Allocazione dei Processori nei Sistemi Distribuiti

Moreno Marzolla

marzolla@dsi.unive.it

http://www.dsi.unive.it/~marzolla

Sommario

- Scheduling nei sistemi centralizzati
- Allocazione dei Processi nei sistemi distribuiti
  - Definizione del problema
  - Tassonomia
- Esempi
  - Un algoritmo deterministico
  - Un algoritmo centralizzato
  - *Bidding Algorithm*
  - Un algoritmo gerarchico
  - Scheduling nei sistemi distribuiti: *coscheduling*
  - *Orphan Cleanup*

Bibliografia

- A.S.Tanenbaum, *Modern Operating Systems*, Prentice Hall, pp. 534-547
- Doreen L. Galli, *Distributed Operating Systems: Concepts and Practice*, Prentice Hall, 1999, pp. 151-175
- Colouris, Dollimore, Kindberg, *Distributed Systems Concepts and Design*, pp. 217-218
- Queste trasparenze: http://www.dsi.unive.it/~marzolla

Scheduling nei sistemi centralizzati/1

- Il processore dedica un *quanto* di tempo ad ogni processo. I processi vengono inseriti in una coda, e lo *scheduler* decide quale assegnare al processore ad ogni quanto.
  - Una infinità di algoritmi di scheduling
    - Round Robin, con priorità, Shortest Job First, Guaranteed Scheduling...
  - Molti obiettivi contrastanti tra loro da raggiungere:
    - Assicurarsi che ogni processo riceva una quota equa di CPU
    - Far sì che la CPU sia impegnata il più possibile
    - Minimizzare il tempo di risposta per gli utenti interattivi
    - Minimizzare il tempo di completamento dei processi batch
    - Massimizzare il *throughput*
    - *...*

Scheduling nei sistemi centralizzati/2

- In realtà, c'è almeno un problema che nei sistemi centralizzati non si pone (quasi) mai:
  - ➔ Decidere a quale processore assegnare un processo (tanto c'è un solo processore!)
- Fanno eccezione i sistemi *multiprocessore simmetrici* (SMP)
  - Composti da pochi (2 o 4) processori identici a memoria condivisa
  - Ai fini dello scheduling un processore vale l'altro, visto che tutti "vedono" la stessa memoria
    - ci sono però altre considerazioni di cui tener conto, che vedremo in seguito
  - Lo scheduler dovrebbe cercare di riassegnare lo stesso processo al medesimo processore, nella speranza di migliorare l'uso della cache interna ai processori *(Processor Affinity)*

Esempio di scheduling SMP in Linux (2.2.14-5.0)

/usr/src/linux/kernel/sched.c

```
static inline int goodness (struct task_struct * prev,
                                    struct task_struct * p, int this_cpu)
{
        int weight;

        if (p->policy != SCHED_OTHER) {
                weight = 1000 + p->rt_priority;
                goto out;
        }

        weight = p->counter;
        if (!weight)
                goto out;

#ifdef __SMP__
        /* Give a largish advantage to the same processor...   */
        /* (this is equivalent to penalizing other processors) */
        if (p->processor == this_cpu)
                weight += PROC_CHANGE_PENALTY;
#endif

        /* .. and a slight advantage to the current MM */
        if (p->mm == prev->mm)
                weight += 1;
        weight += p->priority;

out:
        return weight;
}
```


Allocazione dei processi nei sistemi distribuiti

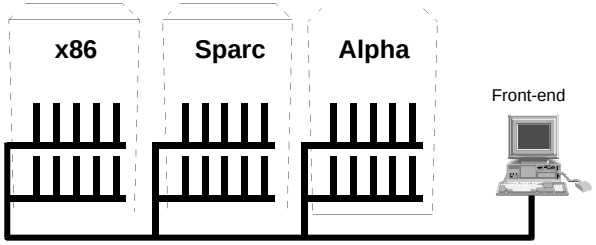
- Per definizione, un sistema distribuito è composto da più processori. Possono essere:
  1) un insieme di workstations
  2) uno o più pool di processori
  3) una combinazione tra i due
- Il problema dell'allocazione dei processori (PA): Quale processore assegnare ad un processo che deve essere eseguito?

