Comunicazione tra processi

SCD

Anno accademico 2009/10
Sistemi Concorrenti e Distribuiti

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

Premesse – 1

- I processi di un sistema concorrente molto raramente sono indipendenti l'uno dall'altro
- La definizione delle interfacce tra le entità concorrenti di un sistema è problema serio
  - Interfaccia = contratto = relazione formale
- Dobbiamo far riferimento a un modello di comunicazione tra entità

Premesse – 2

- Un modello di comunicazione può prevedere
  - Comunicazioni dirette tra entità attive
  - Comunicazioni indirette tra entità attive attraverso entità reattive condivise e di tipo passivo o protetto
- Forme "classiche" di comunicazione
  - Scambio messaggi
  - Variabili condivise
    - Forma del tutto inadeguata se priva di agenti di controllo
    - Non garantisce l'integrità dello scambio dati se l'accesso non è indivisibile

Esempio

```
// il processo A deve accedere a X
// Prima però 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;
   < usa (X) >;
   // e nuovamente liberata dopo l'uso
   lock = 1;
}
```

```
// il processo B deve accedere a X
// Prima però 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;
   <usa (X)>;
   // e nuovamente liberata dopo l'uso
   lock = 1;
}
```

Questo approccio è fallimentare! Perché?

Sincronizzazione – 1

- Mutua esclusione (*exclusion synchronization*)
  - Assicura che, a 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 sezione critica
- Sincronizzazione condizionale (*avoidance synchronization*)
  - Impone condizioni (logiche, di stato interno) che devono valere all'atto della sincronizzazione
    - Problema classico: *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 generale
  - Riconoscibili e risolvibili nel progetto della singola applicazione concorrente
- Stallo (*deadlock*)
- Accodamento potenzialmente infinito (*lockout, starvation*)

Sincronizzazione – 3

- **Stallo (1/2)**
  - **Impedisce di proseguire a tutti i processi coinvolti**
  - **Richiede il verificarsi di 4 pre-condizioni**
    - **Mutua esclusione**
    - **Cumulazione di risorse (*hold-and-wait*)**
      I processi possono accumulare risorse
    - **Assenza di prerilascio**
      Le risorse vengono rilasciate solo volontariamente
    - **Attesa circolare**
      Un processo attende almeno una risorsa in possesso di un altro processo

Sincronizzazione – 4

- **Stallo (2/2)**
  - **4 strategie di trattamento del problema**
    - **Indifferenza**
      Assumere che il problema non possa verificarsi
    - **Prevenzione statica**
      Accertare che progetto **e** realizzazione del sistema siano strutturalmente liberi da condizioni di rischio
    - **Prevenzione dinamica**
      Analizzare lo stato di esecuzione presente e futuro del sistema per **evitare** che esso possa entrare in una condizione di stallo
      - Può bastare impedire strutturalmente il verificarsi di anche soltanto una delle pre-condizioni? Quale?
    - **Rilevazione e trattamento**
      Riconoscere il verificarsi del problema e utilizzare meccanismi complessi per **ripristinare** uno stato noto e sicuro

Sincronizzazione – 5

- **Accodamento potenzialmente infinito**
  - **Non si verifica in presenza di politica di accodamento FIFO**
  - **Si può verificare invece in presenza di qualunque altra politica**
    - A priorità
    - LIFO
    - A urgenza

- **La condizione di libertà da questo problema viene detta *fairness* (e *liveness*)**
  - **Tutti i processi hanno nel tempo uguali opportunità di progredire**
  - **L'attesa attiva è incompatibile con l'accertamento di *fairness***

Requisiti generali

- **Per realizzare un linguaggio concorrente servono**
  - **Meccanismi per evitare o minimizzare l'uso di attesa attiva**
    - In ambiente multi-processore si è costretti a usare *spin-locking*
  - **Forme di accodamento *fair* di processi**
    - Da usare per il supporto di comunicazione e sincronizzazione
    - Ove si possa stimare a priori la durata massima di attesa in coda si parla di ***bounded fairness***
- **Però la correttezza funzionale del programma non deve dipenderne!**

Requisiti di sincronizzazione

- **Una modalità di sincronizzazione è ammissibile se soddisfa 4 condizioni**
  1. **Garantire accesso esclusivo**
  2. **Garantire attesa finita**
  3. **Non fare assunzioni sull'ambiente di esecuzione**
  4. **Non subire condizionamenti dai processi esterni alla sezione critica**

