Corso di Sistemi Operativi

Sincronizzazione

Alberto Ceselli
alberto.ceselli@unimi.it

Dipartimento di Informatica
Università degli Studi di Milano

April 13, 2016

Supporto alla programmazione concorrente

Programmazione concorrente:

- il flusso di controllo è composto da più *"task"* indipendenti.
- ogni "task" 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*

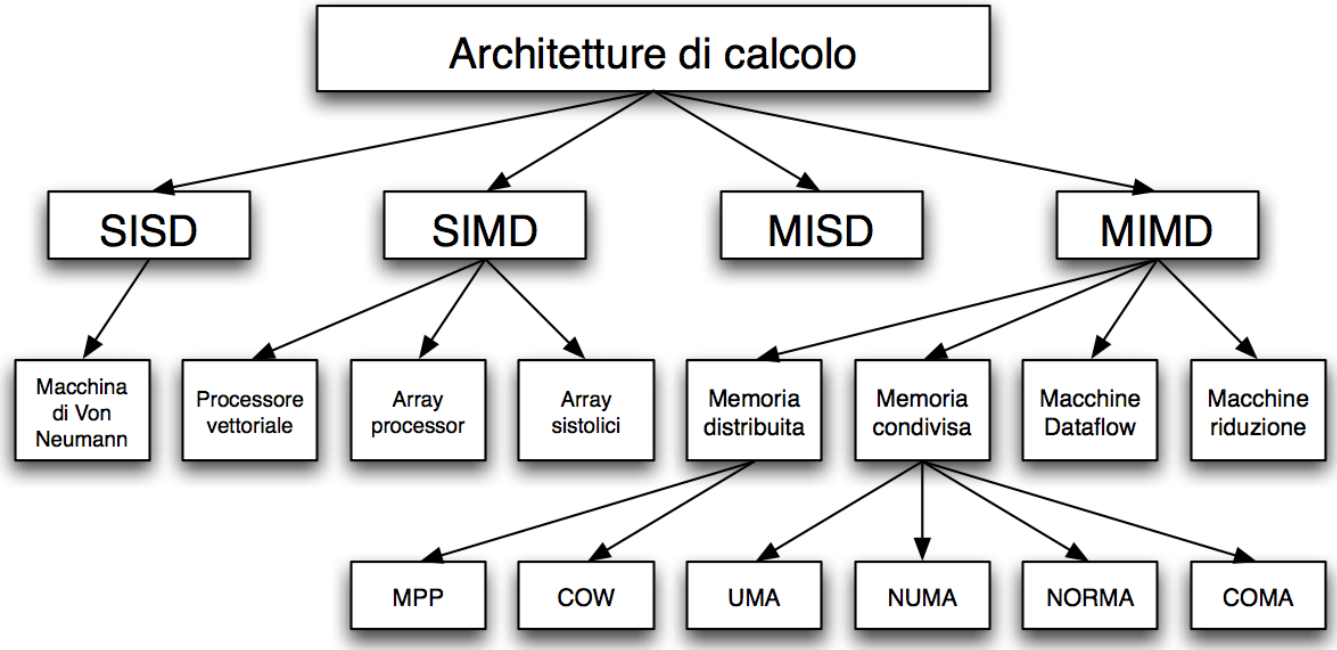
Livelli di concorrenza

- *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**.

