Scheduler O(1) di Linux

Sostituito poi da CFS

Distinguiamo 3 classi di thread:
- RT FIFO (non preemptive)
- RT Round Robin (preemptive)
- time sharing (preemptive)

Le priorità sono da 0 a 139
0 è la più alta, 139 la più bassa.
RT FIFO e Round Robin vanno da 0 a 99 mentre il time sharing da 100 a 139

Le variazioni di priorità possono essere
- statiche: è una syscall nice che conta da -20 a +19
  ↓
  se non intervengono ulteriori modifiche queste rimangono
- dinamiche: da -5 a +5 per I/O bound vs CPU bound

Struttura algoritmo

Per ogni core abbiamo una run queue
Ci troviamo poi all'interno della run queue la coda active e la coda expired.
La active e la expired sono gruppi di code multiple, con una gestione simile allo scheduling a code multiple (c'è doppio scheduling)
In ogni coda è poi applicato il Round Robin

Nell'esecuzione andiamo a lanciare il task a priorità maggiore e scendiamo finché non si vuota la active per poi switchare alla expired

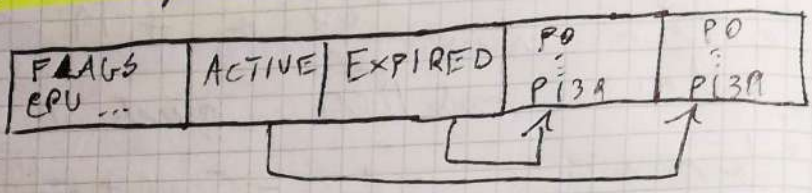
Quando un task finisce il suo quanto, è spostato nella expired.

Il Round Robin funziona che se un task ha I/O dopo $\frac{1}{3}$ del suo q alla fine dell'I/O il task è inserito alla fine della coda ma con $\frac{2}{3}$ del q

Se un processo aveva debiti x ms ma non finisce dopo quei x è spostato nella expired.

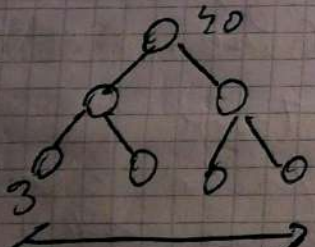
I quanti sono lunghi (800 ms) per priorità alte e corti (5 ms) per priorità basse.
I declassamenti e innalzamenti di priorità avvengono solo quando il processo passa dalla active alla expired
NB active e expired si alternano
L'algoritmo è O(1)
Quando un processo è bloccato è tolto dalla struttura momentaneamente
L'algoritmo è funzionante per multiprocessori.

Struttura RUN queue



Scheduler CFS

Crea lo scheduler garantito, cosa che non fa O(1)
Esso usata una variabile detta Virtual run-Time ovvero il tempo che una task usato la CPU
Usiamo un albero rosso nero che contiene i task e usa come chiavi le rispettive virtual run-time



più bisogno di CPU — meno bisogno

E' un albero autobilanciante quindi buono per la ricerca.
L'intuizione è far girare il task tutto a sx
Il nodo successivo la chiave e l'albero si bilancia.
Il bilanciamento è fatto periodicamente per non appesantire l'algoritmo, quindi preso il task lo si fa correre un po' e poi si fa il bilanciamento. I timeslice sono dinamici e dipendono anche dal numero di nodi

Granularità: bassa (oggetto ci va poco tempo) è usata nei desktop
alta (oggetto ci va un tempo netto) usata nei sistemi tipo-batch.

Sempre: aver registro il virtual run-time di un altro nodo

I numeri crescono continuamente, prima o poi andranno in over/low.
Per evitarlo facciamo un "reset", ovvero il task con key minimo diventa 0, e un altro nodo ora più alto di 30 diventerà infatti 30.
I D rimangono uguali

Quando arriva un task nuovo potrebbe essere reso come minimo 0 come massimo.

~~Com'è implementato~~

Conseguenza: Un I/O-bounded tende a rimanere a sx nell'albero per sua natura, un CPU-bounded al contrario.

Com'è implementato il concetto di priorità?
Coi fattori di decadenza: il virtual run-time dei processi a bassa priorità avanza più velocemente
Il fattore è piccolo per i processi ad alta priorità e più grande per i processi a bassa priorità.
Con questo sistema elegante evitiamo starvation e aging.

