Università degli Studi di Padova

Comunicazione tra processi

SCD

Anno accademico 2003/4
Corso di Sistemi Concorrenti e Distribuiti

Tullio Vardanega, tullio.vardanega@math.unipd.it

Premesse - 1

- I processi di un sistema concorrente sono raramente indipendenti
- La definizione delle interfacce tra le entità concorrenti (attive e reattive) di un sistema è attività fondamentale
- Occorre far riferimento ad un modello di comunicazione

Premesse - 2

- Un modello di comunicazione può prevedere (prima approssimazione)
  - Comunicazioni dirette tra entità attive
  - Comunicazioni indirette attraverso entità reattive di tipo passivo o protetto
- Forme "classiche" di comunicazione
  - Variabili condivise
    - Forma inadeguata nella forma priva di agenti di controllo (tramite entità passiva)
    - Se l'assegnamento non è indivisibile, l'integrità dello scambio non può essere garantita
  - Scambio messaggi

Esempio

```
// il processo A deve accedere ad X
// Prima pero' deve verificarne
// lo stato di libero
if (lock == 0) {
// X è già in uso
// Occorre ritentare il test
}
else {
// X è libera, allora va bloccata
   lock = 0;
// e nuovamente liberata dopo l'uso
   lock = 1;
}
```

```
// il processo B deve accedere ad X
// Prima pero' deve verificarne
// lo stato di libero
if (lock == 0) {
// X è già in uso
// Occorre ritentare il test
}
else {
// X è libera, allora va bloccata
   lock = 0;
// e nuovamente liberata dopo l'uso
   lock = 1;
}
```

Questo approccio è fallimentare! Perché?

Sincronizzazione - 1

- Mutua esclusione
  - Assicura che, ad ogni instante, non più di un processo abbia possesso di una risorsa (fisica o logica) condivisa
  - La sequenza di azioni che opera sulla risorsa è detta, in questo caso, sezione critica
- Sincronizzazione condizionale
  - Impone condizioni (logiche, di stato interno) che devono valere all'atto della sincronizzazione
    - Problema classico: il *buffer* finito condiviso nel modello produttore-consumatore

Sincronizzazione - 2

- L'uso di sincronizzazione espone a problemi delicati
  - Non risolvibili a priori nel progetto di un linguaggio concorrente!
  - Riconoscibili e risolvibili solo nel progetto di un sistema concorrente
- Stallo (*deadlock*)
- Accodamento potenzialmente infinito (*lockout, starvation*)

Università degli Studi di Padova

Comunicazione tra processi

Sincronizzazione - 3

- **Stallo (1/2)**
  - **Impedisce di proseguire a tutti i processi coinvolti**
  - **Richiede 4 ben riconoscibili precondizioni per aver luogo**
    - **Mutua esclusione**
      (già definita)
    - **Cumulazione di risorse**
      I processi possono accumulare risorse e trattenerle mentre attendono di acquisirne altre
    - **Assenza di prerilascio**
      Le risorse vengono rilasciate solo volontariamente
    - **Attesa circolare**
      Un processo attende una risorsa in possesso del successivo processo in catena

Università degli Studi di Padova

Comunicazione tra processi

Sincronizzazione - 4

- **Stallo (2/2)**
  - **4 possibili strategie per affrontare il problema**
    - **Indifferenza**
      Assumere (erroneamente?) che il problema non si possa verificare
    - **Prevenzione statica**
      Richiede accertamento che progetto e realizzazione del sistema siano strutturalmente liberi da condizioni di rischio
    - **Prevenzione dinamica**
      Comporta analisi dello stato di esecuzione presente e futuro del sistema per evitare esso possa entrare in una condizione di stallo (può bastare impedire strutturalmente il verificarsi di almeno una delle 4 precondizioni?)
    - **Rilevazione e trattamento**
      Comporta capacità di riconoscimento del verificarsi del problema e richiede meccanismi complessi per ripristinare uno stato noto e sicuro

Università degli Studi di Padova

Comunicazione tra processi

Sincronizzazione - 5

- **Accodamento potenzialmente infinito**
  - **Non si verifica in presenza di politica di accodamento FIFO**
  - **Si può verificare in presenza di qualunque altra politica**
    - A priorità
    - LIFO
    - Ad urgenza
- **La condizione di libertà da questo problema viene detta *fairness* (o *liveness*)**
  - **Tutti i processi hanno, nel tempo, uguali opportunità di progredire**
  - **L'attesa attiva è incompatibile con l'accertamento di *fairness*!**

