Introduzione alla Programmazione Concorrente

Algoritmo, programma, processo

- **Algoritmo**: Procedimento logico che deve essere eseguito per risolvere un determinato problema.
- **Programma:** Descrizione di un algoritmo mediante un opportuno formalismo (linguaggio di programmazione), che rende possibile l'esecuzione dell'algoritmo da parte di un particolare elaboratore.
- **Processo:** insieme ordinato degli eventi cui dà luogo un elaboratore quando opera sotto il controllo di un programma.

Elaboratore: entità ***astratta*** realizzata in hardware e parzialmente in software, in grado di eseguire programmi (descritti in un dato linguaggio).

Evento: Esecuzione di un'operazione tra quelle appartenenti all'insieme che l'elaboratore sa riconoscere ed eseguire; ogni evento determina una transizione di stato dell'elaboratore

➔ Un programma descrive non un processo, ma un insieme di processi, ognuno dei quali è relativo all'esecuzione del programma da parte dell'elaboratore per un determinato insieme di dati in ingresso.

Processo sequenziale

Sequenza di stati attraverso i quali passa l'elaboratore durante l'esecuzione di un programma (storia di un processo o traccia dell'esecuzione del programma).

Esempio: valutare il massimo comune divisore tra due numeri naturali x e y:

a) Controllare se i due numeri x e y sono uguali, nel qual caso il loro M.C.D. coincide con il loro valore
b) Se sono diversi, valutare la loro differenza
c) Tornare ad a) prendendo in considerazione il più piccolo dei due e la loro differenza

3

Esempio: M.C.D. di x e y (numeri naturali)

```
int MCD(int x, int y)
{ a = x; b = y;
  while (a != b)
      if (a > b) a = a – b
                 else b = b-a;
  return a;
}
```

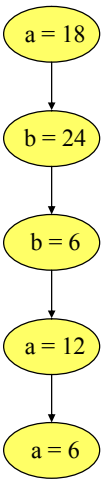
Consideriamo la chiamata: `mcd(18,24)`

x	18	18	18	18	18	18
y	24	24	24	24	24	24
a	-	18	18	18	12	6
b	-	-	24	6	6	6
	stato iniziale					stato finale

4

Grafo di precedenza

- Un processo può essere rappresentato tramite un grafo orientato detto ***grafo di precedenza*** del processo
- I ***nodi*** del grafo rappresentano i singoli eventi del processo, mentre gli ***archi*** identificano le precedenze temporali tra tali eventi
- Un ***evento*** corrisponde all'esecuzione di un'operazione tra quelle appartenenti all'insieme che l'elaboratore sa riconoscere ed eseguire
- Essendo il processo strettamente sequenziale, il grafo di precedenza è ad ***Ordinamento Totale*** (ogni nodo ha esattamente un predecessore ed un successore)



5

Processi non sequenziali

- L'ordinamento totale di un grafo di precedenza:
  - puo` derivare dalla natura *sequenziale* del problema da risolvere .
  - puo` derivare dalla natura sequenziale dell'elaboratore.
- Esistono molti esempi di applicazioni che potrebbero più naturalmente essere rappresentate da ***processi non sequenziali***.

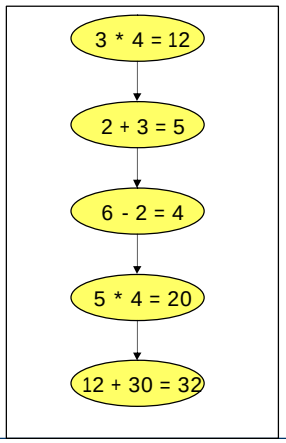
➔***Processo non sequenziale:*** l'insieme degli eventi che lo descrive e` ordinato secondo una ***relazione d'ordine parziale.***

6

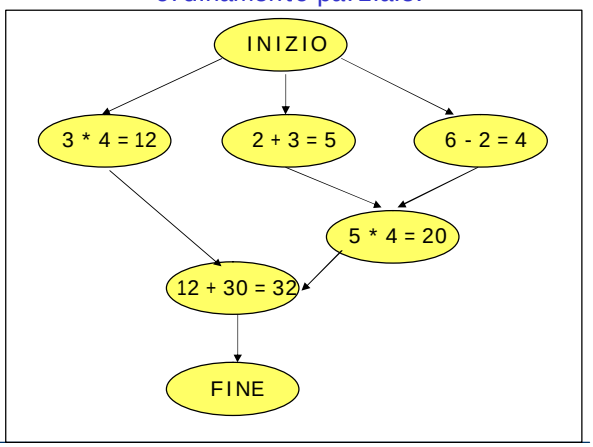
Esempio: valutazione dell'espressione

(3 * 4) + (2 + 3) * (6 - 2)

Grafo di precedenza ad ordinamento totale:



Grafo di precedenza ad ordinamento parziale:



7

Esempio – segue:

- La logica del problema ***non impone un ordinamento totale*** fra le operazioni da eseguire; ad esempio è indifferente che venga eseguito (2 + 3) prima di eseguire (6 - 2) o viceversa.

