Comunicazione tra processi

Anno accademico 2018/19
Sistemi Concorrenti e Distribuiti

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

Premesse – 1

- **Molto raramente i processi di un sistema concorrente 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 per la buona interazione tra loro**
  - **Interfaccia ⇨ contratto ⇨ relazione formale**
- **Per realizzare interfacce 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**
    - Modalità 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 mancata preservazione di questa garanza 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***
  - **Questo è non-determinismo indesiderabile**
  - **Tuttavia, in un programma concorrente, vi sono forme desiderabili di non-determinismo di esecuzione**

Due problemi complementari

- **[P1] Come rendere atomiche le sezioni critiche**
  - **Le *data race* a questo livello sono chiamate *low-level***
- **[P2] Come individuare bene le sezioni critiche**
  - **Le *data race* a questo livello sono *high-level***
- **Errori in P2 espongono a due rischi**
  - ***Non-atomic protection fault*: quando un processo accede più volte la stessa risorsa condivisa con operazioni parziali**
  - ***Lost-update fault*: quando vi è 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**

P1.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: ora 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: ora va bloccata
   lock = 0;
   <sezione critica S2 su X>;
   // e nuovamente liberata dopo l'uso
   lock = 1;
}
```

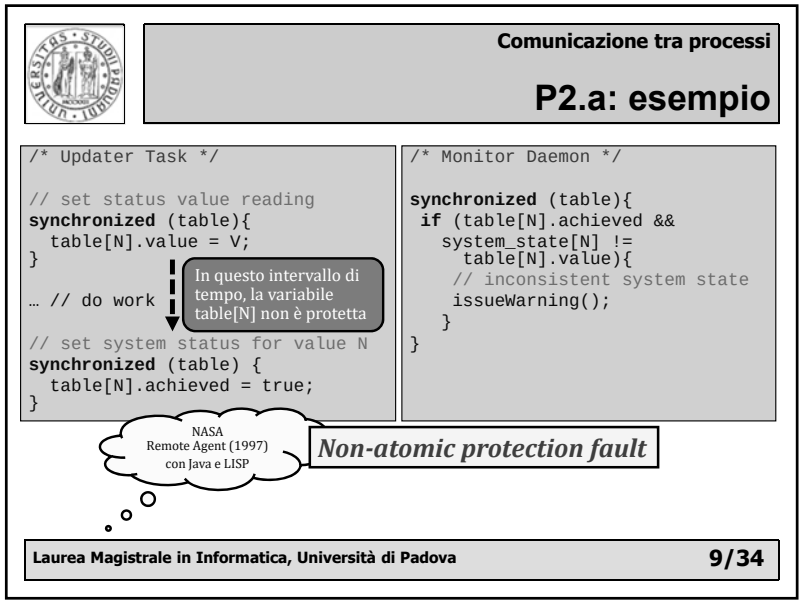
Le sezioni critiche S1 e S2 **non** sono atomiche: perché?

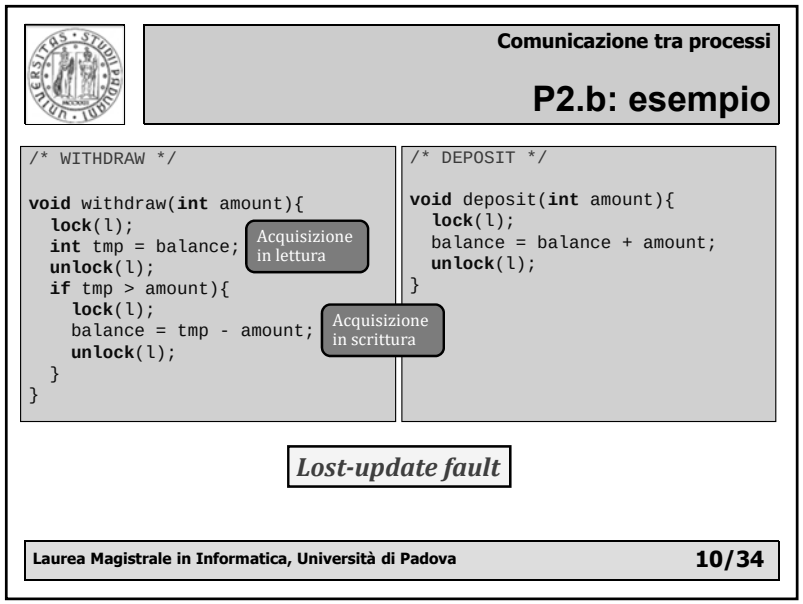
P1.b: esempio

```
/* 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***


Comunicazione tra processi
P2.a: esempio
/* Updater Task */
// set status value reading
synchronized (table){
table[N].value = V;
}
… // do work
In questo intervallo di tempo, la variabile table[N] non è protetta
// 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
Laurea Magistrale in Informatica, Università di Padova
9/34


Comunicazione tra processi
P2.b: esempio
/* WITHDRAW */
void withdraw(int amount){
lock(l);
int tmp = balance;
unlock(l);
if tmp > amount){
lock(l);
balance = tmp - amount;
unlock(l);
}
}
Acquisizione in lettura
Acquisizione in scrittura
/* DEPOSIT */
void deposit(int amount){
lock(l);
balance = balance + amount;
unlock(l);
}
Lost-update fault
Laurea Magistrale in Informatica, Università di Padova
10/34


