

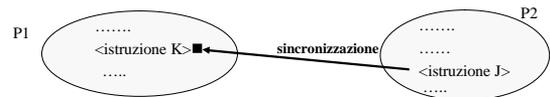
Interazione tra Processi

Processi Interagenti

- L'interazione può avvenire mediante due meccanismi:
 - **Comunicazione:** scambio di informazioni tra i processi interagenti.

– **Sincronizzazione:** imposizione di vincoli temporali sull'esecuzione dei processi.

Ad esempio, l'istruzione K del processo P1 può essere eseguita soltanto dopo l'istruzione J del processo P2



Processi interagenti

Classificazione:

- **processi interagenti/indipendenti:**
 - due processi sono indipendenti se l'esecuzione di ognuno non è in alcun modo influenzata dall'altro.
- **processi interagenti:**
 - **cooperanti:** i processi interagiscono volontariamente per raggiungere obiettivi comuni (fanno parte della stessa applicazione)
 - **in competizione:** i processi, in generale, non fanno della stessa applicazione, ma interagiscono indirettamente, per l'acquisizione di risorse comuni.

Processi Interagenti

- **Realizzazione dell'interazione:** dipende dal modello di processo:
 - **modello ad ambiente locale:** non c'è condivisione di variabili (processo pesante):
 - la comunicazione avviene attraverso scambio di messaggi
 - la sincronizzazione avviene attraverso scambio di eventi (*segnali*)
 - **modello ad ambiente globale:** più processi possono condividere lo stesso spazio di indirizzamento => possibilità di condividere variabili (es. threads):
 - variabili condivise e strumenti di sincronizzazione (es. *semaphori*)

Processi interagenti mediante scambio di messaggi

Facciamo riferimento al modello ad ambiente locale:

- non vi è memoria condivisa
- i processi possono interagire (*cooperano/competono*) mediante scambio di messaggi: *comunicazione*

➤ Il Sistema Operativo offre meccanismi a supporto della comunicazione tra processi (*Inter Process Communication, o IPC*).

Operazioni Necessarie:

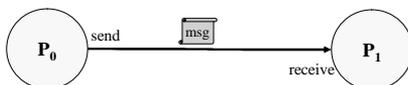
- *send*: spedizione di messaggi da un processo ad altri
- *receive*: ricezione di messaggi

Meccanismi di comunicazione tra processi

Aspetti caratterizzanti:

- caratteristiche del canale
- caratteristiche del messaggio:
 - ✓ dimensione
 - ✓ tipo
- tipo della comunicazione:
 - ✓ diretta o indiretta
 - ✓ simmetrica o asimmetrica
 - ✓ bufferizzata o no
 - ✓ ...

Scambio di messaggi



Lo scambio di messaggi avviene mediante un canale di comunicazione tra i due processi

Caratteristiche del canale:

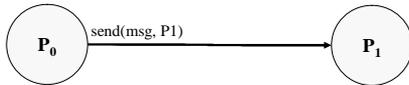
- monodirezionale, bidirezionale
- uno-a-uno, uno-a-molti, molti-a-uno, multi-a-molti
- capacità
- modalità di creazione: automatica, non automatica

Naming

In che modo viene specificata la destinazione di un messaggio?

- **Comunicazione diretta:** al messaggio viene associato l'identificatore del processo destinatario (naming esplicito)
`send(Proc, msg)`
- **Comunicazione indiretta:** il messaggio viene indirizzato a una mailbox (contenitore di messaggi) dalla quale il destinatario preleverà il messaggio:
`send(Mailbox, msg)`

Comunicazione diretta



- Il canale è creato automaticamente tra i due processi che devono *conoscersi* reciprocamente:
 - canale punto-a-punto
 - canale bidirezionale:
 - p0: send(query, P1);
 - p1: send(answ, P0)
 - per ogni coppia di processi esiste un solo canale(<P0, P1>)

Comunicazione asimmetrica

Processo produttore P: main() { msg M; do { produco(&M); ... send(C, M); }while(!fine); }	Processo consumatore C: main() { msg M; pid id; do { receive(&id, &M); ... consumo(M); }while(!fine); }
--	--

Comunicazione asimmetrica:

➤ il destinatario non è obbligato a conoscere l'identificatore del mittente: la variabile `id` raccoglie l'identificatore del mittente.

