Comunicazione tra processi

SCD

Anno accademico 2016/17
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
  - Altrimenti il sistema sarebbe perfettamente parallelo
- La definizione delle interfacce tra le entità concorrenti di un sistema è fondamentale
  - Interfaccia ⇨ contratto ⇨ relazione formale
- Per realizzarle dobbiamo far riferimento a un modello di comunicazione tra entità

Premesse – 2

- Un modello di comunicazione sceglie tra
  - 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 → comunicazione diretta
  - Variabili condivise → comunicazione indiretta
    - Rischiosa se non mediata da agenti di controllo
    - L'assenza di garanzie di atomicità mette a rischio l'integrità dei dati

Variabili condivise – 1

- Condizione di Bernstein (IEEE TREC 15-5, 1966)
  - *Atomic execution is guaranteed if shared variables that are read and modified by a critical section are not modified by any other concurrently executing section of code*
  - La violazione da luogo al rischio di *data race*
  - R. Netzer e B. Miller (ACM LoPLAS 1-1, 1992) mostrano che il problema di verificare la presenza di *data race* in un programma è *NP-hard*

Variabili condivise – 2

- **Le parti di codice che agiscono su variabili condivise sono dette sezioni critiche**
- **La possibilità di accessi non ordinati a una risorsa condivisa viene detta *race condition***
  - **Tuttavia il non-determinismo di esecuzione può essere qualità desiderabile per un programma concorrente!**

Variabili condivise – 2

- **Vi sono due problemi complementari**
  1. **Come rendere atomiche le sezioni critiche**
     - Le *data race* a questo livello sono chiamate *low-level*
  2. **Come individuare bene le sezioni critiche**
     - Le *data race* a questo livello sono *high-level* e sono esposte a due rischi

  a. ***Non-atomic protection fault*** **: un processo accede più volte risorsa condivisa con operazioni parziali**

  b. ***Lost-update fault*** **: dipendenza funzionale tra una acquisizione in lettura e la successiva scrittura di una variabile condivisa da parte di un processo in presenza di contesa in scrittura da parte di un altro processo**

Problema 1.a: 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;
   <sezione critica S1 su 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;
   <sezione critica S2 su X>;
   // e nuovamente liberata dopo l'uso
   lock = 1;
}
```

Le sezioni critiche S1 e S2 **non** sono atomiche

Problema 1.b: esempio

```
/* DEPOSIT */
amount = read_amont();
lock(); /* primitiva ideale che
           apre in modo atomico
           la sezione critica */
balance = balance + amount;
interest = interest + rate *
           balance;
unlock(); /* primitiva ideale che
             rilascia la sezione
             critica */
```

```
/* WITHDRAW */
amount = read_amount();
if (balance < amount) {
   /* notifica l'utente che
      l'operazione è negata */
}
else {
   balance = balance – amount;
   interest = interest +
              rate * balance;
}
```

L'operazione Withdraw è esposta a ***low-level data race***

Problema 2.a: esempio

```
/* Updater Task */

// set status value reading
synchronized (table){
  table[N].value = V;
}

… // do work

// set system status for value N
synchronized (table) {
  table[N].achieved = true;
}
```

```
/* Monitor Daemon */

synchronized (table){
 if (table[N].achieved &&
   system_state[N] !=
     table[N].value){
    // inconsistent system state
    issueWarning();
   }
}
```

NASA
Remote Agent (1997)
con Java e LISP

Non-atomic protection fault

Problema 2.b: esempio

```
/* WITHDRAW */

void withdraw(int amount){
  lock(l);
  int tmp = balance;
  unlock(l);
  if tmp > amount){
    lock(l);
    balance = tmp - amount;
    unlock(l);
  }
}
```