Università degli Studi di Padova

Comunicazione tra processi

Requisiti minimi

- **Per l'implementazione di un linguaggio concorrente abbiamo bisogno di**
  - **Un meccanismo pratico per evitare l'uso di attesa attiva**
  - **Forme di accodamento di processi (in relazione a comunicazione e sincronizzazione!) che diano garanzie di *fairness***
- **La correttezza funzionale del programma non deve dipendere da queste forme e meccanismi!**

Università degli Studi di Padova

Comunicazione tra processi

Esempi classici

- **I filosofi a cena**
  - **Specifica del problema: N filosofi sono seduti ad un tavolo circolare. Ciascuno ha davanti a se un piatto di spaghetti ed una posata alla propria destra. Ciascun filosofo necessita di due posate per mangiare. L'attività di ciascun filosofo alterna pasti a momenti di riflessione**
- **Il barbiere che dorme**
  - **Materiale di esercitazione**

Università degli Studi di Padova

Comunicazione tra processi

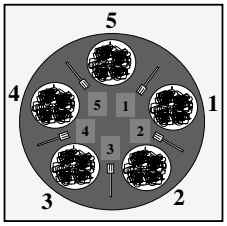
I filosofi a cena - 1

- **Situazione artificiosa, utile per illustrare importanti aspetti progettuali di concorrenza**
  - **Condivisione risorse → le posate**
  - **Mutua esclusione → il possesso di ciascuna posata**
  - **Scambio dati tra processi → diretto tra entità attive o mediato tramite entità reattive**
  - **Implementazione delle risorse → posate come risorse passive, risorse protette oppure *server***
  - **Sincronizzazione condizionale → un filosofo non può mangiare fin quando non ha 2 posate**
  - **Rischio di stallo → vedi riquadro seguente**

Università degli Studi di Padova

Comunicazione tra processi

I filosofi a cena - 2



Una non-soluzione

```
void filosofo_i(){
  while (True) {
    // medita
    prendi_forchetta(i);
    prendi_forchetta((i++)%5);
    // mangia
    posa_forchetta(i);
    posa_forchetta((i++)%5);
  }
}
```

Questo approccio incorre in tutte e 4 le precondizioni di stallo

Università degli Studi di Padova

Comunicazione tra processi

Possibili implementazioni - 1

- **Mutua esclusione con variabili condivise**

```
var flag: array [0..1] of boolean;
turn: 0..1;
repeat
      flag [i] := true;
      while flag [j] do
            if turn = j then
            begin
                  flag [i] := false;
                  while turn = j do no-op;
                  flag [i] := true;
            end;
            critical section
      turn := j;
      flag [i] := false;
      remainder section
until false;
```

Algoritmo concepito da Dekker ed applicato alle sezioni critiche da Edsger Wybe Dijkstra:
"flag" ← 1 indica l'intenzione di entrare in sezione critica
"turn" consente di arbitrare l'accesso tra i processi.
Applica a 2 processi
i ← 0
j ← 1

Università degli Studi di Padova

Comunicazione tra processi

Possibili implementazioni - 2

- **L'algoritmo di Dekker**
  - **Quando P0 vuole entrare in sezione critica lo annuncia a P1 ponendo a 1 la sua variabile di controllo (**flag[0] ← 1**)**
  - **Se P1 ne è già uscito (***remainder section***), la sua variabile di controllo varrà** 0**, pertanto P1 può entrarvi**
  - **Altrimenti occorre usare un'ulteriore variabile di controllo (**turn**) per arbitrare tra i 2 processi**
  - **Se** turn **indica P1 (**turn = 1**) allora P0 rinuncia alla competizione (**flag[0] ← 0**) e si pone in attesa attiva che P1 abbia finito**
- **Applicabile, ma molto pesante, a > 2 processi**

Università degli Studi di Padova

Comunicazione tra processi

Esercizio 1

- **Applicare, usando un linguaggio a piacere, l'algoritmo di Dekker ad un problema concorrente concreto, verificandone il funzionamento e/o gli eventuali problemi, prima nel caso di 2 processi e, poi, con più processi**
  - **Usare come esempio il problema dei filosofi a cena, partendo dal caso semplice con 2 soli processi "filosofo"**

Università degli Studi di Padova

Comunicazione tra processi

I filosofi a cena - 3

Soluzione con semafori senza stallo

```
void filosofo_i(){
  while (True) {
    // medita
    P(mutex);
    P(f[i]);
    P(f[(i+1)%5]);
    V(mutex);
    // mangia
    V(f[i]);
    V(f[(i+1)%5]);
  }
}
```

Utilizziamo un semaforo per ogni forchetta (`f[1..5]`) ed un semaforo per la mutua esclusione (`mutex`) al momento del prelievo delle forchette necessarie. Così più filosofi possono cenare simultaneamente