Esempio: Produttore & Consumatore

Processo produttore P: pid C =....; main() { msg M; do { produco(&M); ... send(C, M); }while(!fine); }	Processo consumatore C: pid P=....; main() { msg M; do { receive(P, &M); ... consumo(M); }while(!fine); }
--	---

Comunicazione simmetrica:

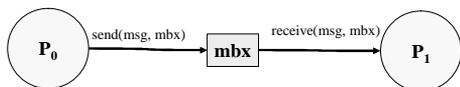
➤ il destinatario fa il *namng* esplicito del mittente

Comunicazione diretta

Problema:

- ❑ scarsa modularità:
 - la modifica del nome di un processo implica la revisione di tutte le operazioni di comunicazione
 - difficoltà di riutilizzo
 - utilità di servizi di directory (name server)

Comunicazione indiretta



- I processi cooperanti non sono tenuti a conoscersi reciprocamente e si scambiano messaggi depositandoli/prelevandoli da una mailbox *condivisa*.
- La **mailbox** (o *porta*) è una risorsa astratta condivisibile da più processi che funge da contenitore dei messaggi.

Buffering del canale

- Ogni canale di comunicazione è caratterizzato da una **capacità**: numero dei messaggi che è in grado di gestire contemporaneamente.
- Gestione secondo politica **FIFO**:
 - i messaggi vengono posti in una coda in attesa di essere ricevuti
 - la lunghezza massima della coda rappresenta la capacità del canale.



Comunicazione indiretta

Proprietà:

- il canale di comunicazione è rappresentato dalla mailbox (non viene creato automaticamente)
- il canale può essere associato a più di 2 processi:
 - ✓ mailbox di sistema: multi-a-molti (come individuare il processo destinatario di un messaggio?)
 - ✓ mailbox del processo destinatario: multi-a-uno
- canale bidirezionale:
 - p1: send(answ, mbx)
- per ogni coppia di processi possono esistere più canali (uno per ogni mailbox condivisa)

Buffering del canale

- **Capacità nulla**: non vi è accodamento perchè il canale non è in grado di gestire messaggi in attesa
 - processo mittente e destinatario devono **sincronizzarsi** all'atto di spedire (send) / ricevere (receive) il messaggio: comunicazione **sincrona** o **rendez vous**
 - **send e receive** possono essere **sospensive**:



Buffering del canale

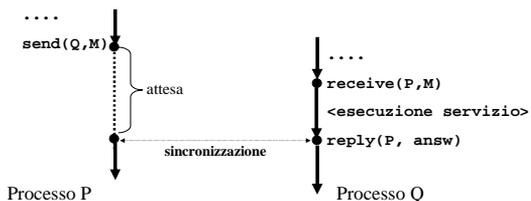
- **Capacità limitata:** esiste un limite N alla dimensione della coda:
 - se la coda **non è piena**, un nuovo messaggio viene posto in fondo
 - se la coda **è piena**: la send è sospensiva
 - se la coda **è vuota**: la receive può essere sospensiva
- **Capacità illimitata:** lunghezza della coda teoricamente infinita: non c'è possibilità di sospensione.

Caratteristiche della comunicazione tra processi Unix

- Due meccanismi di comunicazione:
 - **pipe**: comunicazione locale (nell'ambito della stessa gerarchia di processi)
 - **socket**: comunicazione in ambiente distribuito (tra processi in esecuzione su nodi diversi di una rete)
- **Pipe**: comunicazione
 - ✓ **indiretta** (senza naming esplicito)
 - ✓ canale **unidirezionale** multi-a-molti
 - ✓ **bufferizzata** (capacità limitata): possibilità di sospensione sia per mittenti che per destinatari.

Remote Procedure Call (RPC)

- È un tipo di comunicazione sincrona, in cui il mittente si sospende fino a che il destinatario non restituisce una risposta (*reply*) al messaggio inviato:
 - il messaggio potrebbe richiedere l'esecuzione di un servizio



Sincronizzazione tra processi

Si è visto che due processi possono interagire per:

- **cooperare**: i processi interagiscono allo scopo di perseguire un obiettivo comune
- **competere**:
 - i processi possono essere logicamente indipendenti, **ma**
 - necessitano della stessa **risorsa** (dispositivo, file, variabile, ecc.) per la quale sono stati dei vincoli di accesso: ad esempio:
 - ✓ gli accessi dei due processi alla risorsa devono escludersi mutuamente nel tempo (*Mutua Esclusione* nell'accesso alla risorsa)

➤ In entrambi i casi è necessario disporre di *strumenti di sincronizzazione*.

Sincronizzazione tra processi

La sincronizzazione permette di imporre vincoli sulle 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 la mutua esclusione dei processi nell'accesso alla risorsa condivisa.