```
/* DEPOSIT */

void deposit(int amount){
  lock(l);
  balance = balance + amount;
  unlock(l);
}
```

Lost-update fault

Sincronizzazione – 1

- **Mutua esclusione: *exclusion synchronization***
  - **A ogni instante, non più di un processo può avere possesso di una risorsa condivisa**
    - Garanzia di atomicità
- **Sincronizzazione condizionale: *avoidance synchronization***
  - **Certe pre-condizioni logiche devono valere all'atto della sincronizzazione**
    - Problema classico: *buffer* finito condiviso nel modello produttore-consumatore

Sincronizzazione – 2

- **L'uso di sincronizzazione espone a rischi**
  - **Non eliminabili a priori nel progetto di un linguaggio concorrente generale**
  - **Riconoscibili e risolvibili nel progetto del singolo sistema**
- **Stallo: *deadlock***
- **Accodamento potenzialmente infinito: *lockout, starvation***

Sincronizzazione – 3

- **Stallo (1/2)**
  - **Impedisce di proseguire a tutti i processi coinvolti**
  - **Richiede il verificarsi simultaneo di 4 pre-condizioni**
    1. **Mutua esclusione**
    2. **Accumulo di risorse (*hold-and-wait*)**
    3. **Risorse non prerilasciabili**
    4. **Attesa circolare**

Sincronizzazione – 4

- **Stallo (2/2)**
  - **4 strategie di trattamento del problema**
    1. **Indifferenza** : Sperare che il problema non si manifesti
    2. **Prevenzione statica** : Accertare che progetto e realizzazione del sistema siano liberi da condizioni di rischio
    3. **Prevenzione dinamica** : Analizzare lo stato di esecuzione presente e futuro del sistema per evitare che l'ingresso in condizione di stallo
    4. **Rilevazione e trattamento** : Riconoscere il verificarsi del problema e utilizzare servizi speciali per ripristinare uno stato noto e sicuro

Sincronizzazione – 5

- **Tecniche di prevenzione**
  - **Basta impedire il verificare di anche una sola delle 4 pre-condizioni (quale?)**
    - Tecniche, statiche o dinamiche, che impediscano l'accumulo di risorse
    - Tecniche, statiche o dinamiche, che impediscano il formarsi di attese circolari
- **Tecniche di trattamento**
  - ***Transactional memory* : uso di meccanismi di «*concurrency control*» analoghi a quelle delle basi di dati**
    - Le transazioni aggiungono *consistency (serializability)* e *isolation* ad atomicity

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 (importanza, caratteristica statica)
    - LIFO
    - A urgenza (scadenza, caratteristica dinamica)
- **La condizione di libertà da questo problema viene detta *fairness* (e garantisce *liveness*)**
  - **Tutti i processi hanno uguali opportunità di progredire**
  - **L'attesa attiva è incompatibile con la garanzia di *fairness***

Requisiti generali

- **A un linguaggio concorrente servono**
  - **Meccanismi per evitare o ridurre l'uso di attesa attiva**
    - Ma 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
    - Se si può stimare a priori la massima di attesa in coda si parla di ***bounded fairness***
- **Ma la correttezza funzionale del programma non deve dipenderne!**

Requisiti di sincronizzazione

- **Una modalità di sincronizzazione è accettabile se soddisfa 4 condizioni**
  1. **Garantire accesso esclusivo**
  2. **Garantire attesa finita**
  3. **Non fare assunzioni sull'ambiente di esecuzione**
  4. **Non subire condizionamenti dall'esterno della 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 a.k.a. *busy wait*
    - Violazione della condizione 4
    - Rischio di *data race* sulla variabile di controllo (ma con effetti non gravi)

```
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

- **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[i] ← True indica l'intenzione di **i** di entrare in sezione critica;
il valore di **turn** arbitra l'accesso tra i processi.
Algoritmo pensato per 2 processi.
Si può generalizzare a N>2 processi.

Soluzioni "al limite" – 3

- **Algoritmo di Peterson**
  - **Applica a coppie di processi**
  - **Ogni processo opera su un *flag* privato e una variabile condivisa**
    - Robusto rispetto alla *data race* sul valore della variabile condivisa ma solo in assenza di *cache* !
  - **Libero da *deadlock***
  - **Garantisce bounded fairness**
    - La prenotazione della sezione critica da parte del processo A da priorità al processo B

```
set my.flag
give coin to other
loop
 if (other.flag clear) continue
 if (coin is mine) continue
end loop
// CRITICAL SECTION
clear my.flag
```


