Scheduling 3 Multiprocessore

martedì 28 ottobre 2025 11:33

Andiamo ora a cambiare lo scheduling di CPU: dato che le macchine si sono evolute , passando così ad essere multiprocessori e la possibilità di usare i thread , è stato necessario questo cambio. Quindi andiamo a vedere ora come sono fatti i classici sistemi attuali : **sistemi multiprocessore** . In dettaglio :

Caratteristiche architetturali
- unita' di calcolo (CPU/CPU-core) multiple che condividono una memoria principale comune
- i processori sono controllati da un unico sistema operativo

↳ tightly coupled system (sistema strettamente accoppiato)

Problematiche
- assegnazione dei processi ai processori
- uso (o non) di politiche classiche di multiprogrammazione sui singoli processori
- selezione dell'entita' schedulabile da mandare in esecuzione

Quindi in dettaglio : come assegniamo i processi (thread) alle unità di calcolo :

Statica
- overhead ridotto poiche' l'assegnazione e' unica per tutta la durata del processo
- possibilita' di sottoutilizzo dei processori

Dinamica (convenzionalmente utilizzata in sistemi moderni)
- overhead superiore dovuto a riassegnazioni multiple
- migliore utilizzo dei processori

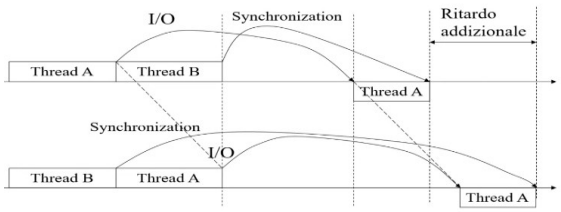
Approccio master/slave
- il sistema operativo viene eseguito su una specifica unita' di calcolo
- richiesta esplicita di accesso a stutture del kernel da parte delle altre
- semplicita' di progetto (estensione di kernel classici per uniprocessori)

Approccio peer (convenzionalmente utilizzato in sistemi moderni)
- il sistema operativo viene eseguito su tutte le unita' di calcolo
- problemi di coerenza dei dati, inclusi quelli di gestione dello scheduling
- complessita' di progetto

Ma conviene avere la **multiprogrammazione su singolo processore ?** In teoria no in quanto se si hanno molti processori, potrebbe capitare che alcuni non vengono utilizzati , in quanto l'impatto è minimo , quindi ci focalizziamo sul tempo di turn around (tempo tra inizio e tempo di completamento) . Quindi facendo un paragone :

Tutto questo vale <u>se e solo se si rimane in ambito di processi</u>. Vediamo ora cosa succede se parliamo di thread:

- la decomposizione di applicazioni in threads introduce criteri di selezione innovativi rispetto alle priorita' classiche (interattivita')
- un processo puo' essere sia CPU che I/O bound dipendendo dal comportamento dei singoli thread che lo compongono



Quindi in dettaglio si ha che si hanno due politiche di scheduling :

Load sharing (e.g. Linux 2.4)
- coda globale di threads pronti ad eseguire
- possibilità di gestire priorità ⟹
- distribuzione uniforme del carico
- problemi di scalabilità nell'accesso alla coda globale in caso di macchine altamente parallele
- ridotta efficienza del caching in caso di cambio di processore da parte dei thread

FCFS
SNTF (smallest number of threads first), con e senza Preemption
.....

Load balancing (e.g. Linux 2.6 e successivi)
- code di threads pronti ad eseguire separate, una per ogni CPU-core
- migliore scalabilità delle operazioni di scheduling su macchine altamente parallele
- spostamento periodico di thread da una coda ad un'altra in caso di sbilanciamento del carico