Modelli di programmazione concorrente

Segmento del corso
Linguaggi di Programmazione

Tullio Vardanega, 2003

Ideologie a confronto - 1

- Premessa
  - Pochi linguaggi di grande diffusione offrono supporto diretto all'espressione di concorrenza
    - Concorrenza intesa come parallelismo potenziale
  - I modelli concorrenza possono essere generali oppure specializzati
    - Un dominio applicativo di crescente importanza è quello della programmazione *real-time*, che richiede un modello di concorrenza specializzato

Ideologie a confronto - 2

- Approcci possibili
  - Supporto alla rappresentazione diretta di concorrenza ortogonale alle altre funzionalità del linguaggio
    - Nessuna interazione diretta, né alcun vincolo imposto, per esempio, dal paradigma OOP adottato
  - Supporto completamente integrato con altre caratteristiche predominanti del linguaggio
    - Il modello di concorrenza non può essere disgiunto, per esempio, dal paradigma OOP adottato

Bibliografia

- *A Comparison of the Concurrency and Real-Time Features of Ada 95 and Java* Benjamin M Brosgol http://info.acm.org/sigada/conf/sa98/papers/brosgol.pdf
- E la relativa bibliografia …

Criteri di raffronto

- Gestione dei flussi di controllo
  - Dichiarazione, esecuzione, terminazione
- Mutua esclusione
- Sincronizzazione e comunicazione
  - A controllo sincrono, asincrono
- Sistemi a tempo reale
  - Predicibilità, politiche e meccanismi di ordinamento, granularità temporale
- Interazione con OOP
  - *Inheritance anomaly*

Gestione dei flussi di controllo - 1

- Dichiarazione (1/3)
  - Ada 95
    - Il `task` è sia unità di modularizzazione, pertanto dotata di specifica e realizzazione (*body*) separate, che entità (oggetto) a tempo di esecuzione
      - Il `task` non è però unità di compilazione, il che ne richiede l'incapsulazione in un `package` (od anche in una procedura)
    - Tale oggetto ha tipo (`task type`) che consente riferimenti (*access-to*) da usare per l'allocazione dinamica (via `new`) di oggetti `task`

Gestione dei flussi di controllo - 2

- Dichiarazione (2/3)
  - Java
    - Un flusso di controllo è istanza di una classe derivata dalla classe `Thread` predefinita e dotata di vari metodi utili
      - L'azione principale del flusso istanziato viene specificata per ridefinizione del metodo `run()`
    - La derivazione diretta esaurisce però l'ereditarietà singola consentita alla classe derivata!
    - Per questo motivo si dà un interfaccia astratto `Runnable` che una classe può implementare realizzandone il metodo `run()`
      - La stessa classe `Thread` implementa `Runnable` ed ha un costruttore che ne accetta una implementazione come parametro

Gestione dei flussi di controllo - 3

- Dichiarazione (3/3)
  - Primo raffronto
    - Il modello ad *heap* di Java non consente un analogo diretto della dichiarazione di tipo od oggetto `task`
    - Ada invece lo consente ed in modo compiutamente soggetto alle regole di visibilità proprie del modello a *stack* (ad ambienti gerarchici)
    - Implicazione: un ambito con flussi interni non può essere rimosso fin quando essi non siano terminati (rischio di *dangling references*)
      - Ada effettua controllo a tempo di esecuzione (con costo!)
      - Java deve trattenere i dati del metodo contenitore visibili ai flussi interni nell'equivalente artificiale di un ambito (*stack frame*)

Gestione dei flussi di controllo - 4

- Esecuzione (1/2)
  - Ada 95
    - La dichiarazione di un oggetto `task` dà luogo a 3 fasi successive e distinte
      - Creazione (esplicita): costruisce l'oggetto nel suo ambito contenitore
      - Attivazione (implicita): elabora la parte dichiarativa dell'oggetto `task` (le sue variabili locali)
      - Esecuzione (implicita): attiva un flusso di controllo concorrente che esegue i comandi della parte realizzativa (*body*) dell'oggetto
    - L'allocazione mediante `new` le racchiude in 1 sola fase

