibilità sufficiente, entrare poi nella fase a 7 stati...
2. Il tempo di terminazione di un thread è **minore** rispetto a quello di un processo
    - Il thread non ha risorse, il processo invece deve farle deallocare tutte
3. Il tempo necessario allo switch tra threads all'interno dello stesso processo è **minore** del tempo di switch tra processi
    - Questo switch è creato dall'interazione di un utente e il thread lo gestisce con l'ausilio di chi ha progettato l'applicativo, il SO non interviene
4. I threads all'interno di uno stesso processo condividono memoria e files, lo scambio dei dati tra di loro non richiede l'intervento dei kernel, ma con la necessità di sincronizzare le attività dei threads
5. I thread permettono di poter eseguire parallelamente più task (**parallelismo**)
6. Si introduce la possibilità di gestire più flussi contemporaneamente

I thread presentano comunque delle difficoltà legate al loro essere collegati ai processi:

- Se quest'ultimo viene sospeso si richiede che tutti i thread siano sospesi contemporaneamente perché si deve liberare spazio in memoria (ricordiamo che tutti i thread usano lo stesso spazio di memoria condivisa)
- La terminazione di un processo richiede che tutti i thread siano terminati

I thread poi non sono indipendenti, l'uno dipende dall'altro, visto che comunicano tra di loro al fine di concorrere all'operazione da eseguire (**concorrenza**)

Differenza tra parallelismo e concorrenza

Si noti la distinzione tra parallelismo e concorrenza:
Un sistema **concorrente** supporta più task permettendo a ciascuno di progredire nell’esecuzione, un sistema **parallelo** invece può eseguire simultaneamente più di un task, è dunque possibile avere concorrenza senza parallelismo.
Prima dell’avvento dei processori SMP e delle architetture multicore, la maggior parte dei sistemi era dotata di un singolo processore e gli scheduler della CPU erano progettati per fornire l’illusione di parallelismo mediante una rapida commutazione tra processi nel sistema, consentendo in tal modo a ogni processo di fare progressi.

Tali processi erano eseguiti in maniera concorrente, ma non in parallelo

Figura 4.3 Esecuzione concorrente su un sistema a singolo core.

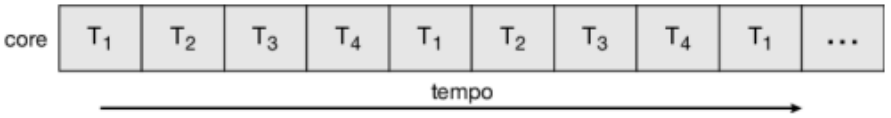
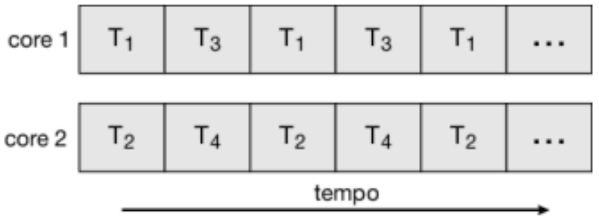


Figura 4.4 Esecuzione parallela su un sistema multicore.



Stato dei thread

I thread possono avere determinati stati:

- Ready
- Running
- Blocked

Lo stato Suspended non ha senso per un thread, è già presente a livello del processo (se un processo viene scaricato dalla memoria centrale lo stesso avviene per tutti i suoi threads)

Le operazioni base per il cambio di stato dei thread sono:

- Creazione (Un thread può creare altri thread)
- Blocco (attesa di un evento, viene salvato il contesto per il thread con PC, Stack Pointer e registri CPU)
- Sblocco (modifica dello stato del TCB da Blocked a Ready, il thread viene accodato a quelli in attesa del processore)
- Terminazione (deallocazione del contesto registri e stack)

Il blocco di un thread può bloccare l'intero processo solo nei sistemi con **User-Level Threads (ULT)**, perché il kernel gestisce il processo come un'unica entità e non distingue i thread al suo interno.
Nei sistemi con **Kernel-Level Threads (KLT)**, invece, solo il thread bloccato viene messo in attesa, mentre gli altri thread dello stesso processo possono continuare a essere eseguiti grazie alla gestione separata dei thread da parte del kernel.