➔ Certi eventi del processo sono tra loro scorrelati da qualunque relazione di precedenza temporale ➔ il risultato dell'elaborazione è indipendente dall'ordine con cui gli eventi avvengono.

- Molti settori applicativi possono essere rappresentati da processi non sequenziali: sistemi in tempo reale, sistemi operativi, sistemi di simulazione, etc…

8

L'esecuzione di un processo non sequenziale richiede:

- **elaboratore non sequenziale**
- **linguaggio di programmazione non sequenziale**

- ***Elaboratore non sequenziale*** (in grado di eseguire più operazioni contemporaneamente):
  - architettura parallela
  - sistemi multielaboratori (a)
  - sistemi monoelaboratori (b)

Linguaggi non sequenziali (o concorrenti). Caratteristica comune: consentire la descrizione di un insieme di attività concorrenti tramite moduli sequenziali che possono essere eseguiti in parallelo (processi sequenziali)

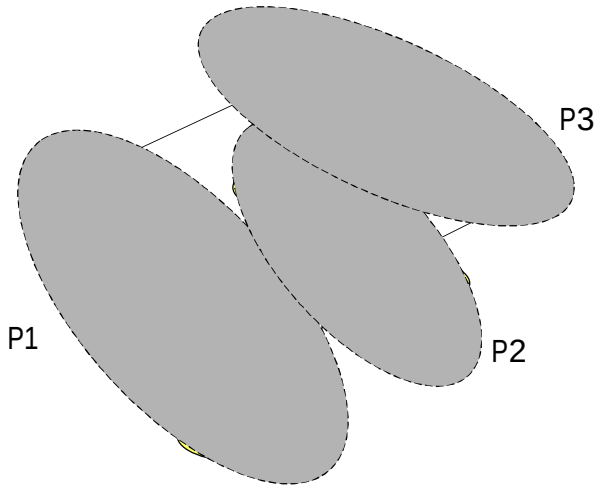
9

Scomposizione di un processo non sequenziale

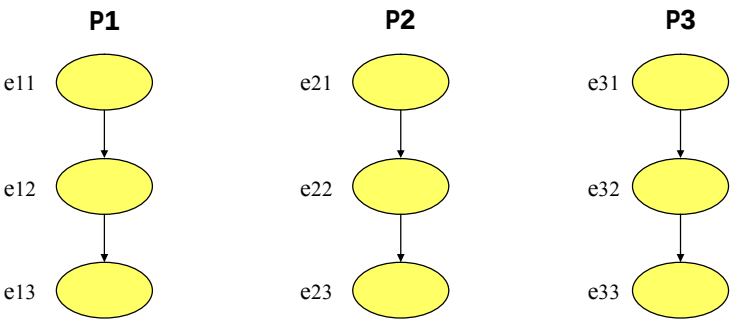
- Scomposizione di un processo non sequenziale in un insieme di processi sequenziali, eseguiti "contemporaneamente", ma analizzati e programmati separatamente.
- Consente di dominare la complessità di un algoritmo non sequenziale
- Le attività rappresentate dai processi possono essere:
  - completamente indipendenti
  - interagenti

10

Scomposizione di un processo non sequenziale

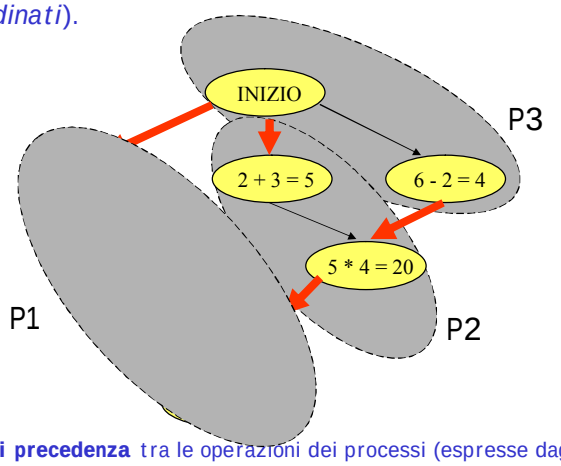


Processi indipendenti



L'evoluzione di un processo non influenza quella degli altri.

