

Università degli Studi di Padova

## Comunicazione tra processi

# SCD

Anno accademico 2006/7  
Corso di Sistemi Concorrenti e Distribuiti

Tullio Vardanega, [tullio.vardanega@math.unipd.it](mailto:tullio.vardanega@math.unipd.it)

Corso di Laurea Specialistica in Informatica, Università di Padova 1/34

Università degli Studi di Padova

## Comunicazione tra processi

### Premesse – 1

- ❑ I processi di un sistema concorrente sono raramente indipendenti l'uno dall'altro
- ❑ Ne segue che la definizione delle interfacce (relazioni formali) tra le entità concorrenti di un sistema è attività di elevata criticità
- ❑ Per questo occorre far riferimento a un modello di comunicazione tra entità

Corso di Laurea Specialistica in Informatica, Università di Padova 2/34

Università degli Studi di Padova

## Comunicazione tra processi

### 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

Corso di Laurea Specialistica in Informatica, Università di Padova 3/34

Università degli Studi di Padova

## Comunicazione tra processi

### 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?

Corso di Laurea Specialistica in Informatica, Università di Padova 4/34

Università degli Studi di Padova

## Comunicazione tra processi

### Sincronizzazione – 1

- ❑ **Mutua esclusione**
  - Assicura che, a ogni istante, 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**
  - Impone condizioni (logiche, di stato interno) che devono valere all'atto della sincronizzazione
    - Problema classico: *buffer* finito condiviso nel modello produttore-consumatore

Corso di Laurea Specialistica in Informatica, Università di Padova 5/34

Università degli Studi di Padova

## Comunicazione tra processi

### 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*)

Corso di Laurea Specialistica in Informatica, Università di Padova 6/34

Università degli Studi di Padova Comunicazione tra processi

## Sincronizzazione – 3

❑ **Stallo (1/2)**

- Impedisce di proseguire a tutti i processi coinvolti
- Per aver luogo richiede il verificarsi di 4 pre-condizioni ben riconoscibili
  - **Mutua esclusione** (già definita)
  - **Cumulazione di risorse**  
I processi possono accumulare risorse e trattenerle mentre attendono di acquisirne altre
  - **Assenza di prelascio**  
Le risorse vengono rilasciate solo volontariamente
  - **Attesa circolare**  
Un processo attende almeno una risorsa in possesso del successivo processo in catena

Corso di Laurea Specialistica in Informatica, Università di Padova 7/34

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 possa verificarsi
  - **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 che esso possa entrare in una condizione di stallo
    - Può bastare impedire strutturalmente il verificarsi di almeno una delle 4 pre-condizioni?
  - **Rilevazione e trattamento**  
Comporta capacità di riconoscimento del verificarsi del problema e richiede meccanismi complessi per ripristinare uno stato noto e sicuro

Corso di Laurea Specialistica in Informatica, Università di Padova 8/34

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 invece in presenza di qualunque altra politica
  - A priorità
  - LIFO
  - A 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*

Corso di Laurea Specialistica in Informatica, Università di Padova 9/34

Università degli Studi di Padova Comunicazione tra processi

## Requisiti generali

❑ **Per realizzare un linguaggio concorrente abbiamo bisogno di**

- Un meccanismo pratico per evitare l'uso di attesa attiva
- Forme di accodamento di processi che diano garanzie di *fairness*
  - Da usare per il supporto di comunicazione e sincronizzazione

❑ **La correttezza funzionale del programma non deve però dipendere da queste forme e meccanismi!**

Corso di Laurea Specialistica in Informatica, Università di Padova 10/34

Università degli Studi di Padova Comunicazione tra processi

## Requisiti di sincronizzazione

❑ **Una soluzione al problema della sincronizzazione di processi è ammissibile se soddisfa le seguenti 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