Soluzioni canoniche – 1

```
typedef struct {
    int count;
    queue q; /* queue of threads waiting on this semaphore */
} Semaphore;

void P(Semaphore s)
{
    Disable interrupts;
    if (s->count > 0) {
        s->count -= 1;
        Enable interrupts;
        return;
    }
    Add(s->q, current_thread);
    sleep(); /* re-dispatch */
    Enable interrupts;
}

void V(Semaphore s)
{
    Disable interrupts;
    if (isEmpty(s->q)) {
        s->count += 1;
    } else {
        thread = RemoveFirst(s->q);
        wakeup(thread); /* put thread on the ready queue */
    }
    Enable interrupts;
}
```

Il valore di inizializzazione di `count` conta:
=1 → semaforo binario
>1 → semaforo contatore
=0 → sincronizzazione condizionale

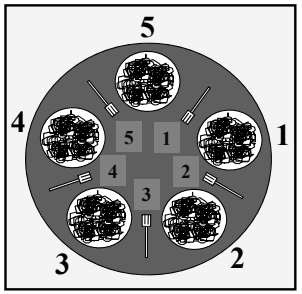
Ma l'uso di P() e V() è lasciato alla disciplina del programmatore

Esempio d'uso: i filosofi a cena – 1

- **Problemi progettuali**
  - **Condivisione di risorse → le posate**
  - **Mutua esclusione → pasto condizionato al possesso esclusivo di 2 posate**
  - **Scambio dati tra processi → informazioni su stato risorse**
  - **Realizzazione 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 → tutte le pre-condizioni soddisfatte**

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);
  }
}
```

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]);
  }
}
```

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

Soluzioni canoniche – 2

- **Mutua esclusione mediante *monitor***
  - **Il *monitor* incapsula la risorsa condivisa e le sue operazioni quindi anche le sezione critiche corrispondenti**
    - 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
  - **Risorsa e 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 con *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
    - Va perso se nessuno lo recepisce cioè non ha effetti di risveglio se nessun processo è in attesa su quella condizione

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;
```

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?)**

Il costrutto *monitor* – 4

- **L'invocazione di Wait e Signal è protetta dal compilatore con mutua esclusione**
  - **Questo esclude il rischio di *data race***
- **Problema**
  - **Come ottenere *avoidance synchronization* fuori dal codice algoritmico delle procedure di monitor?**

Il costrutto *monitor* – 5

- **Java approssima il monitor con metodi `synchronized` senza variabili di condizione**
  - **Attesa sull'intero oggetto e non sulla condizione**

- **`wait()` e `notify()` invocate all'interno di metodi synchronized evitano data race**
  - **`notify()` risveglia un qualsiasi processo in attesa sull'oggetto – per questo esiste `notifyAll()` !**
  - **L'esecuzione di `wait()` può venire interrotta**

Il costrutto *monitor* – 6

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

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

Un vero monitor?

Il costrutto *monitor* – 7

- Separa la sincronizzazione dalla condivisione di dati
- Non consente controllo dinamico sull'ordine degli accessi
  - Chi arriva primo entra anche se poi si deve sospendere
- Usa meccanismi programmatici (wait & signal) per la sincronizzazione condizionale
  - Non c'è alternativa: si tratta di un problema funzionale!

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
  - L'invio è asincrono (senza attesa), la ricezione bloccante
  - D non viene così a conoscere lo stato corrente di M

Scambio messaggi – 2

- Comunicazione sincrona e asincrona sono duali: dall'una si può 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

Scambio messaggi – 3

- 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 e nomi unici – *rendez-vous*
  - Processo A :: B ! Msg Processo B :: A ? Msg
  - In CSP questa forma di comunicazione è unidirezionale


Scambio messaggi – 4

- **Lo scambio dati può essere bidirezionale senza richiedere 2 messaggi (A/R)**
- **Il destinatario D offre un "luogo di entrata" (*entry*) dove ricevere parametri `in` e restituire valori su parametri `out`**
- **M nomina D e il servizio (= canale) mentre per D non vale il viceversa**
  - **Asimmetria di denominazione**

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



Scambio messaggi – 5

- **Una precondizione può essere preposta all'attesa di un [tipo di] messaggio**
  - **Per Dijkstra i comandi di ricezione [alternativi] sono selettivi, non-deterministici e dotati di "guardia"**
    - 1975, "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs", CACM, vol. 18(8), pp. 453-457

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

- **La "guardia" è una espressione Booleana**
  - **Quando vera abilita l'esecuzione del comando associato**

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