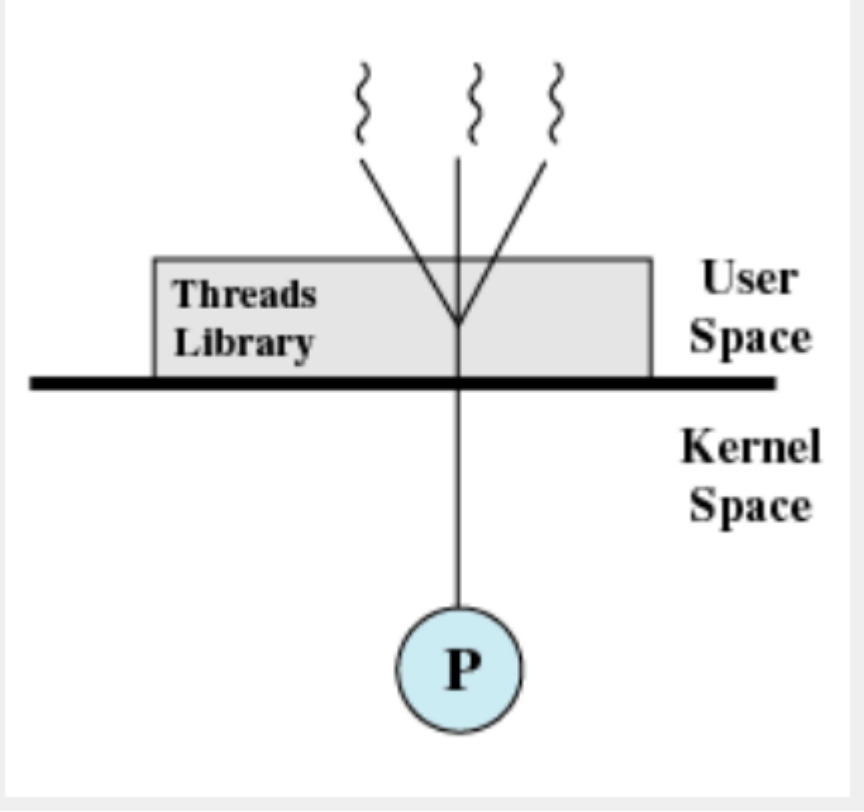
Categorie di thread

Ci sono due categorie di thread:

1. **ULT (User Level Thread)**: creati e gestiti dall'utente tramite una libreria senza l'intervento del kernel e trasparenti ad esso
   - Se il kernel è a singolo thread il blocco del thread di livello utente blocca l'intero processo (Il SO continua a schedulare processi)
2. **KLT (Kernel Level Thread)**: Il kernel si occupa della creazione, scheduling e gestione, questi thread possono essere eseguiti su diversi processori ma la loro gestione è più lenta degli ULT

User Level Thread

L'ULT usa il modello molti a uno, fa corrispondere molti thread a livello utente a un singolo thread a livello kernel.
La gestione dei thread risulta efficiente perché viene effettuata da una libreria di thread nello spazio utente, tuttavia l'intero processo rimane bloccato se un thread invoca una chiamata di sistema di tipo bloccante (come quella di I/O)



La libreria, **presente nello spazio utente**, permette di:

- Creare e distruggere threads
- Scambiare messaggi tra loro
- Schedulazione
- Salvataggio e caricamento dei contesti dei threads
  Tutte queste attività vengono svolte all'interno del singolo processo utente, il kernel quindi continua a schedulare i processi come unità a se stanti e il SO vede un unico processo, non sapendo dell'esistenza di più thread (trasparenza)

I **vantaggi** dell'ULT sono:

- Risparmio di sovraccarico: il cambio del thread avviene all'interno dello spazio di indirizzamento utente, non viene richiesto l'intervento del kernel (non esiste praticamente il context switching)
- Schedulazione diversa per ogni applicazione: ogni applicazione può essere ottimizzata per se (sempre ottimizzata dal programmatore o dal progettista)
- Universale: può essere eseguito a livello di qualsiasi sistema operativo, poiché la libreria è presente nello spazio utente e non nel kernel