Gestione dei flussi di controllo - 5

- Ada 95 (continua …)
  - Consente di passare parametri di inizializzazione ad oggetti `task` con 2 modalità
    - Mediante parametri discriminanti specificati dal `task type` corrispondente
      - I cui valori sono però equiparati a costanti, e pertanto non modificabili (a meno che non siano riferimenti)
    - Tramite comunicazione esplicita e sincronizzata (bloccante) al debutto dell'esecuzione
      - Il che comporta chiara visibilità a livello di programma (nessun effetto implicito)

Gestione dei flussi di controllo - 6

- Esecuzione (2/2)
  - Java
    - Comporta 2 fasi entrambe esplicite
      - Creazione: costruisce un oggetto derivato da `Thread` in corrispondenza della sua istanziazione tramite `new`
      - Esecuzione: attiva il flusso di controllo corrispondente in seguito all'invocazione del metodo `start()` sull'oggetto
    - Rendere esplicito l'avvio dell'esecuzione espone la correttezza del programma a rischio di omissioni od errori del programmatore
      - Ciò però consente di invocare metodi speciali su oggetti `Thread` creati ma non ancora attivati (*daemon thread*)

Gestione dei flussi di controllo - 7

- Java (continua …)
  - Il passaggio di parametri non può avvenire tramite ridefinizione del metodo `start()`, né il profilo (*signature*) di `run()` è modificabile
  - Un costruttore allora deve passare implicitamente a `run()` i parametri necessari, copiandoli in istanze locali che possano essere riferite e manipolate
    - Modalità più potente di quella Ada tramite discriminanti, perché il costruttore è un vero e proprio sottoprogramma che ammette *overloading*, ma più oscura ed esposta ad errori

Gestione dei flussi di controllo - 8

- Terminazione di se (1/2)
  - Ada 95
    - Il `task` attivo, la cui esecuzione è autonoma, ha terminazione implicita (detta completamento) al raggiungimento dell'`end` più esterno, ma condizionata alla terminazione di ogni eventuale `task` interno al proprio ambito dichiarativo
    - Il `task` passivo, la cui esecuzione è condizionata alla sincronizzazione con un cliente esterno, ha terminazione implicita, ma indotta da un'alternativa `terminate` alla sincronizzazione

Gestione dei flussi di controllo - 9

- Terminazione di se (2/2)
  - Java
    - L'oggetto `Thread` termina quando `run()` esegue `return` od incontra la sua `}` più esterna
      - Nessuna distinzione tra completamento e terminazione
    - L'intera applicazione termina però solo con la terminazione forzata di tutte le *daemon thread* (p.es. il *garbage collector*), dopo la terminazione delle istanze `Thread` normali
      - Forzare la terminazione immediate senza accertare che lo stato corrente del *daemon* lo consenta espone il sistema al rischio di inconsistenze

Mutua esclusione - 1

- Meccanismi di base (1/3)
  - Ada 95
    - Una variabile marcata con `pragma Volatile()` non ammette copie temporanee in *cache* o registri
      - Scritture e letture direttamente in memoria
    - Una variabile marcata con `pragma Atomic()` viene acceduta in modo indivisibile
      - Con in più le stesse proprietà di una variabile `volatile`
    - Un tipo od oggetto `protected` consente letture simultanee e scritture in mutua esclusione

Mutua esclusione - 2

- Meccanismi di base (2/3)
  - Java
    - Una variabile marcata `volatile` assume le stesse proprietà di una variabile Ada marcata con `pragma Atomic()`
      - Variabili condivise non marcate consentono copie locali che possono evolvere in modo desincronizzato
    - Un metodo (o blocco) marcato `synchronized` associa possesso esclusivo al corrispondente oggetto
      - Secondo il modello classico del *monitor* con variabile lucchetto intero ≥ 0 acceduta tramite *test-and-set* atomico
        - L'invocazione di `sleep()` non comporta rilascio del lucchetto (indesiderabile latenza), mentre quella di `wait()` lo comporta (con effetti non ovvi sulla consistenza interna dell'oggetto)
      - L'attesa di accesso non è regolata da una coda ordinata, né lo è il meccanismo di rilevazione di lucchetto libero

