Corso di Linguaggi di Programmazione

Lezione 20

Alberto Ceselli
alberto.ceselli@unimi.it

Dipartimento di Tecnologie dell'Informazione
Università degli Studi di Milano

14 Maggio 2013

Supporto alla programmazione concorrente

Programmazione concorrente:

- il flusso di controllo è composto da più *"thread"* indipendenti.
- Esempio: Arcieri, Vento e Giudice (alla lavagna)
- ogni "thread" può coinvolgere istruzioni su processori di macchine diverse, su diversi processori della stessa macchina o sul processore di una macchina singolo-processore.

Ambiti applicativi

- Storicamente: multitasking nei sistemi operativi
- Calcolo intensivo
- Simulazione
- Event–driven programming
- Comunicazione tra applicazioni gi esistenti

Livelli di concorrenza

- *instruction level*
- statement level
- unit (subprogram) level
- *program level*

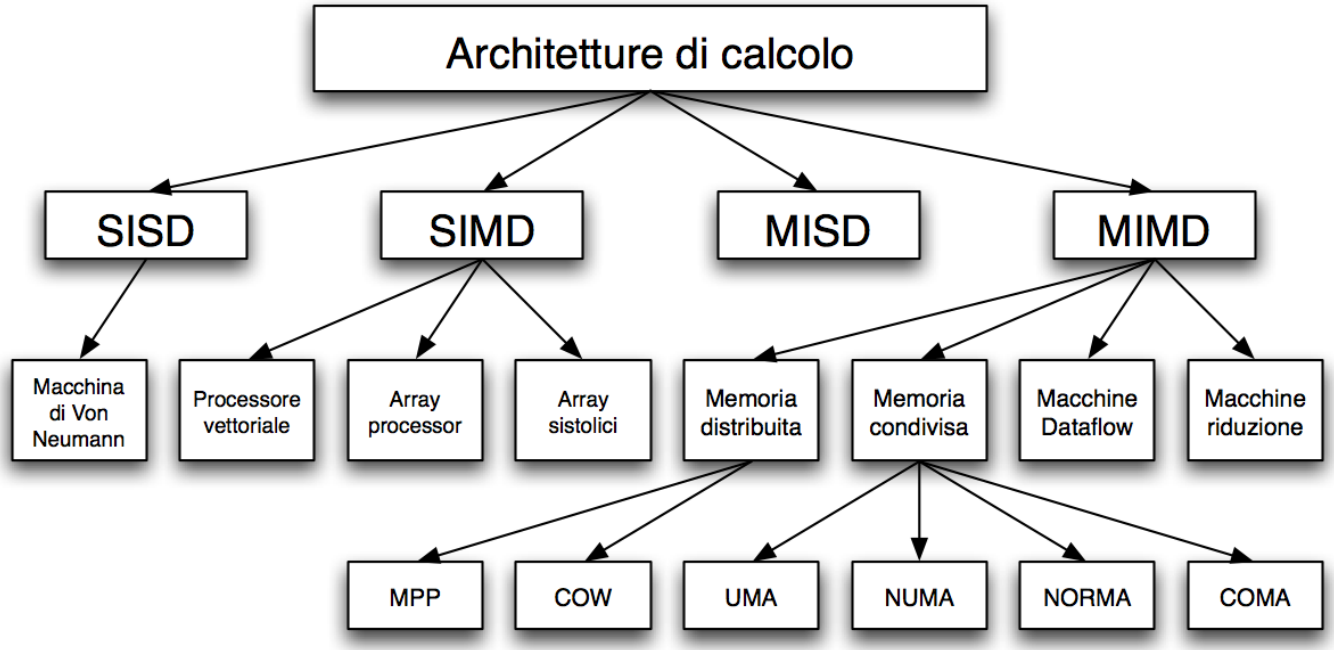
Livelli di concorrenza

- *instruction level*: gestito dalla macchina, es. super-pipelining
- statement level: esempio parbegin / parend
- unit (subprogram) level: es. monitor ...
- *program level*: gestito dal sistema operativo, es. fork / join

