Introduzione ai thread

Processi leggeri

- **Immagine di un processo** (codice, variabili locali e globali, stack, descrittore)
- **Risorse possedute:** (file aperti, processi figli, dispositivi di I/O..),
- L’immagine di un processo e le risorse da esso possedute costituiscono il suo **spazio di indirizzamento.**
- La locazione dello spazio di indirizzamento dipende dalla tecnica di **gestione della memoria** adottata. Potrà essere contenuto in tutto o solo in parte nella memoria principale (registri base ed indice, impaginazione, segmentazione).

Processi (pesanti) e thread

- **Proprietà dei *processi***: *spazi di indirizzamento distinti*, cioe` non condividono memoria (es.**Unix**)
- **Complessità delle operazioni di *cambio di contesto*** tra due processi: comportano il salvataggio ed il ripristino dello spazio di indirizzamento **(overhead).** Analogamente per le operazioni di creazione e terminazione di un processo.
- Utilizzo del modello a **scambio di messaggi** (es.Unix)

Processi & thread

- Il concetto di processo e` basato su **due aspetti indipendenti:**
  - **Possesso delle risorse** contenute nel suo spazio di indirizzamento.
  - **Esecuzione**. Flusso di esecuzione, all'interno di uno o più programmi, che condivide la CPU con altri flussi, possiede uno stato e viene messo in esecuzione sulla base della politica di scheduling.

- I due aspetti sono indipendenti e possono essere gestiti separatamente dal S.O.:
  - processo **leggero** (***thread***): elemento cui viene assegnata la CPU
  - processo **pesante** (***processo o task***): elemento che possiede le risorse

- Un *thread* rappresenta un ***flusso di esecuzione*** all'interno di un *processo pesante*.

- **Multithreading**: molteplicità di flussi di esecuzione all'interno di un processo pesante.

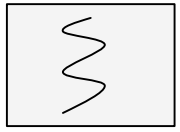
- Tutti i thread definiti in un processo ***condividono*** le risorse del processo, risiedono nello **stesso spazio di indirizzamento** ed hanno **accesso a dati comuni**.

Ogni thread ha:

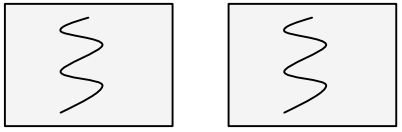
- uno **stato** di esecuzione (runnig, ready, blocked)
- un **contesto** che è salvato quando il thread non è in esecuzione
- uno **stack** di esecuzione
- uno spazio di memoria privato per le **variabili locali**
- accesso alla memoria e alle risorse del task condiviso con gli altri thread.

Vantaggi

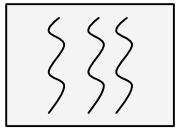
- maggiore efficienza: le operazioni di context switch, creazione etc., sono più economiche rispetto ai processi.
- maggiori possibilità di utilizzo di architetture multiprocessore.