Allocazione dei Processori

- Nel modello (1) occorre decidere, per ogni nuovo processo, se esso vada lanciato localmente o su un'altra workstation;
  - Inoltre: una volta partito sulla workstation X, se questa diviene in seguito troppo carica conviene o no migrarlo su una diversa workstation Y?
- Nel modello (2) occorre decidere, per ogni processo, dove vada lanciato.

Modelli di allocazione

- Solitamente si assume che tutti processori siano identici, o almeno compatibili a livello di codice
  - Possono differire al più per la velocità (es. un cluster di workstation Pentium di diversa frequenza di clock)
  - Occasionalmente sono stati considerati modelli composti da pool disgiunti di processori, ciascuno dei quali composto da processori omogenei:



Obbiettivi degli algoritmi di PA

- L'uso di un algoritmo implica che si sta cercando di ottimizzare *qualcosa*
  - ...altrimenti sarebbe più facile ed economico allocare i processi a caso sui processori

➔ ***Massimizzare* l'utilizzazione della CPU**

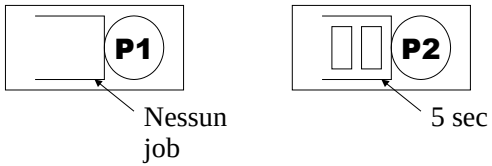
- ossia massimizzare il numero di cicli di CPU effettivamente spesi per eseguire i processi utente;
- in altre parole, ogni CPU deve sempre essere impegnata;

➔ **Distribuire *equamente* il carico di lavoro tra tutte le CPU**

➔ **Minimizzare il tempo *medio* di risposta**

Esempio: Minimizzare il tempo medio di completamento

- Due processori: P1 (10 MIPS), P2 (100 MIPS)
  - P2 ha due processi in coda che termineranno fra 5 secondi. P1 non ha processi in coda
- Due processi: A (10^8 istruzioni), B ($3*10^8$ istruzioni)



Assegnamento 1: A in P1, B in P2

$T_{medio} = (10+8) / 2 = 9$ sec

Assegnamento 2: A in P2, B in P1

$T_{medio} = (30+6) / 2 = 18$ sec

Classificazione degli algoritmi di PA/1

- **Deterministici / Euristici**
  - Algoritmi deterministici assumono di avere completa conoscenza del comportamento futuro dei processi da eseguire. Ciò è solitamente impossibile da ottenere.
- **Migratori / Non-migratori**
  - Nei sistemi migratori i processi possono venire migrati (spostati) su un processore diverso durante l'esecuzione. Nei sistemi non-migratori l'assegnamento processo-processore viene fatto all'inizio e la decisione non viene cambiata una volta presa.
- **Centralizzati / Distribuiti**
  - Concentrare tutte le informazioni in un punto (algoritmi centralizzati) solitamente consente di prendere decisioni migliori, ma è meno robusto e può rappresentare un collo di bottiglia.

Classificazione degli algoritmi di PA/2

- **Ottimi / Sub-ottimi**
  - I problemi di ottimizzazione associati allo scheduling spesso sono ***NP-completi***, per cui il calcolo della soluzione ottima è di fatto computazionalmente intrattabile. In casi simili occorre ripiegare su algoritmi sub-ottimali aventi però complessità computazionale "ragionevole".
- **Locali / Globali** *(Transfer Policy)*
  - Creato un processo occorre decidere se eseguirlo localmente oppure no. La decisione si può basare su criteri *locali* (es., se il carico del processore locale è inferiore ad una soglia), oppure *globali* (raccogliere informazioni sul carico degli altri processori per prendere una decisione).
- ***Sender-initiated* / *Receiver-initiated*** *(Location Policy)*
  - Se si decide di trasferire il processo altrove, occorre decidere dove spedirlo. La decisione può essere intrapresa dal processore locale, o da quelli remoti che segnalano la propria disponibilità.

Alcuni problemi

- Gli algoritmi di PA assumono di conoscere con sicurezza l'*utilizzazione* di un processore (frazione del tempo speso dal processore ad eseguire codice)
  - Come la misuro?
- La complessità computazionale degli algoritmi di PA spesso non viene considerata
  - Il tempo che impiego a decidere cosa fare potrebbe essere più utilmente impiegato per farlo!
- Molti degli algoritmi proposti non tengono in considerazione il tempo necessario per trasferire un processo da un processore ad un altro.
  - Spesso le reti di comunicazione sono il collo di bottiglia.

