

Altri scheduling nei sistemi interattivi

- **Graziano**: E' stabilita una percentuale N sulle CPU e viene fatto tirare per l'algoritmo viene tenuto conto del rapporto tra la CPU usata e quella. Se è 0,3 l'algoritmo fa "reboot" per ripartire verso 1. Se è 2,0 l'algoritmo fa saltando per portare verso 1 così man mano che l'algoritmo come la situazione si stabilizza.
- **Lottizzazione**: E' come magazzinare biglietti con estrazioni random. Per estrarre le risorse si fa, appunto, l'estrazione. Qui si intravede un bisogno di priorità. Infatti se vogliamo che senza elegato venga un processo ad alto priorità basta organizzare gli biglietti. I biglietti si "comprano". Permette di avere processi cooperanti ovvero, se un processo ne vuole può cedere priorità biglietto a un altro processo.
- **Fair-Share**: Si cerca l'equità tra gli utenti. Se forniamo agli utenti "globalmente" non saremmo eguali negli utenti. Idea: fare che bello giving ogni utente e poi nei processi.

Scheduling dei thread

• thread utente (il SO non ne sa nulla)

E' un algoritmo non preemptive e lo scheduling è seassigazionale.

• thread del Kernel:

E' deve rispettare le considerazioni tali che i thread degli OS si considerano anche l'appartenenza a dei processi.

Infatti switchare su un thread non protetto implica riprogrammazione MMU (unità di gestione memoria) e a volte movimento cache della CPU.

In certe situazioni è meglio provare di non dover switchare su un fratello.

Scheduling nei sistemi multiprocessore

Possiamo fare:

• multi elaborazione simmetrica

Un dei processori fa da Master Server ovvero fa da amministratore per la gestione delle risorse. Le altre CPU lavorano sotto sua direttiva. Se ho più CPU il master server rischia di creare l'effetto "collo di bottiglia" e ritribuisce le prestazioni.

• multi elaborazione asimmetrica

Le CPU solo tutte uguali fra i loro, la CPU applica l'algoritmo di scheduling nel momento in cui non ha bisogno di thread.

L'algoritmo può prescindere di una cache unificata dei processi ma potrebbe essere una cache personale per ogni processo.

Coda manifestata: Permette di non dover fare

indirizzamento verso una certa coda ma essa è una struttura dati comune quindi soggetta a race conditions.
E' un problema risolvibile al costo che i processi aspettino per la metà esecuzione.

Coda ripida: obbliga a generare all'indirizzamento verso una certa coda per entrambi a dovere o a problemi legati alla struttura dati comune.

Politiche di Scheduling

Le politiche valgono in base alla priorità o sulla base di preliezione dei processi.
Preliezione: mantenere un thread in esecuzione mentre altri processi per sfruttare al meglio la cache.

La preliezione può essere debole: dove il SO prova a mantenere questa caratteristica ma la migrazione non è impossibile.

Forte: il thread è rimbalzato a tutti gli effetti da quel processo.

La preliezione crea dunque l'esecuzione
Processo Thread - Processo

Bilanciamento del Cocco

E' facile distribuire il lavoro in modo omogeneo nei processi.

E' necessario questo meccanismo se i lavori sono distinti per i processi.

Può avvenire tramite

Migrazione Forzata (PUSH)

Spostamento (PULL)

Push: un thread dedicato a occuparsi di fare migrare i thread per creare equilibrio.

Pull: un processo inattivo che "scatta" con i messaggi di un'altra colpa

Linux offre un approccio ibrido

Schedelet Windows 8

Usa un algoritmo preemptive basato su thread e classi di priorità.

E' possibile gestire le priorità

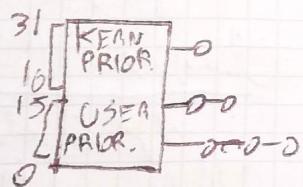
SetPriorityClass: per processi
SetThreadPriority: per thread

Controlliamo una tabella per le

priorità date e relative che indicano la priorità di base

Abbiamo 32 priorità: da 0 a 31

Lo schedulatore guarda solo ai thread.



E applica un algoritmo di selezione per priorità

All'interno della classe scelta abbiamo poi un algoritmo di scheduling orizzontale Round-Robin

La tabella delle priorità è così lista che sono molto difficili per farle saltare altre priorità, quindi si dà uno spazio di tempo definito per i thread real time.

Le priorità sono dimensionate (tranne per RT)

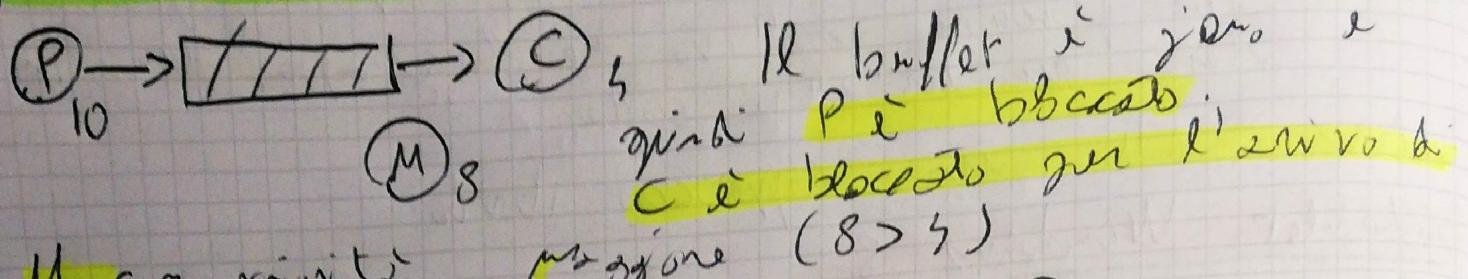
Ovvero a ogni ~~processo~~ thread applica la priorità di base e poi un altro incrementale

- Ma non quanto

- Rigetta da evento I/O

- Si potrebbe aumentare il quanto in certe circostanze senza toccare la priorità

Questo istante potrebbe andare in conto di
Problema di inversione di priorità



M è bloccato e va alla fine di P.
Si risolve facilmente con la tecnica
Autoboot

Come faccio ma è "minato", individua le priorità
tra processi e le risolve modificando
in modo opportuno le tempi massimi delle priorità.

Close di Priorità 0: si rivede un thread
speciale legato solo quando non c'è
altro thread.
E' un task che non è necessario legarlo vero,
necessita della registrazione.

TASK in LINUX

Giungiamo a un'unica entità che sintetizza
concrete le processi e thread, permette di
implementare semafori

E' una task_struct (nella PCB), i campi
e contiene info sul task.

Si usa la syscall clone(func, stack, sharing flags, args)
che restituisce un PID

E' un modo unico per fare fork e thread_create,
infatti queste 3 chiamate sono coincidono sotto
cette condizioni (detta dai flags)

Si usano PID per retrocompatibilità
TID: task identifier
Thread dello stesso processo ← PID uguali
NB: 2 task hanno TID diversi
ma potrebbero avere