Architetture multiprocessore

- Anni '50: processore general purpose + processore per I/O
- Primi anni '60: piu' processori completi, job scheduler distribuisce job estraendoli da una coda di batch (program level concurrency)
- Metà anni '60: computers multiprocessore, con più sezioni (es. moltiplicatori floating point, o FPU)
- Il *compilatore* deve decidere quali istruzioni eseguire in parallelo e su quali dispositivi (instruction level concurrency)

Tassonomia di Flynn (1966)



Single Instruction, Multiple Data (SIMD)

- Single Instruction, Multiple Data (SIMD)
- tipicamente, ogni processore ha la sua memoria locale
- c'è un processore che *controlla* le operazioni degli altri
- esempio: vector processors
- non c'è bisogno di sincronizzazione
- **statement level concurrency**

Multiple Instruction, Multiple Data (MIMD)

- Multiple Instruction, Multiple Data (MIMD)
- ogni processore ha il suo flusso di istruzioni
- possono essere sistemi a memoria
  - *distribuita*: anche fisicamente in locazioni diverse
  - *condivisa*: fisicamente nella stessa locazione
- è richiesta sincronizzazione (specialmente per i sistemi a memoria condivisa, per gestire gli accessi alla memoria)
- **unit level concurrency**

Concorrenza fisica e logica

- Concorrenza **fisica**: assumendo di avere più di un processore, parti dello stesso programma sono effettivamente eseguite contemporaneamente su processori diversi
- Concorrenza **logica**: si assume di avere più di un processore, anche se di fatto la macchina dispone di *un solo processore*, ed il sistema operativo esegue parti dello stesso programma *sequenzialmente* e *ad intervalli* sullo stesso processore.

Concorrenza fisica e logica

- E' compito di chi *implementa* il linguaggio rendere la concorrenza logica trasparente sia al *progettista* del linguaggio che al *programmatore*.
- N.B. I costrutti offerti dal linguaggio dovrebbero essere il più possibile *indipendenti* dal tipo di esecuzione (concetto di *astrazione*)

Definizioni

- Un **task** è una porzione di programma (es. un sottoprogramma) che può essere in esecuzione concorrente con altre porzioni del programma.
- Gli elementi di un insieme di task che *condividono lo stesso spazio di memoria* vengono detti **thread** o task **lightweight**.
- Un insieme di task che non condivide memoria con altri task è detto **processo** o task **heavyweight**.

Tasks e sottoprogrammi

Differenze tra sottoprogrammi e tasks:

- un task può partire implicitamente (mentre i sottoprogrammi devono essere chiamati)
- quando una porzione di programma invoca un task, non deve aspettare che il task sia completato per riprendere l'esecuzione
- quando il task è completo, il controllo potrebbe anche **non** tornare alla porzione di codice che l'ha invocato

Tasks e sincronizzazione

- un task che non comunica con altri e non ne influenza l'esecuzione è detto **disgiunto**
- l'esecuzione e l'accesso ai dati di task non disgiunti deve essere **sincronizzata**
- formalmente, la **sincronizzazione** è il meccanismo che controlla l'ordine in cui i task vengono eseguiti

Tipi di sincronizzazione

- **competition synchronization**: dua task A e B devono accedere ad una risorsa che non può essere utilizzata simultaneamente (es. A e B vogliono scrivere dati in un buffer condiviso)
- **cooperation synchronization**: presi due task A e B, A deve aspettare che B abbia completato un'elaborazione specifica prima di continuare la sua esecuzione (es. B: produttore - A: consumatore)

Esempio di competition synchronization

Operazioni sulla stessa variabile: (alla lavagna)

- READ TOTAL; TOTAL++; STORE TOTAL;
- READ TOTAL; TOTAL *= 2; STORE TOTAL;

- senza competition synchronization, si possono ottenere tre diversi risultati
- quando il comportamento del programma dipende da quale thread arriva prima, si parla di **race condition**.