- Nel caso di grafi connessi ad ordinamento parziale, la scomposizione del processo globale in processi sequenziali consiste nell'individuare sul grafo un insieme P1….Pn di sequenze di nodi (*insiemi di nodi totalmente ordinati*).



- Presenza di **vincoli di precedenza** tra le operazioni dei processi (espresse dagli **archi**) o **vincoli di sincronizzazione**
- In questo caso i tre processi sono **interagenti**

13

Processi interagenti

- I processi, in corrispondenza di ogni arco, **devono sincronizzarsi**, cioè ordinare i loro eventi come specificato dal grafo di precedenza.

- **Vincolo di sincronizzazione:** vincolo imposto da ogni arco del grafo di precedenza che collega nodi di processi diversi.

14

Interazione tra processi

COOPERAZIONE: interazione ***prevedibile e desiderata***, insita cioè nella logica dei programmi (sincronizzazione diretta o esplicita)

COMPETIZIONE: interazione prevedibile e non desiderata, ma necessaria, per l'uso di risorse comuni che non possono essere usate contemporaneamente.

(sincronizzazione indiretta o implicita)

INTERFERENZA: interazione non prevedibile e non desiderata, provocata da errori di programmazione:

1. Inserimento nel programma di interazioni tra processi non richieste dalla natura del problema
2. Erronea soluzione a problemi di interazione (cooperazione e competizione) necessarie per il corretto funzionamento del programma

ARCHITETTURE E LINGUAGGI PER LA PROGRAMMAZIONE CONCORRENTE

Linguaggi per la programmazione concorrente

- Un linguaggio per la programmazione concorrente consente di rappresentare algoritmi non sequenziali.

- L'esecuzione di programmi scritti in un linguaggio concorrente su una macchina concorrente (es. sistema operativo multiprogrammato), produce un insieme di processi sequenziali interagenti.

Esempio: C+ system call linux, java

17

Proprietà di un linguaggio per la programmazione concorrente

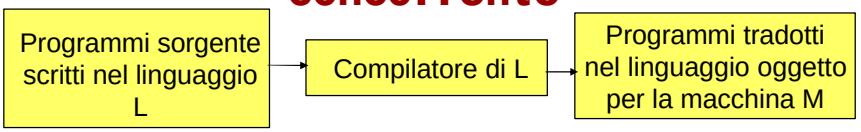
Contenere appositi costrutti con i quali sia possibile dichiarare moduli di programma destinati ad essere eseguiti come *processi sequenziali distinti*:

- Creazione e terminazione: Non tutti i processi costituenti un'elaborazione vengono eseguiti *contemporaneamente.* Alcuni processi vengono svolti se, ***dinamicamente***, si verificano particolari condizioni. E' quindi necessario specificare quando un processo deve essere *attivato e terminato.*

- Interazione: Occorre che siano presenti strumenti linguistici per specificare le *interazioni* che dinamicamente potranno aversi tra i vari processi

18

Architettura di una macchina concorrente

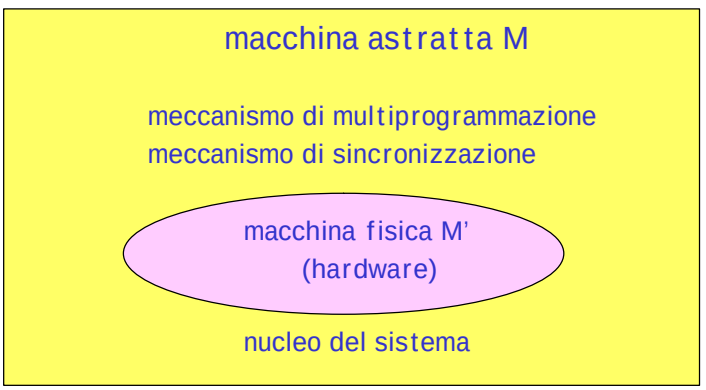


- Difficilmente M ha una struttura fisica con tante unità di elaborazione quanti sono i processi da svolgere contemporaneamente durante l'esecuzione di un programma concorrente.

➔ M è una ***macchina astratta*** ottenuta con tecniche software (o hardware) basandosi su una macchina fisica M' più semplice (con un numero di unità di elaborazione molto minore del numero dei processi).

