

Threads

Dai processi...

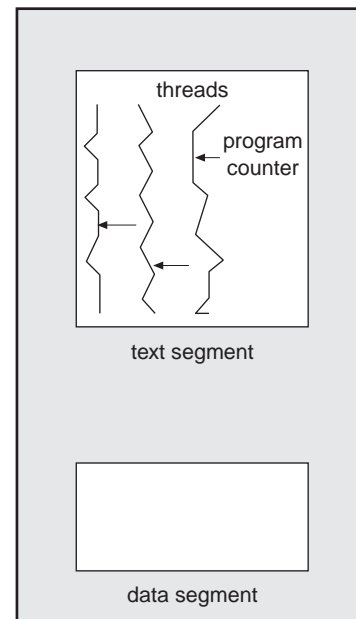
I processi finora studiati incorporano due caratteristiche:

- Unità di allocazione risorse: codice eseguibile, dati allocati staticamente (variabili globali) ed esplicitamente (heap), risorse mantenute dal kernel (file, I/O, workind dir), controlli di accesso ...
- Unità di esecuzione: un percorso di esecuzione attraverso uno o più programmi: stack di attivazione (variabili locali), stato (running, ready, waiting,...), priorità, parametri di scheduling,...

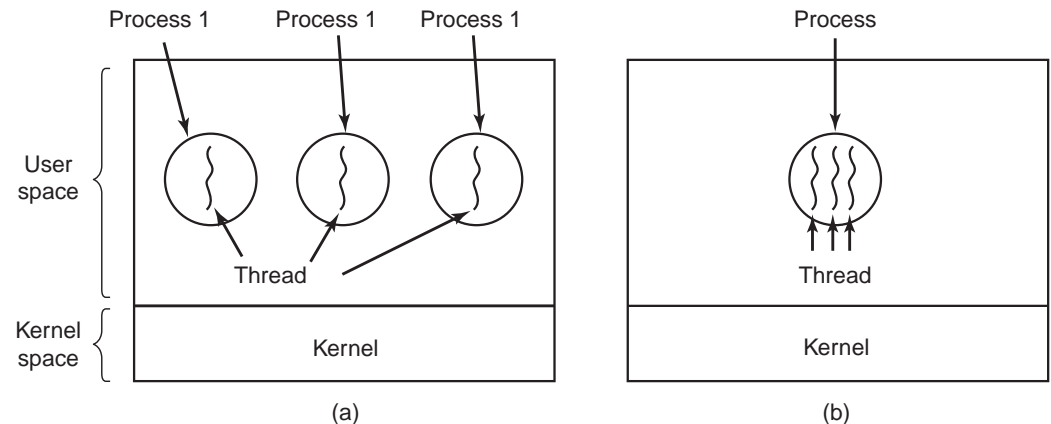
Queste due componenti sono in realtà *indipendenti*

... ai thread

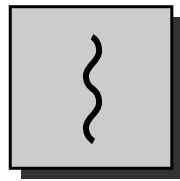
- Un *thread* (o *processo leggero*, *lightweight process*) è una unità di esecuzione:
 - program counter, insieme registri
 - stack del processore
 - stato di esecuzione
- Un thread condivide con i thread suoi pari ^{task} una unità di allocazione risorse:
 - il codice eseguibile
 - i dati
 - le risorse richieste al sistema operativo
- un *task* = una unità di risorse + i thread che vi accedono



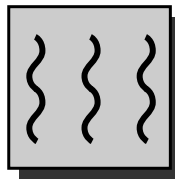
Esempi di thread



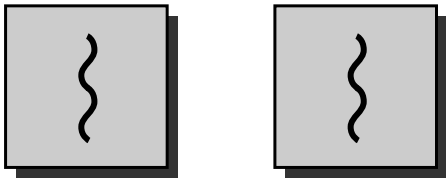
Processi e Thread: quattro possibili scenari