Tuttavia, presenta anche **svantaggi**:

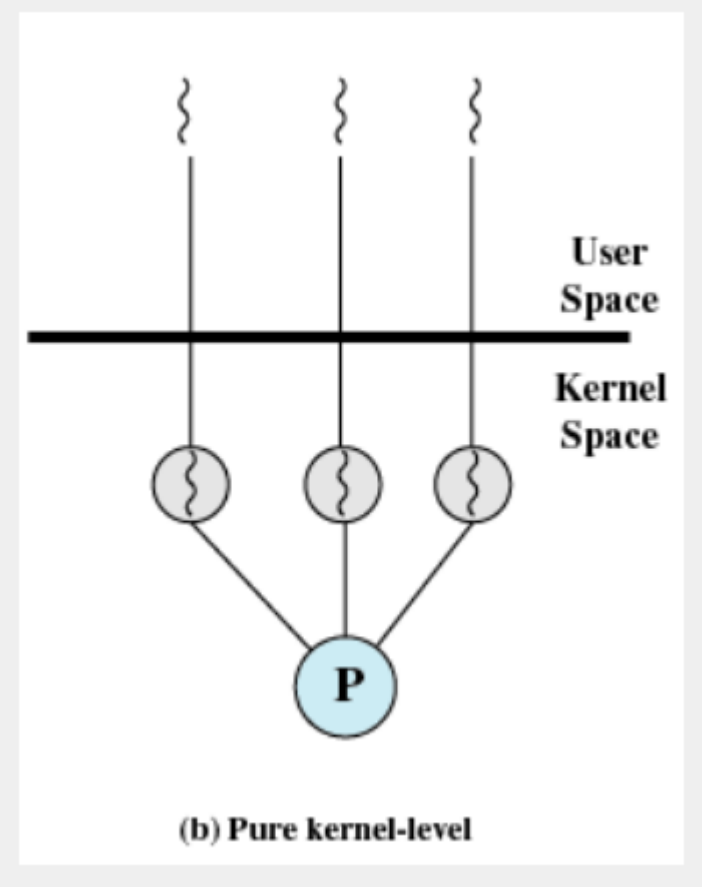
- La chiamata a sistema da parte di un thread blocca tutti i thread a processo (esempio: il thread decide di fare una richiesta di I/O ma deve aspettare perché la risorsa è occupata)
- Il kernel assegna un processo ad un singolo processore, eliminando il **multiprocessing** a livello di thread (thread dello stesso processo su più processori)

Alcune soluzioni (parziali) a questi problemi possono essere:

1. Sviluppo di un applicazione a livello di processi (si perdono i vantaggi dei thread)
2. Jacketing: conversione di una chiamata bloccante in una non bloccante
    - Se si verifica una chiamata del genere si crea un nuovo processo figlio dove si fa l'injection della richiesta bloccante, si chiede l'esecuzione del figlio facendolo rimanere bloccato con la richiesta, il padre (e i suoi threads) vanno avanti

Kernel Level Thread Puro

Il KLT usa il modello uno a uno, cioè mette in corrispondenza ciascun thread a livello utente con un thread a livello kernel.
Questo modello offre un grado di concorrenza maggiore rispetto al precedente, poiché anche se un thread invoca una chiamata di sistema bloccante, è possibile eseguire un altro thread; il modello permette anche l’esecuzione dei thread in parallelo nei sistemi multiprocessore.
L’unico svantaggio di questo modello è che la creazione di ogni thread a livello utente comporta la creazione del corrispondente thread a livello kernel



A livello utente un API consente l'accesso alla parte del kernel che gestisce i thread (ogni API è specifica per S.O.)
Il kernel mantiene info su:

1. Contesto del processo
2. Contesto dei threads
3. Scambio messaggi tra threads
   La schedulazione viene effettuata a livello di threads:

- Se un thread di un P è bloccato, un altro thread dello stesso processo può essere eseguito
- Thread di uno stesso processo possono essere schedulati su diversi processi (1 processo su un core, 1 su un altro)

KLT ha comunque un overhead sul trasferimento del controllo da un thread ad un altro richiede l'intervento del kernel