Mutua esclusione - 3

- Meccanismi di base (3/3)
  - Primo raffronto
    - Il meccanismo Java di blocco di un oggetto garantisce scarsa integrità
      - Nessuna distinzione tra accessi potenzialmente concorrenti (lettura), accessi in mutua esclusione ed accessi anche condizionati allo stato interno dell'oggetto
      - La mutua esclusione non è neppure garantita se l'oggetto presenta metodi non marcati `synchronized` e non riguarda eventuali variabili marcate `static` nella classe

Mutua esclusione - 4

- Primo raffronto (continua …)
  - Come il *monitor* classico consente blocco circolare, così fanno metodi `synchronized` mutuamente dipendenti in Java
    - Ciò è strutturalmente evitato in Ada 95 dalla politica detta *Ceiling Locking*
  - Il modello Java richiede gestione esplicita, mediante uso di `wait()` e `notify()`, delle situazioni che dipendono dallo stato dell'oggetto
    - Maggiore esposizione ad errori di programmazione

Sincronizzazione e comunicazione - 1

- A controllo sincrono (1/3)
  - Ada 95
    - Con scambio dati
      - *Rendezvous* tra oggetti `task`
      - accesso esclusivo condizionato a risorsa condivisa (`protected entry`)
    - Senza scambio dati
      - Sospensione volontaria di oggetto `task` su `Suspension_Object` (basso livello, basso onere) con sblocco comandato da altro `task`

Sincronizzazione e comunicazione - 2

- A controllo sincrono (2/3)
  - Java
    - Senza scambio dati
      - Meccanismo a segnali ereditati dalla classe primordiale `Object`
      - `wait()` per richiedere sospensione fintantoché l'oggetto riferito ha stato inidoneo al proseguimento dell'operazione
        - Comporta il rilascio di ogni lucchetto posseduto dall'invocante
      - `notify()` per risvegliare uno qualunque tra gli oggetti `Thread` sospesi intorno all'oggetto condiviso
      - `notifyAll()` per risvegliare tutti gli oggetti `Thread` attualmente sospesi intorno all'oggetto condiviso

Sincronizzazione e comunicazione - 3

- A controllo sincrono (3/3)
  - Primo raffronto
    - Lo scambio di dati esplicito è preferibile (più sicuro)
    - L' accodamento esplicito e congruente (per condizione) all'esterno di un oggetto condiviso consente di attuare politiche specifiche di risveglio
    - Meccanismi di sincronizzazione ad alto livello sono preferibili (semantica standard → comprensione e manutenzione più agevole)

Sincronizzazione e comunicazione - 4

- A controllo asincrono (1/2)
  - Ada 95
    - Un'entità esterna può arrestare (`Hold`) e riavviare (`Continue`) l'esecuzione di un oggetto `task`
    - Un oggetto `task` può accettare notifica di un evento asincrono abbandonando la propria attività principale
    - A programma, una variabile condivisa marcata `pragma Atomic` può servire come veicolo di notifica asincrona (ma con latenza di accettazione)

Sincronizzazione e comunicazione - 5

- A controllo asincrono (2/2)
  - Java
    - Nessun meccanismo di linguaggio, solo mediante soluzioni a programma
    - Alcune particolarissime eccezioni sono definite come asincrone
      - Inconsueto!

Sistemi a tempo reale - 1

- Predicibilità spaziale e temporale
  - Vi è inevitabile conflitto tra la programmazione *general-purpose* e quella specializzata
  - Ada 95
    - La scalabilità del linguaggio consente di definirne sottoinsiemi ristretti privi di effetti dannosi
  - Java
    - Linguaggio "integralista" con caratteristiche inerenti ed inamovibili (metodi virtuali, modello ad *heap*, *garbage collector*, …)

Sistemi a tempo reale - 2

- Ordinamento
  - Ada 95
    - Intervalli distinti di priorità (≥ 30), assegnabili anche dinamicamente ed interrogabili
    - Distinzione semantica tra priorità base ed attiva
    - Politiche selezionabili mediante `pragma` → semantica data!
  - Java
    - Priorità ad intervallo unico (1-10), sia assegnabili che interrogabili
    - Trattamento discrezionale
      - Dipendente dall'implementazione della JVM → non portabile!