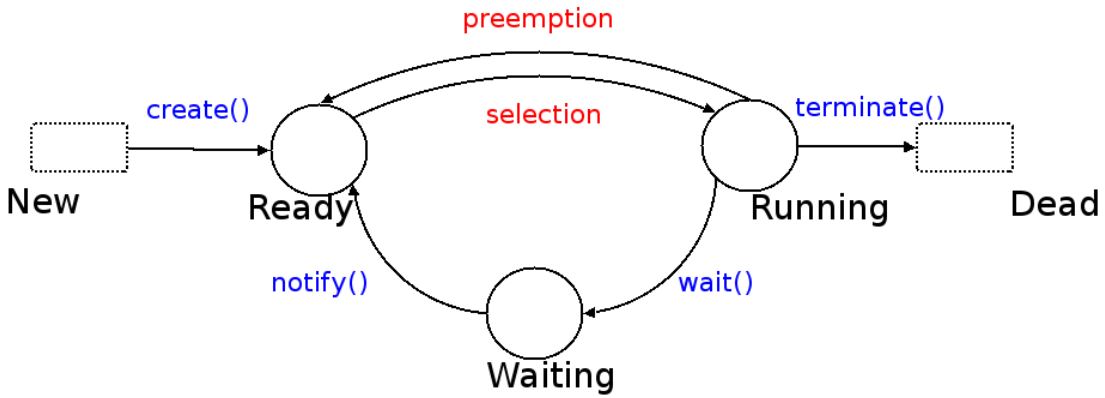
Esempio di cooperation synchronization

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)

Guardie

Guardia: costrutto che garantisce che una certa porzione di codice sia eseguita solo quando una certa condizione è verificata

- Può essere utilizzata per consentire l'accesso ad una porzione di codice ad un solo task per volta.

Semafori

Un *semaforo* (Dijkstra '65) è un'implementazione di una guardia

- **Semaforo:** struttura dati che consiste in
  - Un intero (`counter`)
  - Una coda di *descrittori di task*.
- **Descrittore di task:** struttura dati che memorizza tutte le informazioni relative allo stato di esecuzione di un task.

Semafori

Su un semaforo possono essere effettuate due operazioni:

- richiesta di accesso P() : se `counter > 0`, decrementa `counter` (e consenti l'accesso), altrimenti poni il task in stato waiting
- notifica di rilascio V() : se la coda dei descrittori è vuota, incrementa `counter`, altrimenti risveglia uno dei task nella coda dei descrittori (poni in coda ready)

Semafori

Su un semaforo possono essere effettuate due operazioni:

- richiesta di accesso P() : se `counter > 0`, decrementa `counter` (e consenti l'accesso), altrimenti poni il task in stato waiting
- notifica di rilascio V() : se la coda dei descrittori è vuota, incrementa `counter`, altrimenti risveglia uno dei task nella coda dei descrittori (poni in coda ready)
- Problema: come implemento effettivamente P() e V()?

Semafori

Su un semaforo possono essere effettuate due operazioni:

- richiesta di accesso P() : se `counter > 0`, decrementa `counter` (e consenti l'accesso), altrimenti poni il task in stato waiting
- notifica di rilascio V() : se la coda dei descrittori è vuota, incrementa `counter`, altrimenti risveglia uno dei task nella coda dei descrittori (poni in coda ready)
- Problema: come implemento effettivamente P() e V()?
- Soluzione: ho bisogno di *istruzioni atomiche* di tipo *test–and–set* (hardware o sistema operativo).

Semafori

Esempio: accesso mutuamente esclusivo ad una variabile condivisa utilizzando semafori
Esempio: cooperation synchronization su un buffer condiviso utilizzando semafori (pag 506 Sebesta)
Esempio: competition synchronization su un buffer condiviso utilizzando semafori (pag 507 Sebesta)

Valutazione

- semplice, potente e flessibile
- basso livello (un analogo del "goto" per programmi concorrenti)
- *unsafe* sia per cooperation che per competition
- non c'è modo di controllarne staticamente la correttezza (dipende interamente dalla semantica del programma)

Monitor

- idea: sfruttare i concetti di *data abstraction*
- *incapsulare* strutture dati (o *Abstract Data Types* - ADT) condivisi.
- simili strutture ("ADT sincronizzati") vengono detti **monitor**.
- in un monitor ci può essere **al più** un task attivo (ready o running)

Concurrent Pascal

Concurrent Pascal:

- task $\rightarrow$ `process` (che è un *tipo*)

```
type process_name = process (formal parameters)
   -- local declarations --
   -- process body --
end
```

- una variabile di tipo `process` può essere allocata staticamente o dinamicamente
- la dichiarazione crea semplicemente il codice, l'allocazione avviene tramite l'istruzione `init`

```
init process_variable_name (actual parameters)
```

ed il task diventa ready.