Esempio

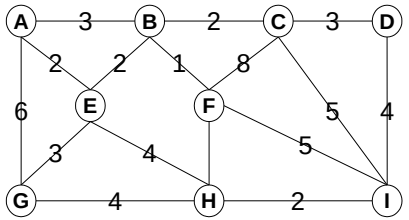
- Due workstation, *L* (locale) e *R* (remota), connesse con una rete da 100Mbit/s (≈10MB/s).
- Un eseguibile da 5MB
  - Su *L* impiega 1s per completare, su *R* che è più veloce, 0.7s.
- Il processo viene generato su *L*. Il software di allocazione dei processi deve decidere se eseguirlo localmente o spedirlo a *R*.
  - Se eseguo localmente su *L*, impiega 1s
  - Se spedisco il programma a *R*:
    - Tempo di trasferimento: 0.5s
    - Tempo di esecuzione su *R*: 0.7s
    - Totale: **1.2s**

Algoritmo 0: Probes

- Il processore su cui sta girando un processo diventa sovraccarico. Il processore decide di trasferire il processo altrove.
- “Sonda” un altro, chiedendo quale è il suo carico. Il processo può essere trasferito al processore meno utilizzato.
  - Se il processore sondato risponde di essere sottoutilizzato, il job viene trasferito a lui.
  - Altrimenti, una nuova sonda è inviata ad un altro processore scelto a caso, fino a che viene trovato un processore sottoutilizzato o si eccede un tetto massimo di tentativi a disposizione.

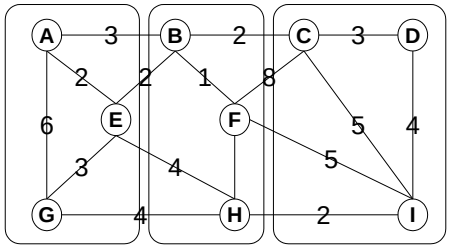
Algoritmo 1: Algoritmo deterministico

- Ho *N* processi da distribuire su *k* processori identici (*N*>*k*); per ogni coppia di processi è noto il traffico di messaggi che si scambiano.
  - **Obbiettivo: distribuire i processi in modo da minimizzare il traffico tra i processori**.
- Esempio con 9 processi e 3 processori:



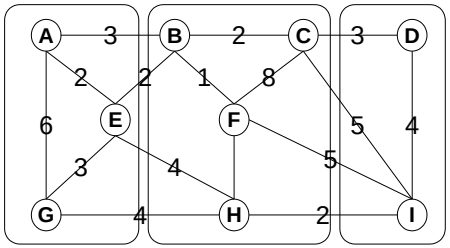
Algoritmo 1: Algoritmo deterministico

- Ho *N* processi da distribuire su *k* processori identici (*N*>*k*); per ogni coppia di processi è noto il traffico di messaggi che si scambiano.
  - **Obbiettivo: distribuire i processi in modo da minimizzare il traffico tra i processori**.
- Esempio con 9 processi e 3 processori:



Algoritmo 1: Algoritmo deterministico

- Ho *N* processi da distribuire su *k* processori identici (*N*>*k*); per ogni coppia di processi è noto il traffico di messaggi che si scambiano.
  - **Obbiettivo: distribuire i processi in modo da minimizzare il traffico tra i processori**.
- Esempio con 9 processi e 3 processori:



Problemi dell'Algoritmo 1

- Conoscere in anticipo il pattern di comunicazione tra i processi è nella maggior parte dei casi impossibile.
- Il flusso di dati scambiati tra processi può variare nel tempo. Quindi una allocazione ottimale in una fase della computazione può non esserlo in un'altra.
- In generale, partizionare un grafo in due sottografi con lo stesso numero di nodi tali che il numero di archi che attraversa la partizione sia minimo è un problema ***NP-hard***.