ESEMPIO: un sistema operativo multiprogrammato e` una macchina astratta concorrente

19



Oltre ai meccanismi di multiprogrammazione e sincronizzazione è presente anche un *meccanismo di protezione* (controllo degli accessi alle risorse).

- Importante per rilevare eventuali interferenze tra i processi.
- Può essere realizzato in hardware o in software nel *supporto a tempo di esecuzione*.

20

Nucleo

- Il nucleo della macchina astratta realizza il **supporto a tempo di esecuzione del linguaggio concorrente**.

- Nel nucleo sono sempre presenti due funzionalità base:
  - meccanismo di ***multiprogrammazione:*** è quello preposto alla gestione delle unità di elaborazione della macchina M' (unità reali) consentendo ai vari processi eseguiti sulla macchina astratta **M di condividere** l'uso delle unità reali di elaborazione (*scheduling*)
  - meccanismo di ***sincronizzazione*** e ***comunicazione:*** estende le potenzialità delle unità reali di elaborazione, rendendo disponibile alle unità virtuali strumenti mediante i quali due o più processi possono sincronizzarsi e comunicare.

Architettura di M

Due diverse organizzazioni logiche:

- Gli elaboratori di M sono collegati ad un'unica memoria principale (sistemi ***multiprocessore***)
- Gli elaboratori di M sono collegati da una sottorete di comunicazione, senza memoria comune (sistemi ***distribuiti***).

Le due precedenti organizzazioni logiche di M definiscono ***due modelli di interazione tra i processi:***

- *Modello a scambio di messaggi*, in cui la comunicazione e la sincronizzazione tra processi si basa sullo scambio di messaggi sulla rete che collega i vari elaboratori (*modello ad ambiente locale*).
- ***Modello a memoria comune****,* in cui l'interazione tra i processi avviene su oggetti contenuti nella memoria comune (*modello ad ambiente globale*).

Il Modello a scambio di messaggi

Interazione nel modello a scambio di messaggi

Se la macchina concorrente e` organizzata secondo il modello a scambio di messaggi:

- PROCESSO=PROCESSO PESANTE
- non vi è memoria condivisa tra processi
- i processi possono interagire mediante scambio di messaggi: *comunicazione e sincronizzazione*

➢ Il nucleo offre meccanismi a supporto:

- della comunicazione tra processi (*Inter Process Communication*, o IPC).
- della sincronizzazione (ad esempio, segnali)

➢ Il linguaggio concorrente offre costrutti per esplicitare l'interazione. Per la comunicazione:

- *send*: spedizione di messaggi da un processo ad altri
- *receive*: ricezione di messaggi

Modalita` di comunicazione

Classificazione in base a due caratteristiche:

1. designazione dei processi sorgente e destinatario di ogni comunicazione
2. tipo di sincronizzazione tra processi comunicanti

➔ Caratteristiche **ortogonali**: le soluzioni proposte per 1) e 2) sono tra loro indipendenti.

25

PRIMITIVE DI SINCRONIZZAZIONE BASATE SULLO SCAMBIO DI MESSAGGI

In generale:

SEND <expression-list> to <destination-designator>;

RECEIVE <variable-list> to <source-designator>;

- l'esecuzione della **send** valuta "***expression list***" che diventa il messaggio; "***destination-designator***" dà al programmatore il controllo su dove inviare il messaggio
- L'esecuzione della **receive** riceve il messaggio nella lista delle variabili "***variable-list***"; "***source-designator***" dà al programmatore il controllo sull'origine dei messaggi.

26

Designazione dei processi Sorgente e Destinatario di ogni comunicazione

Canale: collegamento logico tramite il quale due processi comunicano.

- E' compito del **nucleo** del linguaggio fornire l'astrazione *canale* come meccanismo primitivo per lo scambio di informazioni.
- Il linguaggio deve fornire gli **strumenti linguistici** per specificare sia la sorgente che la destinazione di ogni messaggio.

Designazione dei processi Sorgente e Destinatario

Comunicazione simmetrica:

- Vengono direttamente usati i nomi dei processi per denotare un canale, sia dal mittente che dal destinatario.

Comunicazione asimmetrica:

- Il processo mittente nomina esplicitamente il destinatario
- il processo destinatario non esprime il nome del processo con il quale vuole comunicare.

CANALE DI COMUNICAZIONE

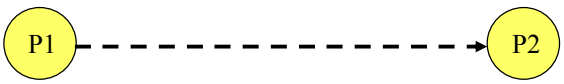
La coppia

"***destination-designator/source-designator***"

definisce un canale di comunicazione.