Corso di Laurea Specialistica in Informatica, Università di Padova 11/34

Università degli Studi di Padova Comunicazione tra processi

## Soluzioni "al limite" – 1

❑ **Mutua esclusione con variabili condivise mediante alternanza stretta tra coppie di processi, ma con 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();
}
```

**Processo 1 ::**

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

← Comando di alternanza

Corso di Laurea Specialistica in Informatica, Università di Padova 12/34

Università degli Studi di Padova Comunicazione tra processi

## 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[j] := 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.

Corso di Laurea Specialistica in Informatica, Università di Padova 13/34

Università degli Studi di Padova Comunicazione tra 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 == j)
    /* busy wait */;
}
OUT(int i) :: {
  flag[i] = FALSE;
}
    
```

```

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

Uscita dalla sezione critica

Corso di Laurea Specialistica in Informatica, Università di Padova 14/34

Università degli Studi di Padova Comunicazione tra processi

## Esempi classici

□ **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, 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**

- [Versione Ada](#)
- [Versione Java](#)

Corso di Laurea Specialistica in Informatica, Università di Padova 15/34

Università degli Studi di Padova Comunicazione tra processi

## I filosofi a cena – 1

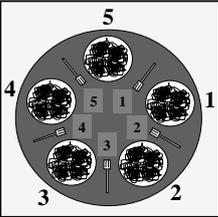
□ **Situazione artificiosa, utile per illustrare importanti aspetti progettuali di concorrenza**

- **Condivisione risorse** → le posate
- **Mutua esclusione** → 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** → vedi riquadro seguente

Corso di Laurea Specialistica in Informatica, Università di Padova 16/34

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

Corso di Laurea Specialistica in Informatica, Università di Padova 17/34

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 che più filosofi possano cenare simultaneamente

Corso di Laurea Specialistica in Informatica, Università di Padova 18/34

Università degli Studi di Padova Comunicazione tra processi

## 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

Corso di Laurea Specialistica in Informatica, Università di Padova 19/34

Università degli Studi di Padova Comunicazione tra processi

## Esercizio – 2

□ **Dato il modello di codice in diapositiva 18 indicare**

- Pregi e difetti della soluzione illustrata
- 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

Corso di Laurea Specialistica in Informatica, Università di Padova 20/34

Università degli Studi di Padova Comunicazione tra processi

## Possibili soluzioni – 1

□ **Mutua esclusione mediante semafori**

- La realizzazione deve garantire che le operazioni sul semaforo (storicamente dette **P** e **V**) siano indivisibili
- All'attesa sul semaforo deve corrispondere una forma di accodamento del processo fino alla condizione di rilascio
- Il semaforo ha un valore di inizializzazione e 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 ( $1 \rightarrow$  **mutua esclusione** |  $> 1 \rightarrow$  più accessi simultanei)
- Questa tecnica dipende troppo dalla disciplina del programmatore

Corso di Laurea Specialistica in Informatica, Università di Padova 21/34

Università degli Studi di Padova Comunicazione tra processi

## Possibili soluzioni – 2

□ **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 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

Corso di Laurea Specialistica in Informatica, Università di Padova 22/34

Università degli Studi di Padova Comunicazione tra processi

## Possibili soluzioni – 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`)  $\rightarrow$  forza l'attesa del chiamante
- Signal (`var_condizione`)  $\rightarrow$  risveglia il processo in attesa
  - Il segnale di risveglio (Signal) **non ha memoria**, dunque va perso se nessuno lo recepisce

Corso di Laurea Specialistica in Informatica, Università di Padova 23/34

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;
    
```

Corso di Laurea Specialistica in Informatica, Università di Padova 24/34

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*

Corso di Laurea Specialistica in Informatica, Università di Padova 25/34

Università degli Studi di Padova

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, non su una condizione
- ❑ **I metodi *wait()* e *notify()* invocati esplicitamente all'interno di metodi *synchronized* realizzano le condizioni evitando il verificarsi di *race condition***
  - *notify()* risveglia uno qualsiasi dei processi in attesa sull'oggetto
  - L'esecuzione di *wait()* può venire interrotta, il che va trattato esplicitamente come eccezione

Corso di Laurea Specialistica in Informatica, Università di Padova 26/34

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

Corso di Laurea Specialistica in Informatica, Università di Padova 27/34

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 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

Corso di Laurea Specialistica in Informatica, Università di Padova 28/34

Università degli Studi di Padova

Comunicazione tra processi

## Il costrutto *monitor* – 5

- ❑ **Il *monitor* separa sincronizzazione da scambio (condivisione) di dati**
- ❑ **Il *monitor* non è però capace di controllare dinamicamente l'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**

Corso di Laurea Specialistica in Informatica, Università di Padova 29/34

Università degli Studi di Padova

Comunicazione tra processi

## Scambio messaggi – 1

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

Corso di Laurea Specialistica in Informatica, Università di Padova 30/34

Università degli Studi di Padova Comunicazione tra processi

## 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

Corso di Laurea Specialistica in Informatica, Università di Padova 31/34

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 a essa 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**

Corso di Laurea Specialistica in Informatica, Università di Padova 32/34

Università degli Studi di Padova Comunicazione tra processi

## Scambio messaggi – 4

- **Lo scambio dati può utilmente essere bidirezionale senza richiedere 2 messaggi**
- **Il primo D 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 M nomina D e il servizio (= canale) senza che valga il viceversa**
  - **Asimmetria di denominazione**

Corso di Laurea Specialistica in Informatica, Università di Padova 33/34

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 M e M rende la loro 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 impone sincronizzazione**

Corso di Laurea Specialistica in Informatica, Università di Padova 34/34