



Comunicazione tra processi

Comunicazione tra processi



Anno accademico 2019/2020
Sistemi Concorrenti e Distribuiti

Tullio Vardanega, tullio.vardanega@unipd.it

Laurea Magistrale in Informatica, Università di Padova 1/34



Comunicazione tra processi

Premesse – 1

- La concorrenza è collaborativa
 - Difficilmente i processi di un sistema concorrente sono indipendenti
 - Altrimenti parleremmo di un sistema parallelo
- La definizione delle interfacce di comunicazione tra le entità concorrenti di un sistema è fondamentale per la loro buona interazione
 - Interfaccia → contratto → relazione formale (analizzabile)
- Per realizzare interfacce dobbiamo far riferimento a un modello di comunicazione tra entità

Laurea Magistrale in Informatica, Università di Padova 2/34



Comunicazione tra processi

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
 - Nessuna condivisione di memoria, ma implicazioni sul progresso
 - 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

Laurea Magistrale in Informatica, Università di Padova 3/34



Comunicazione tra processi

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 garanzia da luogo al rischio di *data race*
 - R. Netzer e B. Miller (ACM LoPLAS 1-1, 1992) mostrano che verificare la presenza di *data race* in un programma è un problema *NP-hard*

Laurea Magistrale in Informatica, Università di Padova 4/34



Comunicazione tra processi

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

Laurea Magistrale in Informatica, Università di Padova5/34



Comunicazione tra processi

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

Laurea Magistrale in Informatica, Università di Padova6/34



Comunicazione tra processi

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?

Laurea Magistrale in Informatica, Università di Padova7/34



Comunicazione tra processi

P1.b: esempio

```
/* DEPOSIT */
amount = read_amount();
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*

Laurea Magistrale in Informatica, Università di Padova8/34

Comunicazione tra processi

P2.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();
    }
}
```

In questo intervallo di tempo, la variabile table[N] non è protetta

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(1);
    int tmp = balance;
    unlock(1);
    if (tmp > amount){
        lock(1);
        balance = tmp - amount;
        unlock(1);
    }
}
```

```
/* DEPOSIT */
void deposit(int amount){
    lock(1);
    balance = balance + amount;
    unlock(1);
}
```

Acquisizione in lettura

Acquisizione in scrittura

Lost-update fault

Laurea Magistrale in Informatica, Università di Padova

10/34

Comunicazione tra processi

Problemi di sincronizzazione

Mutua esclusione: *exclusion synchronization*

- A ogni istante, non più di un processo può avere possesso di una risorsa condivisa
 - Offrire garanzie di atomicità

Sincronizzazione condizionale: *avoidance synchronization* Di questo parleremo più avanti

- Pre-condizioni logiche date devono valere all'atto dell'acquisizione esclusiva 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

 Comunicazione tra processi

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

Laurea Magistrale in Informatica, Università di Padova 13/34

 Comunicazione tra processi

Rischi di sincronizzazione – 2

❑ Stallo (2/2)

- 4 possibili 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 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

Laurea Magistrale in Informatica, Università di Padova 14/34

 Comunicazione tra processi

Rischi di sincronizzazione – 3

❑ Tecniche di prevenzione

- Basta impedire il verificarsi di anche una sola delle 4 pre-condizioni (quale?)
 - Tecniche, statiche o dinamiche, che impediscano l'accumulo di risorse o il formarsi di attese circolari

❑ Tecniche di trattamento

- *Transactional memory* : meccanismi di *concurrency control* analoghi a quelli dei DB
- Le transazioni aggiungono *consistency* (gli effetti sono ordinati) e *isolation* (visibile solo l'effetto finale) ad atomicity
- Molto attraenti per i sistemi paralleli dove il trattamento esplicito della sincronizzazione è molto penalizzante

Laurea Magistrale in Informatica, Università di Padova 15/34

 Comunicazione tra processi

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*

Laurea Magistrale in Informatica, Università di Padova 16/34



Comunicazione tra processi

Requisiti generali

- ❑ **A un linguaggio concorrente servono**
 - Meccanismi per evitare o ridurre l'uso di attesa attiva
 - Ma quando il parallelismo reale è elevato e le sezioni critiche sono brevi conviene usare *spin-locking* perché molto meno costosi degli 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*
- ❑ **La correttezza funzionale del programma non deve dipendere dalle politiche di ordinamento**

Laurea Magistrale in Informatica, Università di Padova

17/34



Comunicazione tra processi

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

Laurea Magistrale in Informatica, Università di Padova

18/34



Comunicazione tra processi

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 ::