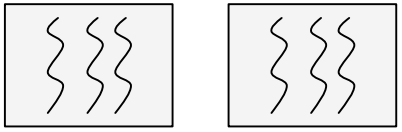
un thread per processo



Processi multipli: un thread per processo



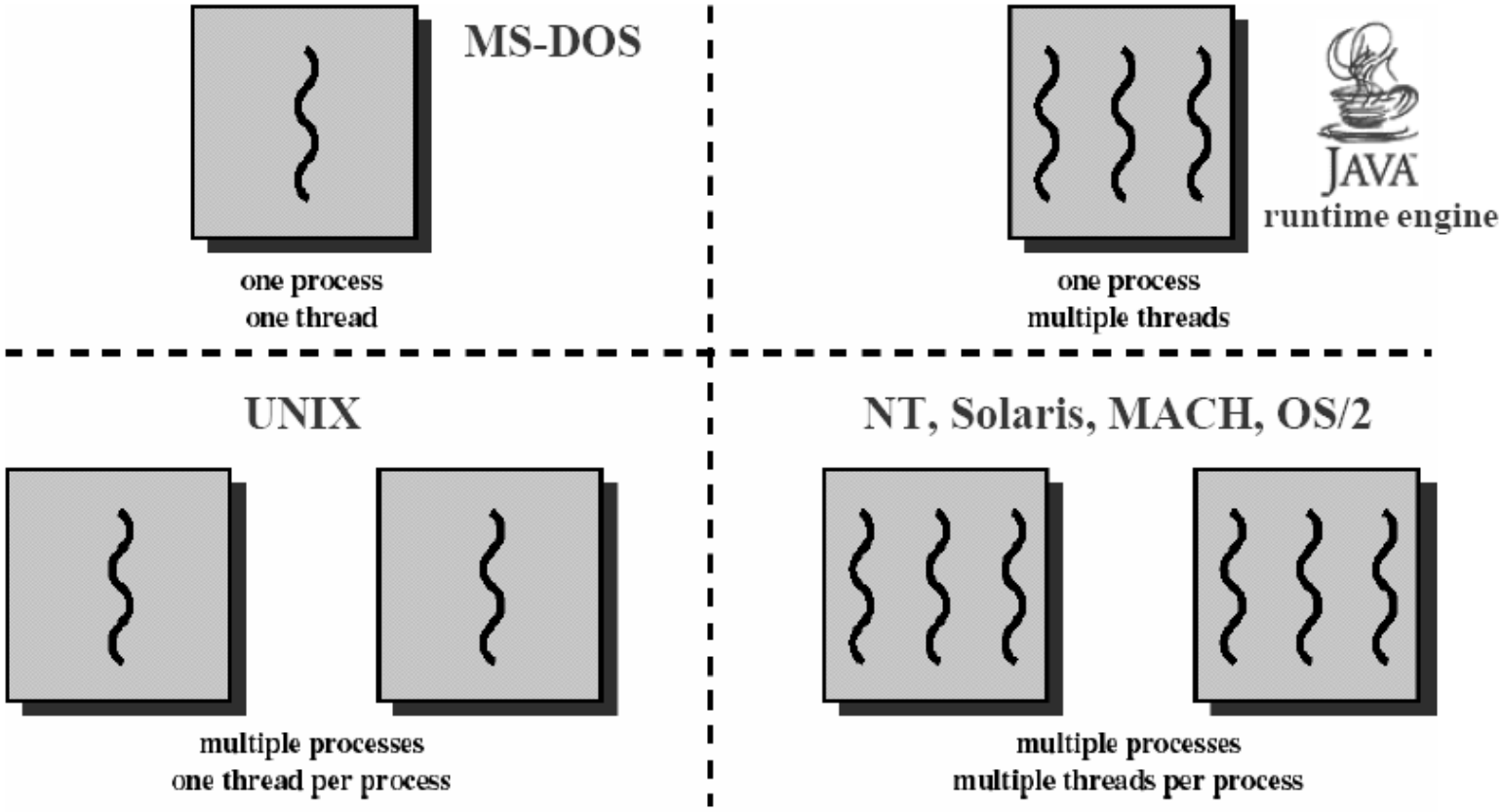
più thread per processo



Processi multipli: più thread per processo

Soluzioni adottate

- **MS-DOS:** un solo processo utente ed un solo thread.
- **UNIX**: più processi utente ciascuno con un solo thread.
- **Supporto run time di Java**: un solo processo, più thread.
- **Linux, Windows NT, Solaris**: più processi utente ciascuno con più thread.


MS-DOS
one process
one thread
JAVA
runtime engine
one process
multiple threads
UNIX
multiple processes
one thread per process
NT, Solaris, MACH, OS/2
multiple processes
multiple threads per process

- I thread possono eseguire ***parti diverse*** di una stessa applicazione.

Esempio: **web browser**

  - *thread* che scrive il testo sul video
  - *thread* che ricerca dati sulla rete

Esempio: **word processing**

  - thread che mostra i grafici sul video
  - thread che legge i comandi inviati dall'utente
  - thread che evidenzia gli errori di scrittura

- I thread possono eseguire le ***stesse funzioni*** (o funzioni simili) di un'applicazione.

Esempio: **web server**

  - *thread* che accetta le richieste e crea altri *thread* per servirle.
  - *thread* che servono la richiesta (possono essere diversi a seconda del tipo di richiesta).

Realizzazione dei thread

A livello utente (es. ***Java***)

- ***Hp:*** sistema operativo multitasking
- Libreria di funzioni (***thread package***) che opera a livello utente e fornisce il supporto per la creazione, terminazione, sincronizzazione dei thread e per la scelta di quale thread mettere in esecuzione (*scheduling*).
- Il sistema operativo ***ignora la presenza dei thread*** continuando a gestire solo i processi.
- Quando un processo è in esecuzione parte con un solo thread che può creare nuovi thread chiamando una apposita funzione di libreria.

- Gerarchia di thread o thread tutti allo stesso livello.
- La soluzione è *efficiente* (tempo di switch tra thread), *flessibile* (possibilità di modificare l'algoritmo di scheduling), *scalabile* (modifica semplice del numero di thread).
- Se un thread *si blocca* in seguito ad una **chiamata ad una *funzione del package*** (es. wait), va in esecuzione un altro thread dello stesso processo.
- I thread possono **chiamare delle *system call*** (es. I/O): intervento del sistema operativo che ***blocca il processo*** e conseguentemente l'esecuzione di *tutti i suoi thread*.
- Il S.O. interviene anche nel caso allo ***scadere del quanto di tempo*** assegnato ad un processo (sistemi time sharing).
- Non è possibile sfruttare il parallelismo proprio di *architetture multiprocessore*: un processo (con tutti i suoi thread) è assegnato ad uno dei processori.

Realizzazione di thread

***A livello di nucleo* (es.** *NT*, *Linux*)**:**

- Il S.O. si fa carico di tutte le **funzioni per la gestione dei thread**. Mantiene tutti i descrittori dei thread (oltre a quelli dei processi).
- A ciascuna funzione corrisponde una ***system call***.
- Quando un thread si *blocca* il S.O. può mettere in esecuzione un altro thread dello *stesso processo*.
- Soluzione ***meno efficiente*** della precedente.
- Possibilità di eseguire thread diversi appartenenti allo stesso processo su unità di elaborazione differenti (*architettura multiprocessore).*

Realizzazione di thread

Soluzione mista (es. Solaris):

- Creazione di thread, politiche di assegnazione della CPU e sincronizzazione a *livello utente.*
- I thread a livello utente sono mappati in un numero (minore o uguale) di thread a livello nucleo.

Vantaggi:

- Thread della stessa applicazione possono essere eseguiti in parallelo su processori diversi.
- Una chiamata di sistema bloccante non blocca necessariamente lo stesso processo