Approcci MISTI

E' il modello **molti a molti**, dove più thread di livello utente sono in **corrispondenza con più thread** di livello kernel, dunque più thread di un singolo processo sono eseguiti contemporaneamente su più **processori** ed una chiamata **bloccante** non blocca totalmente l'intero processo, ma solamente quel thread che gestisce quella micro istruzione.
Per mantenere un numero alto di kernel thread allocati all'applicazione serve una comunicazione tra il kernel e la libreria dei thread, in questo caso la LWP viene usata come

una struttura intermedia e **virtuale** alla libreria dei thread per poter **schedulare** l'esecuzione. In un applicazione CPU-bound su un sistema monoprocessore utilizza un solo thread per volta per essere eseguito, essa necessita quindi di un solo LWP per thread a differenza di un applicazione I/O-bound che richiede un LWP per ciascuna chiamata di sistema.

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, Solaris, OS/2, OS/390, MACH
1:M	*Ambienti distribuiti: threads possono spostarsi tra più calcolatori* Un thread può spostarsi da un processo all'altro; ciò permette di spostare facilmente i thread fra sistemi diversi.	Ra(Clouds), Emerald
M:M	Combina le proprietà degli approcci M:1 e 1:M.	TRIX

Calcolatori attuali

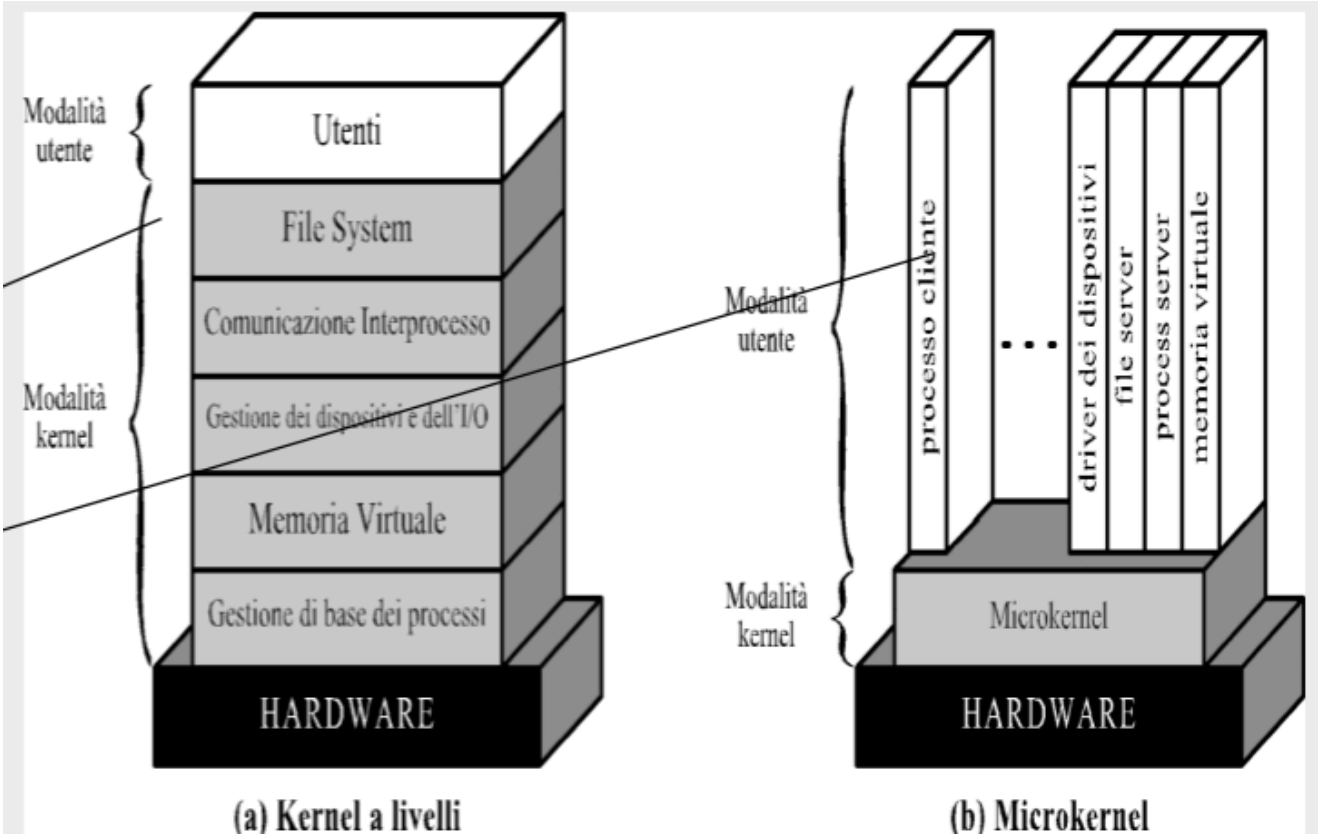
Gli attuali calcolatori sono definiti **Symmetric Multi Processing** (SMP), essi sono disposti di un calcolatore con molti processori, questi ultimi condividono le stesse risorse e possono effettuare tutti le stesse funzioni.
Ogni processore esegue la stessa copia del SO e tutti gestiscono la schedulazione dei processi e thread a loro disposizione.
Come ogni architettura essi non sono perfetti ma presentano alcune difficoltà, come:

- I processori non devono schedulare lo stesso processo
- La possibilità di effettuare accessi simultanei alla memoria va gestito avendo una memoria condivisa, ciò porterebbe a degli errori nel caso una risorsa è modificata dai processori simultaneamente
- Necessità di coerenza nella cache a livello hardware

MicroKernel

Nei primi sistemi operativi tutte le funzioni del Kernel erano memorizzate in memoria **centrale**, per cui si pensò, per non occupare troppa memoria, di chiamare le funzioni minime del kernel solo quando sono necessarie, questo insieme di funzioni prese nome di

MicroKernel:



Come si evince, i moduli (come la Memoria Virtuale) non sono più presenti nella memoria, ma invocati nella memoria solo nel caso di estrema necessità di utilizzo di quel modulo, i servizi che nel modello **Kernel a livelli** erano inclusi nel SO ora sono esterni al MicroKernel ed eseguiti in modalità **utente**, sono al pari dei processi utente.
Nel vecchio sistema a livelli, i moduli del Kernel potevano interagire solo tra **strati** adiacenti, diversamente la comunicazione tra moduli nel sistema a MicroKernel è gestita proprio dal MicroKernel stesso che ridireziona i messaggi.

Vantaggi derivanti dal MicroKernel

- **Interfaccia uniforme**: Nello schema a livelli tutti i livelli usavano un linguaggio comunicativo differente, rendendo difficile e complessa la comunicazione tra di loro, il MicroKernel invece decide un **unica interfaccia di comunicazione**, dato che tutti comunicano solo con quest'ultimo.
  Nei moderni linguaggi di programmazione questo principio prende il nome di **polimorfismo**.
- **Estensibilità**: Possibilità di introdurre nuovi servizi senza **modificare** interamente il MicroKernel, rispettando solamente l'interfaccia stabilita.
- **Flessibilità**: A seconda delle applicazioni e dell'uso che necessita l'utente, alcune caratteristiche possono essere **ridotte o potenziate**. Per esempio Windows utilizza la versione *home* e la versione *pro* del suo SO, la prima ha funzionalità ridotte rispetto all'altra.
  Nei comuni linguaggi di programmazione questo vantaggio si manifesta sottoforma delle **architetture a micro-servizi**, tipiche delle moderne app.

- **Portabilità**: La possibilità di spostare tutti i moduli senza preoccuparci di ricrearli nel caso in cui dobbiamo effettuare un cambio da una macchina ad un altra, bisognerà solamente modificare l'interfaccia del MicroKernel.
- **Affidabilità**: Capacità di scovare meglio gli errori presenti nei **micro-codici** (piccole porzioni di codice) e portarne ad una migliore ottimizzazione.
- **Supporto ai Sistemi distribuiti**: La messaggistica è gestita dal MicroKernel, il client non necessita di sapere dove si trova il server che gli soddisferà la richiesta.

Cosa contiene il MicroKernel?