Sincronizzazione tra processi nel modello ad ambiente globale

- Facciamo riferimento a processi che possono condividere variabili (*modello ad ambiente globale*, o a memoria condivisa) per descrivere alcuni strumenti di sincronizzazione tra processi.

In questo ambiente:

- **cooperazione:** lo scambio di messaggi avviene attraverso strutture dati condivise (ad es., mailbox)
- **competizione:** le risorse sono rappresentate da variabili condivise (ad esempio, puntatori a file)

- In entrambi i casi è necessario sincronizzare i processi per coordinarli nell'accesso alla memoria condivisa:

problema della mutua esclusione

Sincronizzazione tra processi nel modello ad ambiente locale

In questo ambiente non vi è la possibilità di condividere memoria:

- Gli accessi alle risorse "condivise" 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:
 - segnali

Esempio: comunicazione in ambiente globale con mailbox di capacità MAX

```
typedef struct {
    coda mbx;
    int num_msg; } mailbox;
```

• Processo mittente:

```
shared mailbox M;
...
main()
{ <crea messaggio m>
  if (M.num_msg < MAX)
  /* send: */
  { inserisci(M.mbx,m);
    M.num_msg++; }
  /* fine send*/
}
```

• Processo destinatario:

```
shared mailbox M;
...
main()
{ if (M.num_msg >0)
  /* receive: */
  { estrai(M.mbx,m);
    M.num_msg--; }
  /* fine receive*/
  <consumo messaggio m>
}
```

Esempio: Esecuzione

inizialmente $M.num_msg=1;$

• Processo mittente:

```
T0: <crea messaggio m>
T1: if (M.num_msg < MAX)
T2: inserisci(M.mbx,m);
```

• Processo destinatario:

```
•
•
•
•
T3: if (M.num_msg > 0)
T4: estrai(M.mbx,m);
T5: M.num_msg--;
```

Sbagliato!
 $M.num_msg=0$

- La correttezza della gestione della mailbox dipende dall'ordine di esecuzione dei processi!
- E' necessario imporre la mutua esclusione dei processi nell'accesso alla variabile M!

Mutua esclusione

- In generale, per garantire la mutua esclusione nell'accesso a variabili condivise, ogni sezione critica è:
 - preceduta da un **prologo** (*entry section*), mediante il quale il processo ottiene l'autorizzazione all'accesso in modo esclusivo
 - seguita da un epilogo (*exit section*), mediante il quale il processo rilascia la risorsa

```
<entry section>
<sezione critica>
<exit section>
```

Il problema della mutua esclusione

In caso di condivisione di risorse (*variabili*) può essere necessario impedire accessi concorrenti alla stessa risorsa (*variabile*).

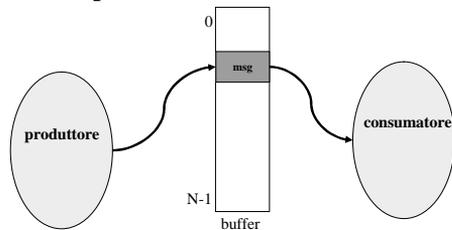
Sezione critica:

è la sequenza di istruzioni mediante le quali un processo accede e può aggiornare variabili condivise.

Mutua esclusione:

ogni processo esegue le proprie sezioni critiche in modo mutuamente esclusivo rispetto agli altri processi

Esempio: Produttore & Consumatore



HP: Buffer (*mailbox*) limitato di dimensione N

produttore&consumatore

- Necessità di garantire la mutua esclusione nell'esecuzione delle sezioni critiche (accesso e aggiornamento del buffer)
- Necessità di sincronizzare i processi:
 - quando il **buffer è vuoto** (il consumatore non può prelevare messaggi)
 - quando il **buffer è pieno** (il produttore non può depositare messaggi)

Produttore/Consumatore

- **Problema:** finchè non si creano le condizioni per effettuare l'operazione di inserimento/prelievo, ogni processo rimane in esecuzione all'interno di un ciclo:

```
while (cont==N); while (cont==0);
```

attesa attiva

- per migliorare l'efficienza del sistema, in alcuni sistemi operativi è possibile utilizzare le primitive:
 - **sleep():** per sospendere il processo che la chiama
 - **wakeup(P):** per riattivare un processo P sospeso (se P non è sospeso, non ha effetto e il segnale di risveglio viene perso)

Produttore & Consumatore: prima soluzione (attesa attiva)