Algoritmo 2: Algoritmo centralizzato

Usage Table o Algoritmo up-down

- Scopo dell'algoritmo non è massimizzare l'utilizzo delle CPU, ma distribuire il carico di lavoro in modo equo
- Ad ogni workstation viene assegnato un punteggio
  - se consuma risorse altrui (spedisce ad altri processi originati localmente) accumula punti di penalità.
  - se fornisce risorse agli altri (fa girare processi originati da altre workstation) il suo punteggio diminuisce.
- Quando un processore è in stato *idle*, riceve il processo *originato dalla workstation col punteggio più basso*
- Variante: il punteggio viene incrementato/ decrementato di una quantità fissa per ogni secondo di CPU richiesto/offerto.

Esempio

Usage Table

WS	*Pt(t=0)*	*Pt(t=1)*	*Pt(t=2)*	*Pt(t=3)*	*Pt(t=4)*
A	6	**5**	5	5	**6**
B	-1	**0**	**-1**	-1	**-2**
C	3	3	3	**4**	4
D	2	2	**3**	**2**	2

Tempo

- t=0: situazione iniziale. Job pendenti $\{J_A, J_B, J_C, J_D\}$
- t=1: A è idle, riceve J_B. Job pendenti $\{J_A, J_C, J_D\}$
- t=2: B è idle, riceve J_D. Job pendenti $\{J_A, J_C\}$
- t=3: D è idle, riceve J_C. Job pendenti $\{J_A\}$
- t=4: B è idle, riceve J_A. Job pendenti $\{\}$

Algoritmo 3: Bidding Algorithm

- L'algoritmo precedente può essere esteso assumendo che l'insieme dei processori sia un sistema economico, e i prezzi delle risorse (tempo di CPU) siano fissati da leggi di mercato
  - Donald F. Ferguson, Christos Nikolaou, Jakka Sairamesh, Yechiam Yemini, *Economic Models for Allocating Resources in Computer Systems*, Market based Control of Distributed Systems, Ed. Scott Clearwater, World Scientific Press, 1996 (disponibile online su http://citeseer.nj.nec.com/ferguson96economic.html)
- Ciascun processore calcola il "prezzo" del servizio da lui offerto, basandosi sulle richieste e sulla "qualità" dei servizi
  - Velocità, quantità di RAM, presenza di unità Floating Point...
- Ogni volta che un processo deve essere eseguito, si valutano i prezzi dei vari processori, e il più conveniente viene scelto

Una applicazione del Bidding Algorithm

- MojoNation (http://www.mojonation.net) era un sistema peer-to-peer che usa un algoritmo simile per realizzare un *distributed load balancing*

1.1 What makes Mojo Nation different from other file-sharing systems?

Other file-sharing systems are plagued by "the tragedy of the commons," in which rational folks using a shared resource eat the resources to death. Most often, the "Tragedy of the Commons" refers to farmers and pasture, but technology journalists are writing about overloaded servers, and users who download repeatedly but never contribute to the system.

In Mojo Nation, every transaction costs some Mojo, and to acquire Mojo, one must contribute resources to the community. Users who do not contribute must wait in line for contested resources, so the Mojo payment system also serves as a distributed load balancer. When demand for content is not great, the cost of providing that data is close to zero. When there is contention for that resource, then the payment system comes into play -- Mojo Nation's distributed load balancing moves some clients to a less-occupied server, while other users have the option to use accumulated credit to move to the head of line.

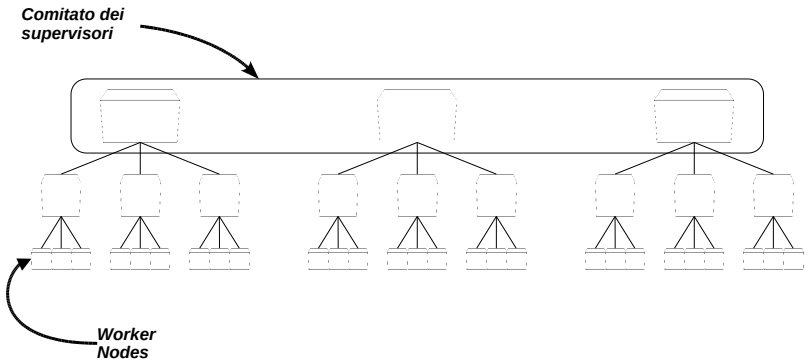
Algoritmo 4: Algoritmo gerarchico/1

- Gli algoritmi centralizzati hanno scalabilità limitata, perché il nodo centrale può diventare il collo di bottiglia.
- Il problema può essere affrontato impieganto un algoritmo gerarchico
  - Mantiene la semplicità dell'algoritmo centralizzato, ma ha migliori caratteristiche di scalabilità.