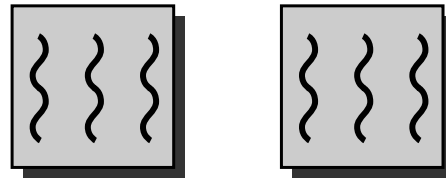
one process
one thread



one process
multiple threads



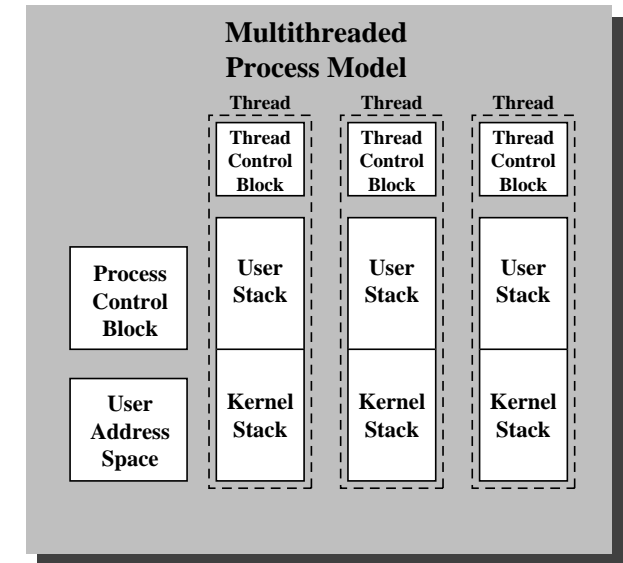
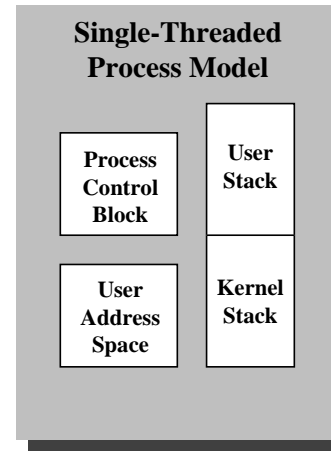
multiple processes
one thread per process



multiple processes
multiple threads per process

5

Modello multithread dei processi



6

Risorse condivise e private dei thread

Tutti i thread di un processo accedono alle stesse risorse condivise

Per process items

Address space
Global variables
Open files
Child processes
Pending alarms
Signals and signal handlers
Accounting information

Per thread items

Program counter
Registers
Stack
State

7

Condivisione di risorse tra i thread

- Vantaggi: maggiore efficienza
 - Creare e cancellare thread è più veloce (100–1000 volte): meno informazione da duplicare/creare/cancellare (e a volte non serve la system call)
 - Lo scheduling tra thread dello stesso processo è molto più veloce che tra processi
 - Cooperazione di più thread nello stesso task porta maggiore throughput e performance
(es: in un file server multithread, mentre un thread è bloccato in attesa di I/O, un secondo thread può essere in esecuzione e servire un altro client)

8

Condivisione di risorse tra thread (Cont.)

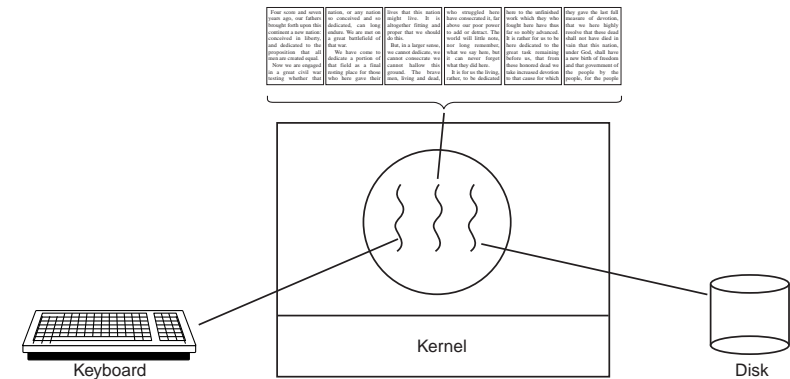
- Svantaggi:
 - Maggiore complessità di progettazione e programmazione
 - * i processi devono essere “pensati” paralleli
 - * minore information hiding
 - * sincronizzazione tra i thread
 - * gestione dello scheduling tra i thread può essere demandato all'utente
 - Inadatto per situazioni in cui i dati devono essere protetti
- Ottimi per processi cooperanti che devono condividere strutture dati o comunicare (e.g., produttore–consumatore, server, ...): la comunicazione non coinvolge il kernel

9

Esempi di applicazioni multithread

Lavoro foreground/background: mentre un thread gestisce l'I/O con l'utente, altri thread operano sui dati in background. Spreadsheets (ricalcolo automatico), word processor (reimpaginazione, controllo ortografico, ...)