Soluzioni "al limite" – 1

- **Mutua esclusione con variabili condivise e alternanza stretta tra coppie di processi**
  - **Tre difetti importanti**
    - Uso di attesa attiva (*busy wait*)
    - Violazione della condizione 4
    - Rischio di *race condition* sulla variabile di controllo

```
Processo 0 ::
while (TRUE) {
  while (turn != 0)
    /* busy wait */ ;
  critical_region();
  turn = 1;
  outside_cr();
}
```

Comando di alternanza

```
Processo 1 ::
while (TRUE) {
  while (turn != 1)
    /* busy wait */ ;
  critical_region();
  turn = 0;
  outside_cr();
}
```

Soluzioni "al limite" – 2

- **L'algoritmo di Dekker**

```
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 /= i do no-op;
                  flag [i] := true;
            end;
         end if;
      end while;
      critical section
      turn := j;
      flag [i] := false;
      remainder section
until false;
```

Attesa attiva!

Algoritmo concepito da T.J. Dekker e applicato alle sezioni critiche da E.W. 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
poi generalizzato da altri a N processi

Soluzioni "al limite" – 3

- **Proposta migliorativa (G.L. Peterson)**
  - **Applica anch'essa a coppie di processi**
    - Robusto rispetto alla *race condition* sul valore di `turn` ma solo in assenza di *cache*!
  - **Al processo `i` tocca attesa attiva fin quando `j` non sia uscito dalla sezione critica**

```
IN(int i) :: {
 int j = (i – 1); // l'altro
 flag[i] = TRUE;
 turn = i;
 while (flag[j] && turn == i)
 /* busy wait */ ;
}
OUT(int i) :: {
 flag[i] = FALSE;
}
```

Uscita dalla sezione critica

```
Processo (i = 0 / 1) ::
while (TRUE) {
  IN(i);
  /* sezione critica */
  OUT(i);
  /* altro lavoro    */
}
```


Soluzioni canoniche – 1

- **Mutua esclusione mediante semafori**
  - **Le operazioni sul semaforo devono essere indivisibili**
    - `V(S) ::= S.Sem := S.Sem + 1;`
    - `P(S) ::= if S.Sem = 0 then Suspend_til(S.Sem > 0); end if; S.Sem := S.Sem – 1;`
    - Con **S.Sem = 1** come valore iniziale si ha **mutua esclusione**
    - Con **S.Sem = N > 0** inizialmente si realizza un **semaforo contatore** dove N è il massimo grado di concorrenza interna consentito dalla risorsa
    - Con **S.Sem = 0** iniziale si ha rudimentale **sincronizzazione condizionale**
  - **All'attesa sul semaforo deve corrispondere accodamento del processo fino alla condizione (Suspend_til) di rilascio**
  - **Troppo dipendente dalla (in)disciplina del programmatore!**

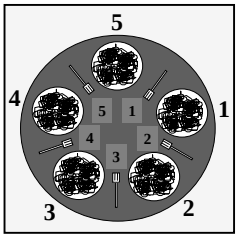
Un problema classico

- **I filosofi a cena**
  - **Specifica del problema: N filosofi sono seduti a un tavolo circolare. Ciascuno ha davanti a se cibo (spaghetti?) in un piatto, e una posata alla propria destra. Ogni filosofo usa due posate per mangiare. L'attività di ciascun filosofo alterna pasti a momenti di riflessione**

I filosofi a cena – 1

- **Situazione artificiosa ma utile per illustrare importanti problemi progettuali di concorrenza**
  - **Condivisione risorse → le posate**
  - **Mutua esclusione → pasto condizionato al possesso di 2 posate**
  - **Scambio dati tra processi → diretto tra entità attive o mediato tramite entità reattive (disponibilità delle risorse)**
  - **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 → ci sono tutte le condizioni necessarie e sufficienti**

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

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]) e un semaforo per la mutua esclusione (mutex) al momento del prelievo delle forchette necessarie, così che più filosofi possano cenare simultaneamente

Esercizio 1

- **Applicare al problema dei filosofi a cena una delle soluzioni "al limite"**
  - **Usando un linguaggio a piacere**
  - **Verificando il funzionamento e/o gli eventuali problemi della soluzione**
    - Prima al caso di 2 processi e, poi, con più processi

Esercizio – 2

- **Dato il modello di codice in diapositiva 18 indicare**
  - **Pregi e difetti della soluzione illustrata**
  - **Se esiste una soluzione che preveda l'utilizzo del solo semaforo mutex**
    - Senza impiegare un semaforo per ciascuna posata
  - **Se il processo `i` possa impossessarsi del semaforo mutex trovando una delle due forchette a lui adiacenti occupate**
    - Quali le conseguenze di tale eventualità?