Algoritmo 4: Algoritmo gerarchico/2

- I processori hanno una struttura gerarchica:
  - Ogni k processori esiste un *supervisore* che tiene traccia del carico di lavoro di ogni CPU
  - La gerarchia può essere estesa definendo dei *supervisori* dei *supervisori*
  - Per evitare di avere un singolo supervisore al vertice della gerarchia, è possibile troncare l'albero in modo da avere un "comitato" quale autorità di vertice.
- In caso di fallimento di un supervisore, deve essere eletto un sostituto
  - Qualsiasi algoritmo di *coordinator election* può essere impiegato
  - Il nuovo supervisore può essere scelto tra i supervisori dello stesso livello, oppure uno del livello superiore, o infine può essere "promosso" uno dai livelli inferiori.

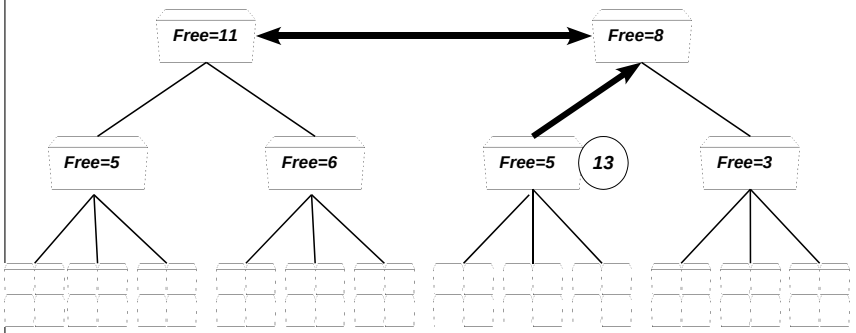
Esempio di struttura gerarchica



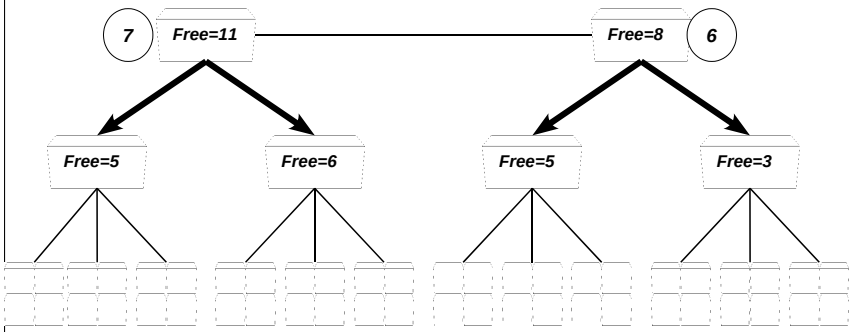
Funzionamento dell'algoritmo gerarchico

- Ipotesi di lavoro:
  - I processori sono *monoprogrammati* (eseguono un solo processo alla volta).
  - Le richieste di eseguire un job che richiede *S* processori possono apparire in qualunque nodo della gerarchia.
- Ciascun supervisore tiene traccia del numero di processori disponibili nella partizione di cui è a capo.
- Se riceve la richiesta di *S* processori avendone solo *F*<*S* a disposizione, la richiesta è propagata al livello superiore.
- Altrimenti, la richiesta viene suddivisa in parti, in base alla disponibilità di CPU libere che risultano ai livelli inferiori, e le parti vengono propagate verso il basso, fino al livello dei processori.