Il MicroKernel deve contenere:

- Le funzioni che dipendono dall'**hardware**
- Le funzioni per la comunicazione tra processi (**IPC**)
- Le funzioni per gestire la **memoria primitiva**: la memoria virtuale è un sistema **sofisticato** e per questo non se ne occuperà il MicroKernel, la memoria primitiva fa riferimento alla memoria RAM, essa si chiama primitiva poiché deve eseguire solo le azioni primitive.
  Il MicroKernel gestisce solo la RAM poiché mantiene solo le operazioni più importanti, delegando le funzionalità avanzate e astratte ai moduli.

Il MicroKernel non è perfetto:
Ogni richiesta ad un servizio genera una batteria di messaggi che dovrà essere scambiata e ciò comporta che le azioni di costruzione, invio, accettazione e decodifica per un messaggio ha un **costo maggiore** di tempo rispetto ad una chiamata diretta al SO come nel sistema a livelli.
Le possibili soluzioni implementate sono state:

- La riduzione maggiore del MicroKernel
- Aggiunta di funzionalità al MicroKernel in modo da ridurre il numero di cambiamenti tra stato utente e stato kernel

Gestione primitiva della memoria

Un modulo esterno al MicroKernel mappa pagine virtuali in pagine fisiche, è una tecnica chiamata **mapping** e queste informazioni sono conservate nella memoria centrale.
Nel momento in cui l'applicazione chiede un dato che non si trova in memoria RAM, genera un **page fault**, ovvero un errore simile al miss della cache, siccome questo dato è mancante è doveroso andare a recuperarlo, il MicroKernel si occupa di segnalare la mancanza di questo dato con un **messaggio** al paginatore, riferendo la pagina richiesta.
La pagina verrà caricata dal paginatore utilizzando il MicroKernel e attraverso la comunicazione tra questi due, il paginatore avviserà **l'applicazione utente**, tramite un messaggio, che la pagina richiesta è stata caricata.
In un sistema a MicroKernel questa operazione è rapida ed eseguibile in pochi passaggi,

differentemente in un sistema Kernel a livelli, dove i moduli comunicano solo coi loro corrispettivi adiacenti, il tempo sarebbe stato eccessivamente più lungo, dato che ogni richiesta sarebbe dovuta partire da un modulo, passare per tutti gli altri ed arrivare alla **modalità utente**, per poi riscendere fino al livello hardware per risolvere il problema e comunicarlo nuovamente all'utente, ripetendo nuovamente il giro.

Comunicazione tra processi

I messaggi nei SO, sono composti da tre parti:

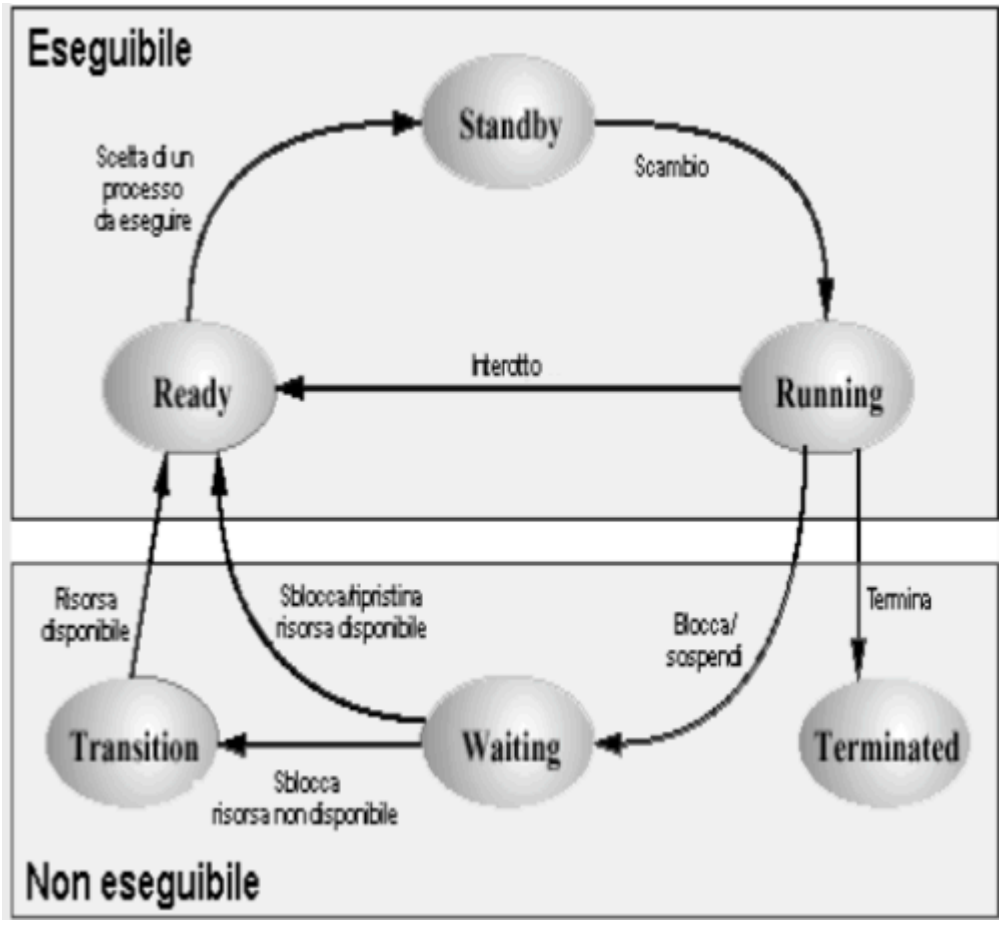
- **Head**(intestazione): contiene **l'indirizzo del mittente** e quello del **destinatario**
- **Payload** (corpo): contiene il **contenuto** del messaggio
- **Tail** (coda): sono le **informazioni di controllo**, come i puntatori o il codice di Hamming
  È importante specificare che ad ogni processo è associa una **porta**, ogni messaggio viene inviato ad una determinata porta e ogni porta detiene un proprio **servizio**.