Soluzioni canoniche – 2

- **Mutua esclusione mediante *monitor***
  - **La struttura *monitor* incapsula la risorsa condivisa e le operazioni che agiscono su di essa e 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 e il suo stato sono nascosti alla vista dei processi utente**
  - **Il *monitor* esercita controllo sull'esecuzione delle operazioni invocate dall'esterno**
  - **Il compilatore (e non il programmatore!) inserisce il codice necessario al controllo degli accessi**

Soluzioni canoniche – 3

- **Sincronizzazione condizionale mediante *monitor***
  - **Riesce a rappresentare e 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**, dunque va perso se nessuno lo recepisce

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; // inizializzazione
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;
```

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 di Signal**
    - La convenzione più spesso usata è che Signal sia l'ultima azione eseguita in procedure di *monitor* (vero?)
- **Wait e Signal sono invocate in mutua esclusione**
  - **Questo esclude il verificarsi di *race condition***
- **Come si ottiene *avoidance synchronization* fuori dal *monitor*?**

Comunicazione tra processi

Il costrutto *monitor* – 3

- **Java approssima il *monitor* tramite classi con metodi synchronized ma senza variabili di condizione**
  - **L'attesa è sull'intero oggetto e non sulla condizione**
- **I metodi wait() e notify() invocati all'interno di metodi synchronized evitano *race condition***
  - **notify() risveglia uno qualsiasi dei processi in attesa sull'oggetto**
    - Sotto queste ipotesi notifyAll() semplifica la vita del programmatore!
  - **L'esecuzione di wait() può venire interrotta**
    - Il che va trattato esplicitamente come eccezione

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

Comunicazione tra processi

Esercizio 3

- **Dato il funzionamento di `wait()` e `notify()` in Java si verifichi se**
  - **Il codice mostrato nel riquadro di diapositiva 27 emuli correttamente il funzionamento di un *monitor* nel caso di 1 produttore e 1 consumatore**
  - **Ciò si verifichi anche nel caso di produttori e consumatori multipli**

Comunicazione tra processi

Il costrutto *monitor* – 5

- **Il *monitor* separa la sincronizzazione dalla condivisione di dati**
- **Il *monitor* non consente controllo dinamico sull'ordine degli accessi**
  - **Entra chi arriva primo anche se poi magari si deve sospendere**
- **Il *monitor* ha bisogno di meccanismi supplementari (wait & signal) per realizzare sincronizzazione condizionale**

Comunicazione tra processi

Scambio messaggi – 1

- **Comporta sincronizzazione solo nella sua variante sincrona**
  - **Sincronizzazione = conoscenza dello stato dell'altro**
  - **Mittente M e destinatario D si attendono reciprocamente**
    - Procedendo poi solo a scambio avvenuto
- **Nella forma asincrona M non attende D**
  - **D non viene così a conoscere lo stato corrente di M**

Scambio messaggi – 2

- M e D devono conoscersi per indirizzarsi?
  - Nomi unici
    - Di processo; di casella postale; di porta; di canale
  - Tipo di messaggio (da intendere a destinazione)
- Forma sincrona con nomi unici = *rendezvous*
  - Processo A :: B ! Msg Processo B :: A ? Msg
  - In CSP questa forma di comunicazione è unidirezionale
- Una precondizione può essere preposta all'attesa di un [tipo di] messaggio
  - Dijkstra ha proposto che i comandi di ricezione [alternativi] siano selettivi, nondeterministici e dotati di "guardia" (caso del *buffer*)
  - 1975, "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs", CACM, vol. 18(8), pp. 453-457

Scambio messaggi – 3

- La "guardia" è una espressione Booleana
- Se e quando vera abilita l'esecuzione del comando associato

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

Quando più guardie sono vere simultaneamente ne viene scelta una **non deterministicamente**

Scambio messaggi – 4

- Lo scambio dati può essere bidirezionale senza richiedere 2 messaggi (A/R)
- Il destinatario D offre un "luogo di entrata" (entry) presso il quale ricevere parametri `in`, elaborarli e restituirvi valori su parametri `out`

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

- In questo modo M nomina D e il servizio (= canale) senza che valga il viceversa
  - Asimmetria di denominazione

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 M e D produce comunicazione asincrona
    - Al costo aggiuntivo dell'entità intermedia (minor costo se non attiva!)
  - Asincrona → Sincrona
    Accoppiare Send (di conferma) a Receive dal lato D e Receive a Send dal lato M comporta sincronizzazione
    - Al costo di due comunicazioni