Schema simmetrico: i processi si nominano esplicitamente:

send <message> to P2 ***receive <message> from P1***



➔***comunicazione uno-a-uno***

29

SCHEMA ASIMMETRICO

- Il mittente nomina esplicitamente il destinatario ma questi, al contrario, non esprime il nome del processo con cui desidera comunicare
- Modello **CLIENTE-SERVITORE** ⇨ corrisponde all'uso di un processo come gestore di una risorsa

Pi (cliente)	Pr (servitore)
.	receive<richiesta>;
.	<esecuzione>;
send<richiesta> to Pr;	
	send<risultato>to Pi;
receive <risultato>;	.
.	.
.	.

30

SCHEMA ASIMMETRICO

- **Schema *molti a uno*:** i processi clienti specificano il **destinatario** delle loro richieste. Il processo servitore è pronto a ricevere messaggi da **qualunque cliente.**
- **Schema *uno a molti* o *molti a molti*:** processi cliente che inviano richieste non ad un particolare servitore, ma ad **uno qualunque** scelto tra un insieme di **servitori equivalenti**
- Problemi di **natura realizzativa: i**l supporto a tempo di esecuzione del linguaggio deve garantire che un messaggio di richiesta sia **inviato a tutti i processi servitori** ed assicurare che, non appena il messaggio è ricevuto da uno di essi, lo stesso **non sia più disponibile per tutti gli altri servitori**

31

Tipi di sincronizzazione dei processi comunicanti

Tre alternative per la semantica della *send*:

1. ***send* asincrona**
2. ***send* sincrona**
3. ***send*** tipo ***chiamata di procedura remota*** *(RPC)*

Due alternative per la *receive*:

1. ***receive* bloccante**
2. ***receive* non bloccante**

32

send asincrona

- Il processo mittente **continua la sua esecuzione** immediatamente dopo che il messaggio è stato inviato.
- Il messaggio ricevuto contiene informazioni che **non** possono essere **associate allo stato attuale del mittente** (difficoltà nella verifica dei programmi).
- La send **non è un punto di sincronizzazione**.

➔Puo` essere necessario prevedere, per realizzare un particolare schema di interazione, lo scambio esplicito di messaggi di controllo (che non contengono informazioni da elaborare).

33

send asincrona

Primitiva di basso livello:

- Facilità di realizzazione.
- Carenza espressiva (difficoltà di uso del meccanismo e di verifica dei programmi).
- **Necessita` di bufferizzazione:** il nucleo associa al destinatario un buffer di *capacità illimitata per l'accodamento dei messaggi*. Se si vuole mantenere immutata la semantica della send occorre sospendere il processo mittente in caso di coda piena.

34

Send sincrona ("rendez-vous" semplice)

- Il processo mittente si blocca in attesa che il messaggio sia stato ricevuto.

➔sincronizzazione stretta tra mittente e destinatario

- Ogni messaggio ricevuto contiene informazioni corrispondenti **allo stato attuale** del processo mittente.

➔Ciò semplifica la scrittura e la verifica dei programmi.

35

Send tipo *chiamata a procedura remota* ("rendez-vous" esteso)

- Il processo **mittente** rimane **in attesa** fino a che il ricevente non ha terminato di **svolgere l'azione richiesta.**
- Analogia semantica con la chiamata di procedura.
- Modello cliente-servitore.
- Riduzione di parallelismo.
- Punto di sincronizzazione: semplificazione della verifica dei programmi.

36

receive bloccante

Sospende il processo se non ci sono messaggi sul canale; costituisce un **punto di sincronizzazione.**

Problema:

un processo desidera ricevere **solo alcuni** messaggi ritardando l'eleborazione di altri.

Soluzione:

specificare **più canali** di ingresso per ogni processo, ciascuno dedicato a messaggi di tipo **diverso.**

Deve essere possibile inoltre specificare su quali canali attendere, in base a condizioni arbitrarie e/o allo stato del canale.

In caso di presenza contemporanea di messaggi su questi canali la scelta può essere fatta secondo criteri di priorità e in modo non deterministico.

37

receive non bloccante

- Si può ricorrere a una **primitiva (receive non bloccante)** che **verifica** lo stato di un canale, restituisce un messaggio se presente o un'indicazione di canale **vuoto.**
- Ciò consente ad un processo di selezionare l'insieme di canali da cui prelevare un messaggio.
- Inconveniente: qualora sia necessario attendere un messaggio da uno specifico canale occorre far uso di cicli di attesa attiva.