Sistemi a tempo reale - 3

- Granularità temporale (1/2)
  - Ada 95
    - Accesso a valori di orologio di grana grossa (`Ada.Calendar`) o fine (`Ada.Real_Time`, con garanzia di monotonicità)
    - Sospensioni temporali relative (`delay`) o assolute (`delay until`)
    - *Time-out* programmabile in caso di sincronizzazioni "aperte"

Sistemi a tempo reale - 4

- Granularità temporale (2/2)
  - Java
    - Accesso a valori di orologio di grana variabile (classi e metodi predefiniti ed estendibili, ma senza semantica garantita dall'implementazione)
    - Sospensioni temporali solo relative con `sleep()`
      - Prerilascio può intercorrere tra la richiesta e l'esecuzione → effetto diverso dall'atteso
    - *Time-out* su `wait()` ed altri metodi

Interazione con OOP - 1

- *Inheritance anomaly* (1/3)
  - L'interazione tra concorrenza ed OOP può produrre effetti indesiderabili
  - Metodi che contengono codice di sincronizzazione (per esecuzione in mutua esclusione, per sospensione o risveglio condizionali) possono non essere ereditabili senza analisi e/o modifiche della classe padre!
    - S. Matsuoka and A. Yonezawa *Analysis of inheritance anomaly in object-oriented concurrent programming languages* In: G. Agha, P. Wegner, and A. Yonezawa (editors), **Research directions in concurrent object-oriented programming**, pp. 107-150. MIT Press, 1993
  - Tale eventualità viola il principio di incapsulazione

Interazione con OOP - 2

- *Inheritance anomaly* (2/3)
  - Situazioni canoniche di sincronizzazione
    - Esecuzione condizionale
      - Le operazioni effettuabili possono dipendere da:
        - Stato interno della risorsa
        - Sequenza di operazioni precedenti (storia)
        - Parametri della richiesta
      - La risorsa può necessitare accesso esclusivo
    - Controllo di ordinamento
      - Le richieste non immediatamente soddisfacibili possono venire accodate

Interazione con OOP - 3

- *Inheritance anomaly* (3/3)
  - L' anomalia ha tipicamente luogo all'introduzione di:
    - Ulteriore partizionamento degli stati
      - L'estensione effettua una diversa ripartizione degli stati interni accettabili per specifiche operazioni
    - Modifica degli stati
      - L'estensione aggiunge nuovi stati interni, ignoti e non trattati dalla classe originaria
    - Dipendenza dalla storia di esecuzione
      - L'estensione impone vincoli sulle sequenze accettabili di richieste di operazione sulla risorsa
  - Che comportano conoscenza e modifica della logica interna complessiva della classe originaria

Interazione con OOP - 4

- Esempio
  - *Bounded buffer* con `get()`e `put()`
    - Non vuoto → `get()`
    - Non pieno → `put()`
  - Partizionamento degli stati
    - Prelievo doppio `get2()` → con almeno 2 articoli → stato non primitivo → modifica di classe originaria
  - Modifica degli stati
    - Stato inaccessibile `lock()` → `get()`e `put()` non ammesse → stato non primitivo → modifica di classe originaria
    - Stato accessibile `unlock()` → ammesse
  - Dipendenza dalla storia
    - Preleva 1 articolo ogni N immissioni `Nget()` → stato non primitivo → modifica di classe originaria

Interazione con OOP - 5

- Ada 95
  - Modello di concorrenza e paradigma OOP sono ortogonali
    - Meno interazioni implicite
    - Complessivamente minor potenza espressiva
      CP + OOP < COOP
- Java
  - Paradigma OOP dominante
    - Grande esposizione all'anomalia
    - *Deadlock* certo nel caso classico dei *monitor* annidati