Concurrent Pascal

- monitor gestiti in modo simile ad ADT

```
type monitor_name = monitor (parametri formali)
   -- dichiarazione variabili condivise --
   -- definizione procedure ‘‘locali’’ (private) --
   -- definizione procedure ‘‘esportate’’ (public) --
   -- inizializzazione --
end
```

- procedure “esportate” sono identiche a quelle locali, ma marcate con la parola chiave `entry`

Synchronization

- Competition synchronization: *safe* ed automaticamente garantita dal monitor.
- Cooperation synchronization: deve essere gestita con costrutti a parte.
- Concurrent Pascal ha un tipo queue, che ammette due operazioni
  - `delay(queue)`: mette il chiamante in attesa sulla coda specificata e libera il monitor
  - `continue(queue)`: disconnette il chiamante dal monitor; se la coda specificata è non vuota, riattiva un processo della coda specificata (altrimenti non ha effetto)
- N.B. `delay()` e `continue()` hanno un comportamento *diverso* da P() e V().

Valutazione

- Meccanismo più rigido, ma più sicuro
- (meno parallelismo)
- Gestione safe di competition synchronization
- Stessi problemi di P() e V() per cooperation synchronization

Unità concorrenti in Java

- Le unità concorrenti in Java sono *metodi* chiamati **run**
- Le entità in cui girano i metodi run sono dei *thread* (a differenza dei task di ADA, che sono *processi*)
- Ci sono due modi per definire una classe con un metodo run:
  - Estendere la classe `Thread` (predefinita)
  - Implementare l'interfaccia `Runnable` (predefinita)

Definizione di Threads in Java

Esempio: `java_threads`

Metodi della classe Thread

- `sleep`: il thread rimane in stato waiting per il tempo specificato (in millisecondi) (*statico*)
- `yield`: "preemption" volontaria, il thread passa in stato ready (*statico*)
- `join`: l'esecuzione del thread è sospesa, in attesa del completamento dell'esecuzione di un altro thread

```
public void anyThreadMethod() {
   ...
   Thread myTh = new Thread();
   myTh.start();
   // computazione
   myTh.join(); // myTh.join(2000);
   // computazione
}
```

- `interrupt`: setta un bit, che può essere controllato con isInterrupted() (e solleva l'eccezione `InterruptedException`)
- `setPriority()`.e `getPriority()`
- `stop`, `suspend` e `resume` sono *deprecated*.

Competition synchronization

- Simile a monitors
- Ciascun metodo può essere marcato come **synchronized**.
- Ci può essere **al più un** thread attivo (ready o running).
- Ogni oggetto con metodi `synchronized` conserva una coda di thread in attesa di ottenere l'accesso.

```
class BufferManager {
   private int shared_resource;
   ...
   public synchronized void put(int item) { ... }
   public synchronized int get() { ... }
}
```

Cooperation synchronization

- Gestita con metodi `wait`, `notify` e `notifyAll` della classe `Object`.
- Questi metodi possono essere chiamati solo all'interno di metodi `synchronized` (perchè sfruttano il lock sull'oggetto)
- `wait`: da running a waiting
- `notify`: un thread tra quelli in coda (waiting) sull'oggetto passa da waiting a ready (non-deterministico)
- `notifyAll`: tutti i thread in coda (waiting) sull'oggetto passano da waiting a ready.
- N.B. Utilizzando `notifyAll` è spesso necessario ri-controllare le condizioni per cui il thread si era messo in attesa (attesa **semi-attiva**).