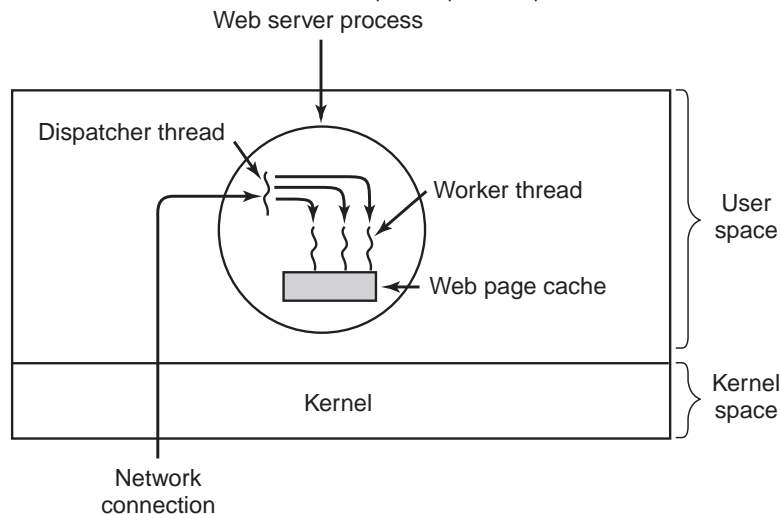
Elaborazione asincrona: operazioni asincrone possono essere implementate come thread. Es: salvataggio automatico.



10

Esempi di applicazioni multithread (cont.)

Task intrinsecamente paralleli: vengono implementati ed eseguiti più efficientemente con i thread. Es: file/http/dbms/ftp server, ...



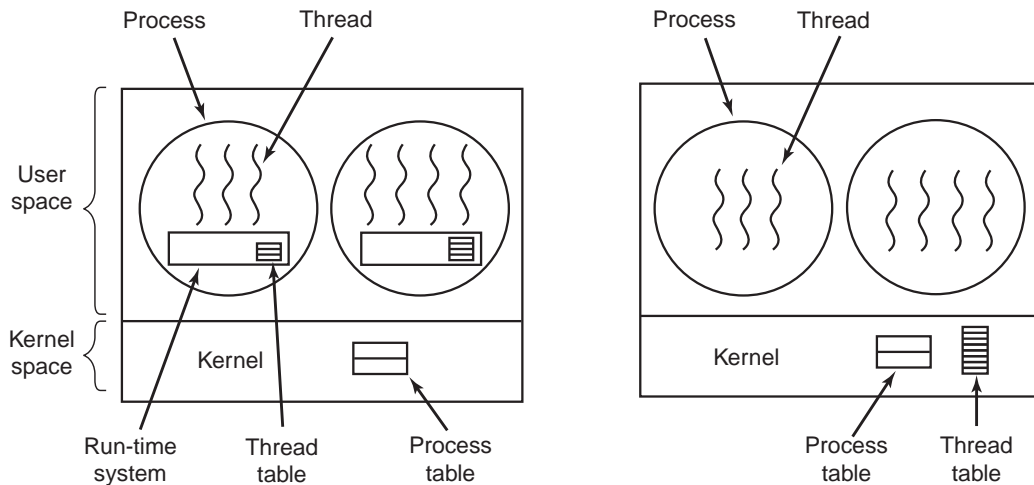
11

Stati e operazioni sui thread

- Stati: *running*, *ready*, *blocked*. Non ha senso “swapped” o “suspended”
- Operazioni sui thread:
 - creazione (spawn):** un nuovo thread viene creato all'interno di un processo (`thread_create`), con un proprio punto d'inizio, stack, ...
 - blocco:** un thread si ferma, e l'esecuzione passa ad un altro thread/processo. Può essere volontario (`thread_yield`) o su richiesta di un evento;
 - sblocco:** quando avviene l'evento, il thread passa dallo stato “blocked” al “ready”
 - cancellazione:** il thread chiede di essere cancellato (`thread_exit`); il suo stack e le copie dei registri vengono deallocati.
- Meccanismi per la sincronizzazione tra i thread (semafori, `thread_wait`): indispensabili per l'accesso concorrente ai dati in comune

12

Implementazioni dei thread: Livello utente vs Livello Kernel



13

User Level Thread

User-level thread (ULT): stack, program counter, e operazioni su thread sono implementati in librerie a livello utente.