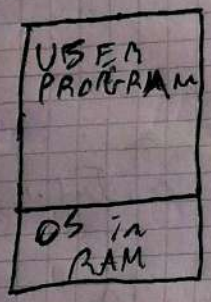
La schedulazione può avvenire anche per gruppi, così creiamo più fair share tra utenti che processi di gruppi.

Esempio: Nell'albero ci sono gli utenti, ogni utente "contiene" i processi propri
Una a basso volta saranno schedulati con un albero.

Gestione Memoria

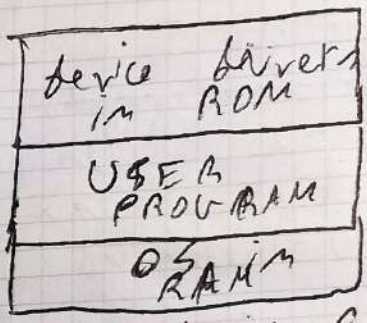
La memoria è divisa per gerarchia
La RAM è il livello più basso che è utilizzabile dalla CPU direttamente
Nei sistemi moderni i processi sono aiutati nella gestione della memoria tramite le astrazioni che diventano più via più complesse.

I primi modelli erano senza astrazione
I programmi usavano direttamente gli indirizzi fisici il che rendeva molto complicato fare multiprogrammazione
Esistono vari modelli



OS nella parte inferiore della RAM

OS in cima nella ROM



drivet in ROM e il resto in RAM

Multiprogrammazione senza astrazione

Mantenere più processi in memoria partizionando la RAM
Si presentano problemi risolvibili con la rilocazione

Rilocazione < Compile-time
Statica:
Si fa una "traduzione" degli indirizzi logici in quelli fisici per evitare casini con lettura e scrittura.
La traduzione avviene semplicemente tenendo conto della cella a cui inizia P.
Svantaggio: il loader viene rallentato.

Se per un qualunque motivo un codice genera un indirizzo fuori dalla porzione dobbiamo pensare a un meccanismo di protezione della memoria

- lock & key: questo meccanismo fa da intermediario tra CPU che genera indirizzi e memoria che deve essere letta / scritta.
Per ogni blocco abbiamo un numero che è una chiave.
La CPU avrà una chiave dentro a un registro speciale
L'MMU controlla se la chiave della CPU coincide con la chiave del blocco e in caso contrario nega l'accesso

Se ho un processo che occupa più blocchi, quindi blocco hanno la stessa chiave.
Nel context switch la CPU cambia la sua chiave.

SPAZIO DEGLI INDIRIZZI

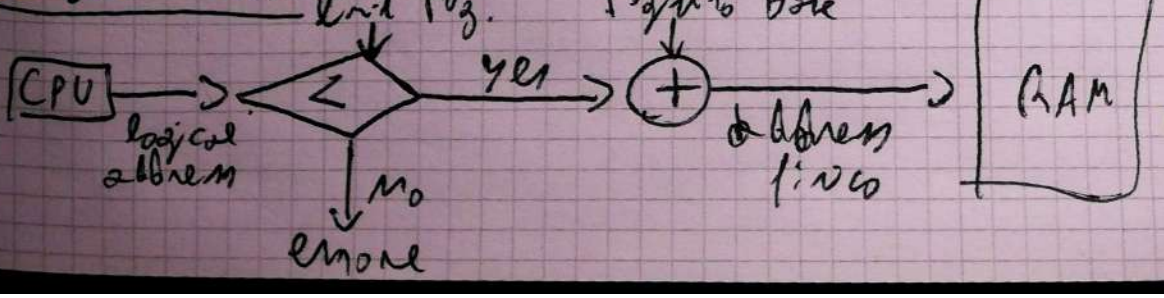
E' un'astrazione per la memoria che può essere usata da un processo. (run-time)
Faremo una rilocazione dinamica usando registro base e registro limite.

CPU → MMU → RAM La CPU genera gli indirizzi logici tradotti da MMU usando i due registri.

In sistemi meno evoluti non c'è l'MMU a fare da intermediario.

- registro base: indirizzo fisico inizio spazio
- registro limite: "ampiezza" porzione.

Controllo MMU



Context switch: MMU ricarica i due registri