Processo produttore:

```
....
shared int cont=0;
shared msg Buff [N];
main()
{ msg M;
  do
  { produco(&M);
    while (cont==N);
    inserisco(M, Buff);
    cont=cont+1;
  }while(true);
}
```

Processo consumatore:

```
....
shared int contatore=0;
shared msg Buff [N];
main()
{ msg M;
  do
  { while (cont==0);
    prelievo(&M, Buff);
    cont=cont-1;
    consumo(M);
  }while(true);
}
```

Produttore & Consumatore: seconda soluzione

Processo produttore P:

```
....
shared msg Buff [N];
shared int cont=0;

main()
{msg M;
do
{produco(&M);
if(cont==N)sleep();
inserisco(M, Buff);
cont = cont + 1;
if (cont==1)
wakeup(C);
}while(true);
}
```

Processo consumatore C:

```
....
shared msg Buff [N];
shared int cont=0;
main()
{ msg M;
do
{ if(cont==0)sleep();
prelievo(&M, Buff);
cont=cont-1;
if (cont==N-1)
wakeup(P);
consumo(M);
}while(true);
}
```

Produttore & Consumatore seconda soluzione

Possibilità di blocco dei processi: consideriamo la sequenza temporale:

1. `cont=0` (buffer vuoto)
2. C legge `cont`, poi viene descheduled (stato **pronto**)
3. P inserisce un messaggio, `cont++` (`cont=1`)
4. P esegue una `wakeup(C)`: C non è bloccato (è pronto), il segnale è perso
5. C verifica `cont` e **si blocca**
6. P continua a inserire Messaggi, fino a **riempire il buffer**

➤ **Blocco di entrambi i processi**

Soluzione: garantire la mutua esclusione dei processi nell'esecuzione delle sezioni critiche (accesso a `cont`, inserimento e prelievo)

Semafori (Dijkstra, 1965)

Definizione di Semaforo:

- È un tipo di dato astratto al quale sono applicabili solo due operazioni (*primitive*):

- `wait (s)`
- `signal (s)`

- Al semaforo `s` sono associate:

- una variabile intera `s.value` non negativa con valore iniziale ≥ 0 .
- una coda di processi `s.queue`

➤ Il semaforo può essere condiviso da 2 o più processi per risolvere problemi di sincronizzazione (es. mutua esclusione)

Produttore e consumatore

- È necessario garantire la mutua esclusione dei processi quando aggiornano le variabili condivise (**Buff** e **cont**), cioè nelle **sezioni critiche**

Soluzione: uso dei **semafori**

Operazioni sui semafori: definizione

```
void wait(s)
{ if (s.value == 0)
  <il processo viene sospeso e il suo
  descrittore inserito in s.queue>
  else s.value=s.value-1;
}
```

```
void signal(s)
{ if (<esiste un processo nella coda s.queue>
  <il suo descrittore viene estratto da s.queue
  e il suo stato modificato in pronto>
  else s.value=s.value+1;
}
```

wait/signal

- **Wait:**

- la **wait(s)**, in caso di **s.value=0**, implica la sospensione del processo che la esegue (stato running -> waiting) nella coda **s.queue** associata al semaforo.

- **Signal:**

- l'esecuzione della **signal (s)** non comporta concettualmente nessuna modifica nello stato del processo che l'ha eseguita, ma può causare il risveglio a un processo waiting nella coda **s.queue**.
- La scelta del processo da risvegliare avviene secondo una politica FIFO (il primo processo della coda).

➤ wait e signal agiscono su variabili condivise e pertanto sono a loro volta **sezioni critiche!**

Mutua esclusione con semafori: Esempio

- Consideriamo due processi P1 e P2 che condividono una struttura dati D sulla quale vogliamo quindi imporre il vincolo di mutua esclusione:

shared data D;

P1: ... /*sezione critica: */ Aggiorna1(&D); /*fine sez.critica: */ ...	P2: ... /*sezione critica: */ Aggiorna2(&D); /*fine sez.critica: */ ...
---	---

☞ Aggiorna1 e Aggiorna2 sono sezioni critiche e devono essere eseguite in modo mutuamente esclusivo.

Atomicità di wait e signal

Affinchè sia rispettato il vincolo di mutua esclusione dei processi nell'accesso al semaforo (mediante wait/signal), wait e signal devono essere operazioni indivisibili (azioni **atomiche**):

- durante un'operazione sul semaforo (**wait** o **signal**) nessun altro processo può accedere al semaforo fino a che l'operazione è completata o bloccata.

➤ Il Sistema Operativo realizza **wait** e **signal** come operazioni **non interrompibili** (ad es. system call).