Università degli Studi di Padova

Comunicazione tra processi

Esercizio - 2

- **Dato il modello di codice nel riquadro precedente, indicare**
  - **Pregi e difetti della soluzione delineata**
  - **Se sia possibile una soluzione che preveda l'utilizzo del solo semaforo** `mutex` **(dunque senza un semaforo per ciascuna posata)**
  - **Se sia possibile il caso in cui il processo i-esimo si impossessi del semaforo mutex ma trovi una delle due forchette a lui adiacenti occupate, indicando le conseguenze di tale eventualità**

Università degli Studi di Padova

Comunicazione tra processi

Possibili implementazioni - 3

- **Mutua esclusione mediante semafori**
  - **L'implementazione deve garantire che le operazioni sul semaforo (storicamente dette P e V) siano indivisibili**
  - **All'attesa sul semaforo deve essere associata una forma di accodamento del processo fino alla condizione di rilascio**
  - **Il semaforo ha un valore di inizializzazione ed un valore massimo fissato**
    - Il valore iniziale, a risorsa libera, deve consentire il primo accesso
    - Il valore massimo indica il grado di concorrenza consentito dalla risorsa (**1 → mutua esclusione** | > 1 → più accessi simultanei)
  - **Questa tecnica dipende troppo dalla disciplina del programmatore**

Università degli Studi di Padova

Comunicazione tra processi

Possibili implementazioni - 4

- **Mutua esclusione mediante *monitor***
  - **La struttura *monitor* incapsula la risorsa condivisa e le operazioni che agiscono su di essa (dunque anche le corrispondenti sezione critiche)**
    - 1974, Charles A R Hoare, "**Monitors – An Operating System Structuring Concept**", Communications of the ACM vol. 17, pp. 549-557 + Erratum in CACM vol. 18, p. 95.
  - **La risorsa ed il suo stato sono nascosti alla vista dei processi utente**
  - **Il *monitor* esercita controllo sull'esecuzione delle operazioni invocate dall'esterno**
  - **Il compilatore (non il programmatore!) inserisce il codice necessario al controllo degli accessi**

Università degli Studi di Padova

Comunicazione tra processi

Possibili implementazioni - 5

- **Sincronizzazione condizionale mediante *monitor***
  - **Riesce a rappresentare ed a controllare condizioni logiche di accesso più sofisticate della mutua esclusione**
    - Mediante variabile di condizione o segnale
  - **Wait (var_condizione) → forza l'attesa del chiamante**
  - **Signal (var_condizione) → risveglia il processo in attesa**
    - Il segnale di risveglio (Signal) non ha memoria, va perso se nessuno lo attende

Università degli Studi di Padova

Comunicazione tra processi

Il costrutto monitor - 1

```
monitor PC
   condition non-vuoto, non-pieno;
   integer contenuto;
   procedure inserisci(prod : integer);
   begin
      if contenuto = N then wait(non-pieno);
      <inserisci nel contenitore>;
      contenuto := contenuto + 1;
      if contenuto = 1 then signal(non-vuoto);
   end;
   function preleva : integer;
   begin
      if contenuto = 0 then wait(non-vuoto);
      preleva := <preleva dal contenitore>;
      contenuto :=  contenuto - 1;
      if contenuto = N-1 then signal(non-pieno);
   end;
   contenuto := 0;
end monitor;
```

```
procedure Produttore;
begin
   while true do
   begin
      prod := produci;
      PC.inserisci(prod);
   end;
end;
```

```
procedure Consumatore;
begin
   while true do
   begin
      prod := PC.preleva;
      consuma(prod);
   end;
end;
```


Università degli Studi di Padova

Comunicazione tra processi

Il costrutto monitor - 2

- **Wait blocca il chiamante quando le condizioni logiche della risorsa non consentono di procedere**
  - **Contenitore pieno (produttore), contenitore vuoto (consumatore)**
- **Signal notifica il verificarsi della condizione attesa al primo processo bloccato, risvegliandolo**
  - **Il processo risvegliato compete per l'esecuzione con il chiamante della Signal**
- **Wait e Signal sono invocate in mutua esclusione**
  - **Questo esclude il verificarsi di *race condition***

Università degli Studi di Padova

Comunicazione tra processi

Il costrutto monitor - 3

- **Java approssima il monitor, tramite classi con metodi** synchronized**, ma <u>senza</u> variabili di condizione**
  - **L'attesa è sull'intero oggetto, non su una condizione**
- **I metodi** wait() **e** notify() **invocati <u>esplicitamente</u> all'interno di metodi** synchronized **realizzano le condizioni evitando il verificarsi di *race condition***
  - notify() **risveglia <u>arbitrariamente</u> uno dei processi in attesa sull'oggetto**
  - wait() **può venire interrotto e l'interruzione va trattata come eccezione**