➔ Una delle soluzioni più adottate per garantire le stesse funzionalità di una primitiva non bloccante, evitando le attese attive e consentendo la verifica dello stato dei canali è quella che fa uso dei

comandi con guardia

38

Comandi con Guardia

- L'uso della *receive* bloccante può risultare complesso qualora un processo desideri ricevere solo alcuni messaggi ritardando l'elaborazione degli altri (processi **gestori di risorse**: ricezione di messaggi compatibili con lo stato delle risorse).

Comandi con guardia: sono istruzioni del tipo

<guardia> -> <istruzione>

dove <guardia> è *un'espressione logica* (guardia logica) seguita da un'eventuale *primitiva receive* (guardia di ingresso).

- La guardia viene valutata quando il comando deve essere eseguito. Si ha:
  - **guardia fallita** se l'espressione logica è **falsa**
  - **guardia valida** se l'espressione logica è **vera** e se *receive* può essere eseguita senza ritardi
  - **guardia ritardata** se l'espressione logica è **vera** ma *receive* **non può essere eseguita**

Comando con guardia *alternativo*:

```
if
        [] G1  ->       istruzione1;
        [] G2  ->       istruzione2;
        .
        .
        [] Gn  ->       istruzione n;
end;
```

tra tutte le Gi valide ne viene scelta una in modo **non deterministico**, viene eseguita **receive** e quindi l'istruzione associata.

–Se tutte le guardie **falliscono** il comando viene ***abortito***

–Se tutte le guardie non fallite sono **ritardate**, il processo **si blocca in attesa** che almeno una diventi valida.

Comando con guardia *ripetitivo*:

```
do
 [] G1  ->      istruzione1;
   .
   .
[] Gn  ->      istruzione n;

end
```

Il comando viene ripetuto finché tutte le guardie falliscono. In questo caso il comando termina la sua esecuzione.

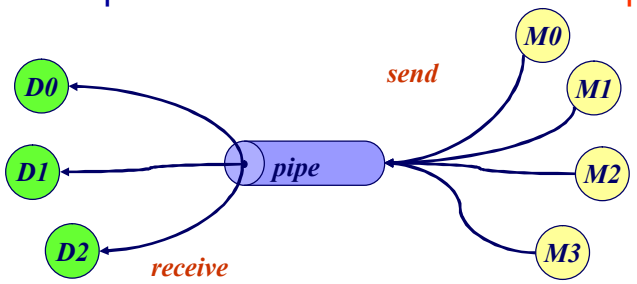
41

- L'uso dei comandi con guardia consente ad un processo di rimanere in attesa **sull'insieme di canali** per i quali è verificata la corrispondente guardia logica e di **prelevare**, non appena pronto, **il primo dei messaggi in arrivo**.
- Ciascuna guardia logica contraddistingue uno **stato della risorsa compatibile** con la ricezione di questi messaggi.
- Linguaggi concorrenti che prevedono i comandi con guardia: Ad esempio: ADA, CSP, Occam

42

Comunicazione tra processi nel sistema operativo Unix: pipe

Processi Unix possono comunicare mediante pipe.



Una pipe rappresenta un canale di comunicazione con le seguenti caratteristiche:

- schema **asimmetrico molti-a-molti**
- **send asincrona** (con sospensione del mittente in caso di buffer pieno)
- **receive bloccante**

43

Sincronizzazione tra processi nel modello a scambio di messaggi

La sincronizzazione permette di imporre vincoli sull'ordine di esecuzione delle operazioni dei processi interagenti.

Ad Esempio:

Nella cooperazione:

- Per imporre un particolare ordine cronologico alle azioni eseguite dai processi interagenti.
- per garantire che le operazioni di comunicazione avvengano secondo un ordine prefissato.

Nella competizione:

- per garantire il rispetto di politiche di accesso (ad esempio, mutua esclusione) dei processi nell'accesso alla risorsa condivisa.

44

Sincronizzazione tra processi nel modello a scambio di messaggi

In questo ambiente non vi e` la possibilita` di condividere memoria:

- Gli accessi alle risorse vengono controllati e coordinati dal sistema operativo.
- La sincronizzazione avviene mediante meccanismi offerti dal sistema operativo che consentono la notifica di "eventi" asincroni (privi di contenuto informativo) tra un processo ed altri.
- Per esempio, Linux/Unix:
  - *segnali*