Valutazione

- Semplice, ma efficace.
- Thread (lightweight) rispetto ai Task (heavyweight) di ADA: non adatti ad ambiente distribuito (librerie!)
- Si possono simulare monitor e semafori

Sincronizzazione in ambiente distribuito

- Monitor funzionano bene se ho *memoria condivisa* (code ecc.)
- In ambiente distribuito, i monitor si adattano male
- è invece naturale progettare un meccanismo basato su *scambio di messaggi*

Guarded commands

- Proposito 'storico' (Dijkstra '75): programmazione che assicuri correttezza durante lo sviluppo anziché ricorrere a testing.
- Es: costrutto di selezione:

```
if <espr. booleana> -> <istruzione>
[] <espr. booleana> -> <istruzione>
[] ...
[] <espr. booleana> -> <istruzione>
fi
```

- [] è detto 'fatbar' e permette la concatenazione delle clausole
- semanticamente diverso dalla selezione multipla (case): *tutte* le espressioni booleane sono valutate simultaneamente durante l'esecuzione
- se più di una è vera, ne viene scelta una in modo *non-deterministico*, se *nessuna* è vera, c' è errore a runtime ed il programma è terminato

Nota

- in modo analogo, possono essere definiti *cicli*

```
do <espr. booleana> -> <istruzione>
[] <espr. booleana> -> <istruzione>
[] ...
[] <espr. booleana> -> <istruzione>
od
```

- La verifica della correttezza di un programma è più semplice se si utilizzano solo cicli e selezione, oppure solo guarded commands.

Scambio di messaggi sincrono

- *"... tasks are often busy and, when busy, are unwilling to be interrupted by other units."* (R. Sebesta)
- Ci sono due tipi di costrutti linguistici:
  - costrutti per l'*invio* di messaggi
  - costrutti per specificare che il ricevente è in *attesa* di ricevere messaggi
- lo scambio effettivo di informazioni è detto **rendezvous**, e può avvenire anche bidirezionalmente.

Esempio:

Tasks in ADA (`main_tasks.adb`)
Actor e Server in ADA (`consumerpackage, producerpackage`)

Accept con guardia

- Ogni `accept` può avere una clausola `when` associata (guardia).

```
when not Full ( Buffer ) =>
   accept Deposit ( New_Value ) do
```

- `accept` con `when` può essere **aperta** (se la condizione è vera) o **chiusa** (se la condizione è falsa)

Cooperation synchronization

- cooperation synchronization: può essere ottenuta combinando `select` e `accept` con guardia, ed utilizzando un task per la sincronizzazione: es put() e get()

```
loop
   select
      when Filled < Bufsize =>
         accept Put( v : in Integer) do
            ...
         end Put;
   or
      when Filled > 0 =>
         accept Get( v : out Integer) do
            ...
         end Get;
   end select;
end loop;
```

Competition synchronization

- competition synchronization:può essere ottenuta incapsulando le strutture dati a cui garantire l'accesso esclusivo all'interno di un task 'server':

```
task Buffer_Server is
   entry Put( v : in Integer);
   entry Get( v : out Integer );
end Buffer_Server;
task body Buffer_Server is
   size : constant Integer := 100;
   buffer : array (1 .. size) of Integer;
   filled : Integer range 0 .. size := 0;
   begin
      ...
   end Buffer_Server;
```

Esempio

Produttore e consumatore in ADA (Sebesta pagg. 520 – 521)
`ada_prod_cons`

Valutazione

- In ambiente distribuito, i monitor non possono essere gestiti in modo naturale, lo scambio di messaggi sì
- In ambiente non distribuito, lo scambio di messaggi è a metà strada tra monitor e semafori:
  - safe per competition synchronization
  - unsafe per cooperation synchronization (ma meno dipendente dal corretto utilizzo)

Scambio di messaggi asincrono

Buona programmazione concorrente significa anche sfruttare il parallelismo:

- Caso semplice di asincronia: accesso a procedure o funzioni non sincronizzate
- Altro caso di asincronia: un task lancia un altro task, che svolge un particolare calcolo indipendentemente
- Infine: l'esecuzione di un task può essere interrotta da un evento: **asynchronous select** in ADA
  Esempio pag. 524 Sebesta