```
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();
}
```

Laurea Magistrale in Informatica, Università di Padova

19/34



Comunicazione tra processi

Soluzioni "al limite" – 2

- ❑ **Algoritmo di Dekker**

```
var flag: array [0..1] of boolean;
turn: 0..1; -- i, j sono due processi
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;
```

Algoritmo concepito da T.J. Dekker e applicato alle sezioni critiche da E.W. Dijkstra:
flag[i] ← true
 il processo i chiede 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.

Laurea Magistrale in Informatica, Università di Padova

20/34



Comunicazione tra 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
```

Laurea Magistrale in Informatica, Università di Padova

21/34



Comunicazione tra processi

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

Argli! →

L'uso di $P(s)$ e $V(s)$ è lasciato alla disciplina del programmatore



Laurea Magistrale in Informatica, Università di Padova

22/34



Comunicazione tra processi

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

Laurea Magistrale in Informatica, Università di Padova

23/34



Comunicazione tra processi

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 finché la condizione `var` non diventi vera
 - `signal(var)` → risveglia il processo in attesa sulla condizione `var`
- La primitiva di risveglio non ha memoria
 - Effetto nullo se nessun processo è in attesa su quella condizione

Laurea Magistrale in Informatica, Università di Padova

24/34

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; // 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;
                    
```

Laurea Magistrale in Informatica, Università di Padova
25/34

Comunicazione tra processi

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

Laurea Magistrale in Informatica, Università di Padova
26/34

Comunicazione tra processi

Il costrutto *monitor* – 3

- ❑ **Critica**
 - **Il monitor non consente all'applicazione di decidere dinamicamente l'ordine degli accessi alla sezione critica**
 - Chi vi arriva primo entra anche se poi si deve sospendere a causa di condizione logica interna non favorevole (sincronizzazione condizionale)
 - Questo è un indesiderabile spreco
 - **Il monitor usa meccanismi programmatici (wait & signal) per realizzare sincronizzazione condizionale**
 - La *avoidance synchronization* è problema intrinsecamente funzionale, però è indesiderabile che l'algoritmo dell'utente debba decidere su di essa

Laurea Magistrale in Informatica, Università di Padova
27/34

Comunicazione tra processi

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?

Laurea Magistrale in Informatica, Università di Padova
28/34



Comunicazione tra processi

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()`?
- ❑ **Come avviene il trasferimento del *lock* ?**

Laurea Magistrale in Informatica, Università di Padova

29/34



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

Laurea Magistrale in Informatica, Università di Padova

30/34



Comunicazione tra processi

Scambio messaggi – 2



- ❑ **Comunicazione sincrona e asincrona sono duali: dall'una si può ottenere l'altra**
 - **Sincrona → Asincrona**
Introdurre un'entità intermedia V tra M e D produce comunicazione asincrona tra M e D
 - 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

Laurea Magistrale in Informatica, Università di Padova

31/34



Comunicazione tra processi

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

Laurea Magistrale in Informatica, Università di Padova

32/34





Comunicazione tra processi

Scambio messaggi – 4

- ❑ Lo scambio dati può essere bidirezionale senza richiedere 2 coppie alternate di send/receive
- ❑ Il destinatario D offre un “canale” (*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;
```

Laurea Magistrale in Informatica, Università di Padova

33/34





Comunicazione tra processi

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)
- ❑ La “guardia” è una **espressione Booleana**
 - Quando vera abilita l’esecuzione del comando associato

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

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

Laurea Magistrale in Informatica, Università di Padova

34/34