Mutua esclusione con semafori: Esempio

- **Soluzione:** uso di un semaforo (binario) **mutex**, il cui valore è inizializzato a 1.

shared data D;
semaphore mutex=1;
mutex.value=1;

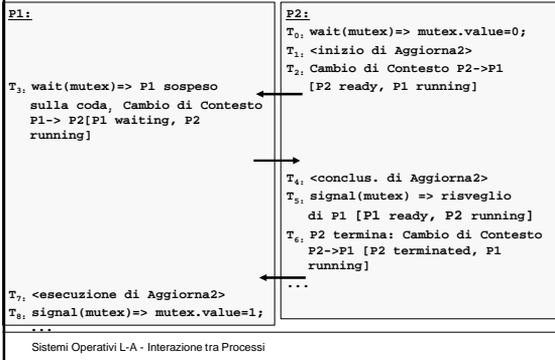
P1: ... wait(mutex); Aggiorna1(&D); signal(mutex); ...	P2: ... wait(mutex) Aggiorna2(&D); signal(mutex); ...
--	---

- Il valore del semaforo **mutex** può assumere soltanto due valori (0,1): semaforo **binario**

➤ la soluzione è sempre **corretta**, indipendentemente dalla sequenza di esecuzione dei processi (e dallo scheduling della CPU).

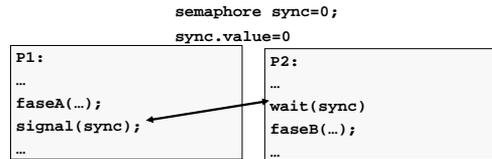
Mutua esclusione con semafori: esecuzione

- **Ad es:** verifichiamo la seguente sequenza di esecuzione:



Sincronizzazione di processi cooperanti

- **Soluzione:** introduco un semaforo **sync**, inizializzato a 0.

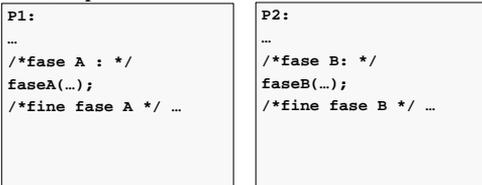


- se P2 esegue la **wait** prima della terminazione della fase A, P2 viene sospeso;
- quando P1 termina la fase A, può:
 - Sbloccare P1, oppure
 - Portare il valore del semaforo a 1 (se P2 non e' ancora arrivato alla **wait**).

Sincronizzazione di processi cooperanti

- Mediante i semafori possiamo imporre **vincoli temporali** sull'esecuzione di processi cooperanti.

Ad esempio:



Obiettivo: Vogliamo imporre che l'esecuzione della fase A (in P1) preceda sempre l'esecuzione della fase B (in P2)

Produttore & Consumatore con semafori

- Problema di mutua esclusione:
 - produttore e consumatore non possono accedere contemporaneamente al buffer:
 - introduciamo il semaforo **binario mutex**, con valore iniziale a 1;
 - Problema di sincronizzazione:
 - il produttore non può scrivere nel buffer se questo è pieno:
 - introduciamo il semaforo **vuoto**, con valore iniziale a N;
 - il consumatore non può leggere dal buffer se questo è vuoto:
 - introduciamo il semaforo **pieno**, con valore iniziale a 0;
- Sistemi Operativi L-A - Interazione tra Processi 44

Produttore & Consumatore con semafori

```
shared msg Buff [N];
shared semaforo mutex; mutex.value=1;
shared semaforo pieno; pieno.value=0
shared semaforo vuoto; vuoto.value=N
```

```
/* Processo produttore P:*/
main()
{msg M;
do
{produco(&M);
wait(vuoto);
wait(mutex);
inserisco(M, Buff);
signal(mutex);
signal(pieno);
}while(true);}
```

```
/* Processo consumatore C:*/
main()
{ msg M;
do
{ wait(pieno);
wait(mutex);
prelievo(&M, Buff);
signal(mutex);
signal(vuoto);
consumo(M);
}while(true);}
```

Strumenti di Sincronizzazione

- **Semafori:**
 - consentono una efficiente realizzazione di politiche di sincronizzazione tra processi
 - La correttezza della realizzazione e' completamente a carico del programmatore
- **Alternative:** esistono strumenti di piu' alto livello (costrutti di linguaggi di programmazione) che eliminano a priori il problema della mutua esclusione sulle variabili condivise
 - Variabili Condizione
 - Regioni critiche
 - Monitor
 - ...