Comunicazione tra processi
Problemi di sincronizzazione
Mutua esclusione: exclusion synchronization
A ogni instante, non più di un processo può avere possesso di una risorsa condivisa
Offrire garanzie di atomicità
Sincronizzazione condizionale: avoidance synchronization
Certe pre-condizioni logiche devono valere all'atto dell'acquisizione della risorsa
Permettere di esprimerle e di gestirne la dinamica
Problema classico: buffer finito condiviso nel modello produttore-consumatore
Laurea Magistrale in Informatica, Università di Padova
11/34


Comunicazione tra processi
Rischi di sincronizzazione – 1
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 infinito: lockout, starvation
Laurea Magistrale in Informatica, Università di Padova
12/34

Rischi di sincronizzazione – 2

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

Rischi di sincronizzazione – 2

- **Stallo (2/2)**
  - **4 possibile strategie per affrontare il problema**
    - **Indifferenza** : sperare che il problema non si manifesti
    - **Prevenzione statica** : accertare che progetto e realizzazione del sistema siano liberi da condizioni di rischio
    - **Prevenzione dinamica** : analizzare lo stato di esecuzione presente e futuro del sistema per evitare che l'ingresso in condizione di stallo
    - **Rilevazione e trattamento** : riconoscere il verificarsi del problema e utilizzare azioni speciali per ripristinare uno stato noto e sicuro

Rischi di sincronizzazione – 3

- **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 dei DB**
  - **Le transazioni aggiungono *consistency* (gli effetti sono ordinati) e *isolation* (visibile solo l'effetto finale) ad atomicity**

Rischi di sincronizzazione – 4

- **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**
    - **Attenzione**: quando il parallelismo reale è elevato e le sezioni critiche sono brevi conviene usare ***spin-locking*** ☹ perché molto meno costosi di accodamenti con *context switch*
  - **Forme di accodamento *fair***
    - Per comunicazione e sincronizzazione tra processi
    - Se l' attesa massima in coda è nota si parla di ***bounded fairness***
- **Ma la correttezza funzionale del programma non deve dipendere dalle politiche di ordinamento**

Requisiti di sincronizzazione

- **Una modalità di sincronizzazione è accettabile se soddisfa 4 condizioni**
  - Garantire accesso esclusivo
  - Garantire attesa finita
  - Non fare assunzioni sull'ambiente di esecuzione
  - 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 = ***busy wait***
    - Violazione della condizione 4
    - Rischio di *data race* sulla variabile di controllo (ma con effetti non gravi)

```
Processo 0 ::                          Processo 1 ::
while (TRUE) {                         while (TRUE) {
  while (turn != 0); /* busy wait */     while (turn != 1); /* busy wait */
  critical_region();                     critical_region();
  turn = 1;                              turn = 0;
  outside_cr();      Comando di alternanza  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 of computation
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 dà 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;
}
```

Argh!

```
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(s) e V(s) è lasciato alla disciplina del programmatore

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*", CACM 17(10):549-557 (1974)
  - **Risorsa e 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**

Soluzioni canoniche – 3

- **Sincronizzazione condizionale con *monitor***
  - **Riesce a rappresentare e a controllare condizioni logiche di accesso più sofisticate della mutua esclusione**
  - **Mediante segnali associati a variabile di condizione `Var`**

    Wait (Var) → mette il chiamante in attesa

    Signal (Var) → risveglia il processo in attesa
  - **La primitiva di risveglio non ha memoria**
    - L'effetto è nullo 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 (Var) blocca il chiamante quando lo stato della risorsa non consente di procedere**
  - **Produttore @ contenitore pieno / consumatore @ contenitore vuoto**
- **Signal (Var) risveglia il primo processo bloccato quando la condizione attesa è verificata**
  - **Il processo risvegliato contende la CPU al chiamante di Signal(Var)**
  - **La convenzione più usata è che Signal (Var) sia l'ultima azione eseguita in procedure di monitor**
- **Il compilatore garantisce mutua esclusione nell'esecuzione di Wait (Var) e Signal (Var)**
  - **Questo previene il rischio di *data race* sulle variabili di controllo**

Il costrutto *monitor* – 3

- **Critica**
  - **Il monitor non consente controllo dinamico sull'ordine degli accessi alla sezione critica**
    - Chi arriva primo entra anche se poi si deve sospendere: indesiderabile spreco
  - **Il monitor usa meccanismi programmatici (wait & signal) per la sincronizzazione condizionale**
    - La *avoidance synchronization* è problema intrinsecamente funzionale, però è indesiderabile che l'algoritmo dell'utente debba decidere su di essa

In Java (non *built-in*) ☹ – 1

```
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?

In Java (non *built-in*) ☹ – 2

- ***Exclusion synchronization* e *avoidance synchronization* sono problemi distinti**
  - **Una regola gli accessi, l'altra le condizioni algoritmiche**
  - **Servirebbero code distinte**
- **Java offre una sola coda**
  - **Questa scelta è fonte di confusione semantica**
  - **Chi viene risvegliato da `notify()`?**
  - **Quali conseguenze dall'uso di `notifyAll()`?**
- **Nota come avviene il trasferimento del *lock***

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 (*ack* 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"**
    - E.W. Dijkstra, "*Guarded Commands, Nondeterminacy, and Formal Derivation of Programs*", CACM, 18(8):453-457 (1975)

```
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**