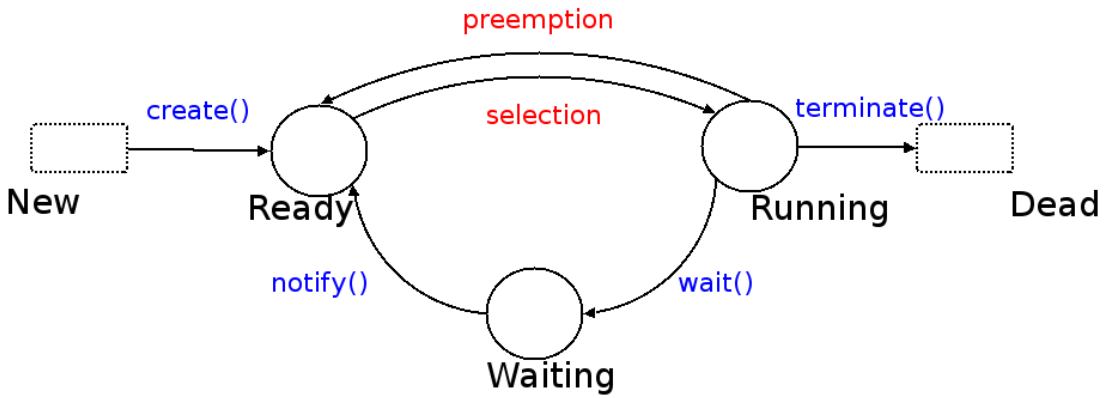
Esempio di cooperation synchronization

Produttore–Consumatore (alla lavagna)

- Produttore: INPUT TOTAL; ComputeP(TOTAL); STORE TOTAL;
- Consumatore: READ TOTAL; ComputeC(TOTAL); OUTPUT TOTAL;

Stato di un task

- Garantire l'accesso mutuamente esclusivo significa considerare la risorsa condivisa (codice o dati) come *atomica*, e permettere ad un solo task alla volta di averne possesso
- Task che necessitano di una risorsa in uso si mettono in attesa
- Task che rilasciano una risorsa ne notificano il rilascio



Attesa attiva

- Come gestire lo stato `Waiting`?
- "soluzione" 1 (impraticabile!): **attesa attiva**

```
void wait ( time_t delay ) {
time_t current_time;

    while (current_time < delay) {
     gettime(&current_time);
    }
}
```

- soluzione 2: lasciare il controllo al *sistema operativo*.

Non determinismo

- La gestione delle code, le operazioni di selection e preemption ecc sono demandate al sistema operativo
- sono, pertanto *incognite* al programmatore
- il flusso effettivo di attivazione dei task è, agli occhi del programmatore, *non-deterministico*
- **liveness**: è la caratteristica di un programma di proseguire l'esecuzione, raggiungendo ad un certo punto la terminazione (es. un programma che cade in ciclo infinito non è più *alive*)
- N.B. liveness è una proprietà *semantica* del programma
- come "garantire" liveness se il comportamento è non-deterministico?

Deadlock

Deadlock: ogni task possiede alcune risorse, ma è in attesa del rilascio di altre risorse da parte di altri task.
Condizioni necessarie per il verificarsi di un Deadlock

- Mutua esclusione
- Possesso ed attesa (possibile soluzione: richiedo allocazione simultanea delle risorse necessarie)
- Nessuna preemption
- Attesa circolare (possibile soluzione: ordino le risorse)

Starvation

Starvation: un task non riesce a proseguire la propria esecuzione perchè in attesa di risorse che vengono iterativamente occupate da altri task.
Esempio: la Cena dei Filosofi

Buona programmazione significa ...

Idea:

- Il mio linguaggio di programmazione deve mettere a disposizione *costrutti* per la programmazione concorrente
- Utilizzando questi costrutti, il programmatore deve essere in grado di rendere il programma "a prova di task".
- Domanda: questi costrutti sono *safe* (ad esempio: se usati male, possono pregiudicare la liveness del programma)?

Buona programmazione significa:

- Sfruttare al massimo la possibilità di parallelismo
- garantendo sincronizzazione
- evitando deadlock e starvation (quando possibile)