Gestione Interrupt e I/O

Abbiamo visto che uno dei moduli del MicroKernel è dedicato all'hardware, tra cui quindi gli interrupt, quest'ultimo **riconosce** gli interrupt ma non li gestisce direttamente per mantenere la struttura del Kernel più snella.

Il **micro-servizio driver** del dispositivo è in **ascolto** sulla porta del mittente, ovvero il **messaggio** che invierà il MicroKernel a **livello utente** dopo aver riconosciuto un interrupt: se il messaggio corrisponde al processo del driver, quest'ultimo gestirà il suo interrupt, che sarà inviato in modalità **broadcast** a tutti i driver collegati.
La modalità broadcast significa l'invio **simultaneo** dei messaggi a tutti i dispositivi/periferiche ma sarà accettato solo da coloro ca cui è effettivamente destinato.

Stati dei Thread in Windows



- Il primo stato è quello di **Ready**, dove il thread è pronto per essere eseguito.
- **Standby** è un nuovo stato in cui transita il thread prima di andare in Running. Questo stato si occupa di inserire in **lista di attesa** il thread che sta facendo richiesta ad uno specifico core del *SMP*, se la priorità è elevata verrà portato in esecuzione, interrompendo il processo che stava in esecuzione in quel preciso momento.
  Il primo thread nello stato di Ready viene assegnato al **primo processore disponibile** non ancora utilizzato, successivamente i dati di quel thread saranno memorizzati nella **cash** (L2,L3) dato che quel thread in futuro potrebbe richiedere il medesimo processore.
- Dopo sarà portato in **Running**.
- Dopo la fase di Running potrebbe accadere di andare nella fase di **Waiting**, in questo stato vengono eseguite le operazioni di I/O per poter poi continuare l'esecuzione del thread.
- Come sappiamo al termine dell'esecuzione il thread termina finendo nella fase di **Terminated**.
- Dalla fase di Waiting non è certo che si ritorni in Ready, ma potrebbe finire in **Transition**, esso si comporta nella stessa maniera della fase di *Suspended* in cui si necessita di risorse non ancora disponibili e il processo quindi sarà spostato sull'area di swap.

Anche se possono essere eseguiti su ogni processore, grazie al supporto di SMP, è opportuno che ogni thread sia sempre **schedulato sullo stesso core**;

Inoltre i thread appartenenti allo stesso processo possono essere eseguiti su **diversi processi contemporaneamente**.