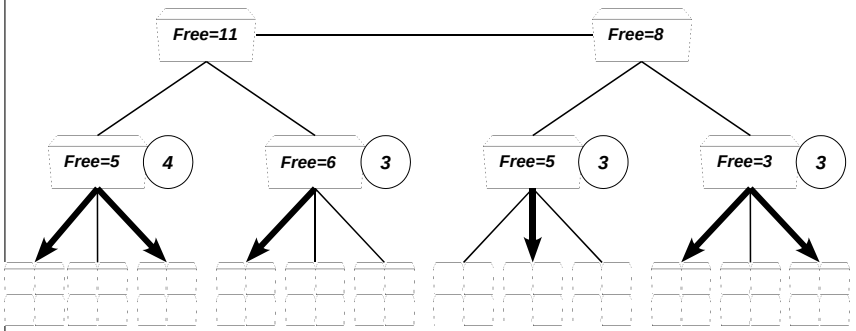
Esempio/1



Esempio/2

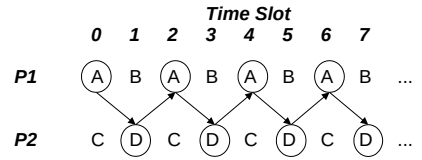


Esempio/3



Scheduling in sistemi distribuiti

- Ogni processore effettua il proprio scheduling dei processi che girano localmente.
- Tuttavia, schedulare indipendentemente processi che comunicano frequentemente non è una buona idea
  - Supponiamo che il processore P1 esegua i processi A e B, mentre il processore P2 esegua i processi C e D. Supponiamo che A e D comunichino frequentemente. La schedulazione seguente è piuttosto penalizzante per A e D:



Coscheduling

- L'idea è evidente: *conviene eseguire insieme tutti i processi che comunicano tra di loro*.
- Consideriamo una matrice di *N* colonne (dove *N* è il numero di processori).
  - Ogni riga della matrice rappresenta un *time slot*.
  - Ogni cella della matrice può essere vuota o contenere l'ID di un processo

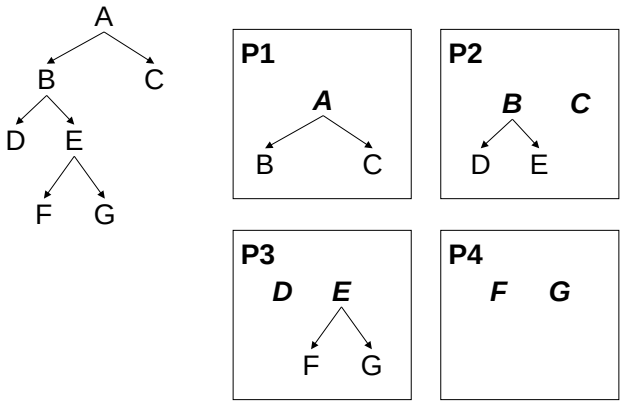
Time slot	Processori 0	1	2	3	4	5	6
0	A			D		B	
1		C	E				G
2	A	C		D	H		G
3			F				
4	A			D		B	

- Ad ogni passo, tutti i processori eseguono i processi loro assegnati per quel time slot

Orphan Cleanup

- Un processo può creare processi figli, che possono venire migrati su altri processori rispetto al padre.
- Se il processo padre termina prematuramente, è necessario terminare anche tutti i discendenti.
- Metodo del *Family Tree:*
  - Nella versione centralizzata, il padre deve mantenere (in memoria non volatile) *l'intero albero genealogico*, contenente i propri figli e tutti i loro discendenti.
  - Nella versione distribuita, ogni processo tiene memorizzato (in memoria non volatile) la lista dei processi figli da lui *direttamente* creati.
  - Al riavvio dopo un crash, il Sistema Operativo provvede a terminare tutti i processi figli, e ricorsivamente anche i loro discendenti

Es. Distributed Family Tree



Process Allowance

- Ogni processo figlio chiede al padre un intervallo di tempo (*allowance*) in cui è autorizzato ad eseguire.
- Se scaduto il tempo la sua computazione non è terminata, deve chiedere una nuova *allowance* al padre.
  - A sua volta, se il padre è figlio di un altro processo, deve chiedere l'allowance al proprio padre prima di concederla al figlio.
- In questo modo, il processo radice della gerarchia rilascia periodicamente autorizzazioni ai propri discendenti ad eseguire per un intervallo di tempo.
- Se un processo termina prematuramente, i discendenti ancora in esecuzione non riceveranno più *allowance* e termineranno.