Vantaggi:

- efficiente: non c'è il costo della system call
- semplici da implementare su sistemi preesistenti
- portabile: possono soddisfare lo standard POSIX 1003.1c (pthread)
- lo scheduling può essere studiato specificatamente per l'applicazione

14

User Level Thread (Cont.)

Svantaggi:

- non c'è scheduling automatico tra i thread
 - non c'è prelazione dei thread: se un thread non passa il controllo esplicitamente monopolizza la CPU (all'interno del processo)
 - system call bloccanti bloccano tutti i thread del processo: devono essere sostituite con delle routine di libreria, che blocchino solo il thread se i dati non sono pronti (*jacketing*).
- L'accesso al kernel è sequenziale
- Non sfrutta sistemi multiprocessore
- Poco utile per processi I/O bound, come file server

Esempi: thread CMU, Mac OS ≤ 9 , alcune implementazioni dei thread POSIX

15

Kernel Level Thread

Kernel-level thread (KLT): il kernel gestisce direttamente i thread. Le operazioni sono ottenute attraverso system call. Vantaggi:

- lo scheduling del kernel è per thread, non per processo \Rightarrow un thread che si blocca non blocca l'intero processo
- Utile per i processi I/O bound e sistemi multiprocessor

Svantaggi:

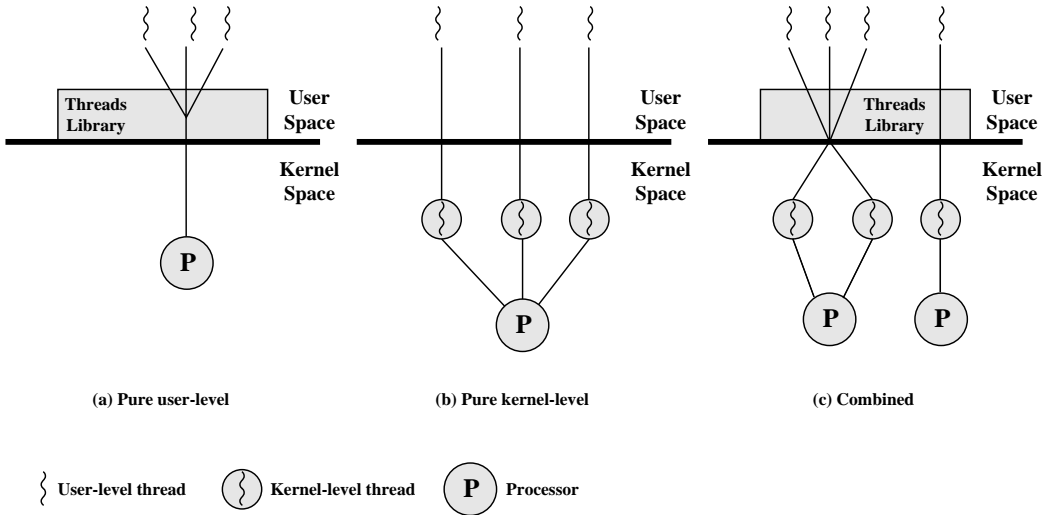
- meno efficiente: costo della system call per ogni operazione sui thread
- necessita l'aggiunta e la riscrittura di system call dei kernel preesistenti
- meno portabile
- la politica di scheduling è fissata dal kernel e non può essere modificata

Esempi: molti Unix moderni, OS/2, Mach.

16

Implementazioni ibride ULT/KLT

Sistemi ibridi: permettono sia thread livello utente che kernel.



17

Implementazioni ibride (cont.)