Università degli Studi di Padova

Comunicazione tra processi

Il costrutto monitor - 4

```
class monitor{
   private int contenuto = 0;
   public synchronized void inserisci(int prod){
      if (contenuto == N) blocca();
      <inserisci nel contenitore>;
      contenuto = contenuto + 1;
      if (contenuto == 1) notify();
   }
   public synchronized int preleva(){
      int prod;
      if (contenuto == 0) blocca();
      prod = <preleva dal contenitore>;
      contenuto = contenuto - 1;
      if (contenuto == N-1) notify();
      return prod;
   }
   private void blocca(){
      try{wait();
      } catch(InterruptedException exc) {};}
}
```

```
static final int N = <…>;
static monitor PC = new monitor();
// …
PC.inserisci(prod); // produttore
// …
prod = PC.preleva(); // consumatore
```

Attesa e notifica sono responsabilità del programmatore!

Università degli Studi di Padova

Comunicazione tra processi

Esercizio 3

- **Dato il funzionamento di `wait()` e `notify()` in Java si verifichi se**
  - **Il codice mostrato nel riquadro precedenti emuli correttamente il funzionamento di un monitor nel caso di 1 produttore ed 1 consumatore**
  - **Ciò si verifichi anche (o solo) nel caso di produttori e consumatori multipli**

Università degli Studi di Padova

Comunicazione tra processi

Il costrutto monitor - 5

- **Il monitor separa sincronizzazione da scambio (condivisione) dati**
- **Il monitor non è però capace di controllare dinamicamente l'ordine degli accessi**
- **Il monitor ha bisogno di meccanismi supplementari (wait & signal) per realizzare sincronizzazione condizionale**

Università degli Studi di Padova

Comunicazione tra processi

Scambio messaggi - 1

- **Comporta sincronizzazione (conoscenza dello stato dell'altro) solo nel modello sincrono ☺**
  - **Il mittente ed il destinatario si attendono reciprocamente, procedendo solo a scambio avvenuto**
- **Nella forma asincrona, il mittente non attende il destinatario**
  - **Il destinatario non viene così a conoscere lo stato attuale del mittente**

Università degli Studi di Padova

Comunicazione tra processi

Scambio messaggi - 2

- **Mittente e destinatario devono conoscersi per indirizzarsi?**
  - **Nomi unici**
    - Di processo; di casella postale; di porta; di canale
  - **Tipo di messaggio (a destinazione)**
- **La forma sincrona con uso di nomi unici è detta *rendezvous***
  - **Processo A :: B ! Msg** **Processo B :: A ? Msg**
  - **In CSP questa forma è unidirezionale**
- **All'attesa di un [tipo di] messaggio può essere necessaria una precondizione**
  - **E W Dijkstra ha proposto che i comandi di ricezione siano selettivi nondeterministicamente e dotati di "guardia"**
  - **1975, "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs", CACM, vol. 18(8), pp. 453-457**

Università degli Studi di Padova

Comunicazione tra processi

Scambio messaggi - 3

- **La "guardia" è una espressione Booleana la cui verità abilita l'esecuzione del comando ad essa associato**

```
select
   Guard_1 => Statement_1;
or
   Guard_2 => Statement_2;
or
   …
or
   Guard_N => Statement_N;
end select;
```

Se più di una guardia è vera simultaneamente, ne viene scelta una nondeterministicamente

Università degli Studi di Padova

Comunicazione tra processi

Scambio messaggi - 4

- **Lo scambio dati può utilmente essere bidirezionale senza richiedere 2 messaggi**
- **Il primo destinatario allora offre un "luogo di entrata" (entry) presso il quale ricevere parametri `in`, elaborarli e restituire valori su parametri `out`**

```
Guard =>
   accept Service ( in … out …) do
      …
   end;
```

- **Con questa forma l'invocante nomina il destinatario ed il servizio (= canale) ma non viceversa**
  - **Asimmetria di denominazione**

Università degli Studi di Padova

Comunicazione tra processi

Scambio messaggi - 5

- **Forma sincrona e forma asincrona sono duali: l'una può essere utilizzata per ottenere l'altra**
  - **Sincrona → Asincrona**
    Introdurre un'entità intermedia tra mittente e destinatario rende la loro comunicazione asincrona
    - Al costo aggiuntivo dell'entità intermedia (minor costo se non attiva!)
  - **Asincrona → Sincrona**
    Accoppiare Send (conferma) a Receive dal lato destinatario e Receive a Send dal lato mittente forza sincronizzazione