Vantaggi:

- tutti quelli dei ULT e KLT
- alta flessibilità: il programmatore può scegliere di volta in volta il tipo di thread che meglio si adatta

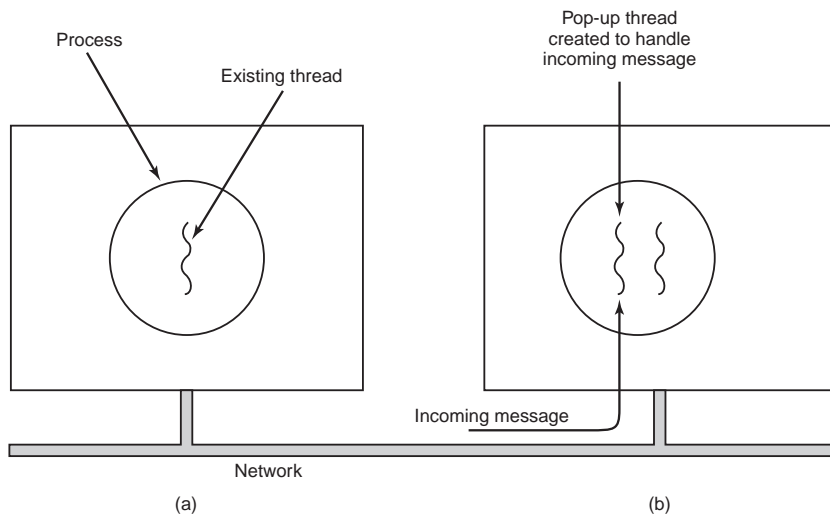
Svantaggio: portabilità

Es: Solaris 2 (thread/pthread e LWP), Linux (pthread e cloni), Mac OS X, Windows NT, ...

18

Thread pop-up

- I thread *pop-up* sono thread creati in modo asincrono da eventi esterni.



(a) prima; (b) dopo aver ricevuto un messaggio esterno da gestire

19

- Molto utili in contesti distribuiti, e per servizio a eventi esterni
- Bassi tempi di latenza (creazione rapida)
- Complicazioni: dove eseguirli?
 - in user space: safe, ma in quale processo? uno nuovo? crearlo costa...
 - in kernel space: veloce, semplice, ma delicato (thread bacati possono fare grossi danni)
- Implementato in Solaris

Processi e Thread di Windows 2000

Nel gergo Windows:

Job: collezione di processi che condividono quota e limiti

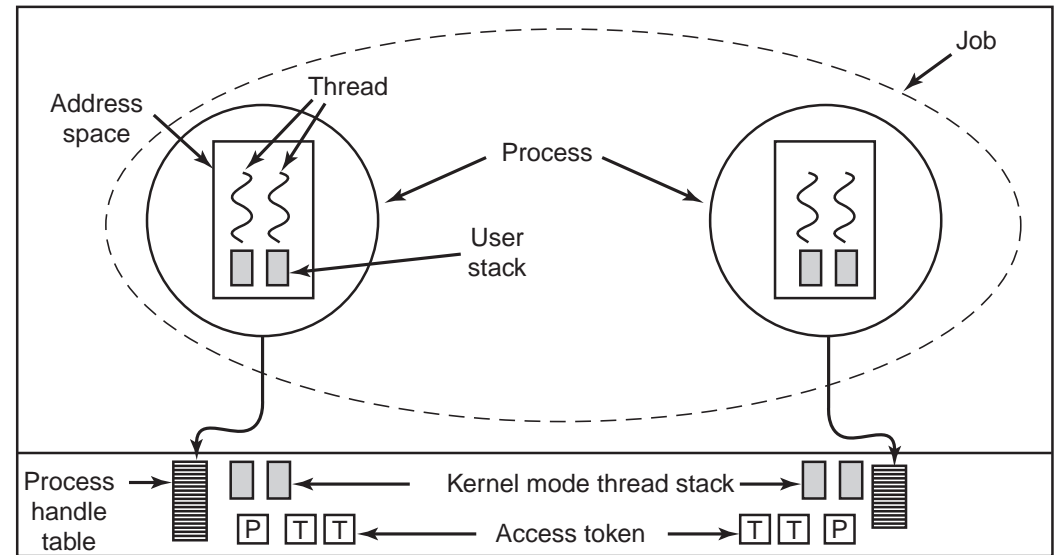
Processo: Dominio di allocazione risorse (ID di processo, token di accesso, handle per gli oggetti che usa). Creato con `CreateProcess` con un thread, poi ne può allocare altri.

Thread: entità schedulata dal kernel. Alterna il modo user e modo kernel. Doppio stack. Creato con `CreateThread`.

Fibra (thread leggero): thread a livello utente. Invisibili al kernel.

20

Job, processi e thread in Windows 2000



21