

Interazione, sincronizzazione e comunicazione tra processi

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*

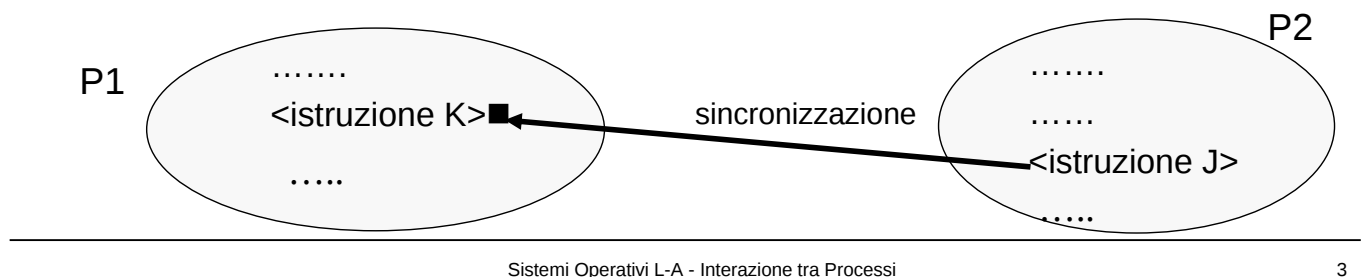
- **cooperanti**: i processi interagiscono **volontariamente** per raggiungere **obiettivi comuni** (fanno parte della stessa applicazione)
- **in competizione**: i processi, in generale, non fanno parte della stessa applicazione, ma **interagiscono indirettamente per l'acquisizione di risorse comuni**

Processi interagenti

L'interazione può avvenire mediante due meccanismi:

- **Comunicazione**: scambio di informazioni tra i processi interagenti
- **Sincronizzazione**: imposizione di **vincoli temporali, assoluti o relativi**, 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

Realizzazione dell'interazione: dipende dal modello di esecuzione per i processi

- **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 (come nei *thread*)
 - **variabili condivise e strumenti di sincronizzazione** (ad esempio, **semafori**)

Processi interagenti mediante scambio di messaggi

Facciamo riferimento al *modello ad ambiente locale*:

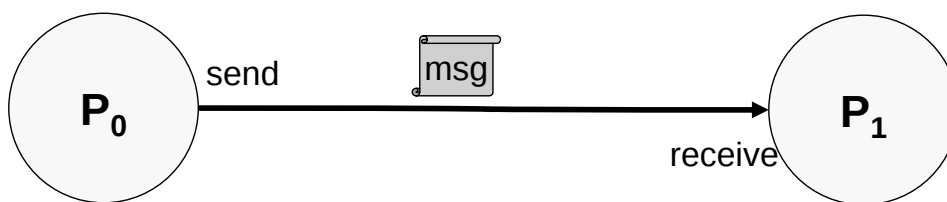
- ❑ ***non vi è memoria condivisa***
- ❑ i processi possono interagire mediante ***scambio di messaggi: comunicazione***

➤ ***SO offre meccanismi a supporto della comunicazione tra processi (Inter Process Communication)***

Operazioni Necessarie

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

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, molti-a-molti
- capacità
- modalità di creazione: automatica, non automatica

Meccanismi di comunicazione tra processi

Aspetti caratterizzanti:

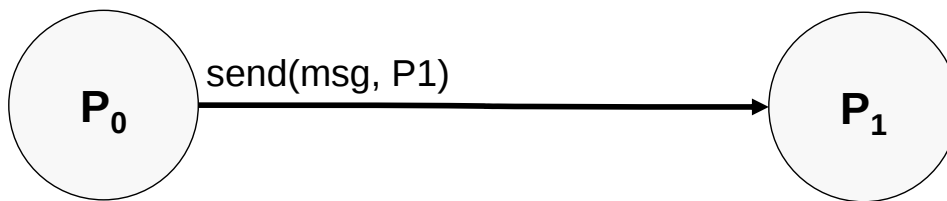
- ***caratteristiche del canale***
- ***caratteristiche del messaggio***
 - ✓ dimensione
 - ✓ tipo
- ***tipo di comunicazione***
 - ✓ diretta o indiretta
 - ✓ simmetrica o asimmetrica
 - ✓ bufferizzata o no
 - ✓ ...

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>)

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 **naming** esplicito del mittente

Comunicazione asimmetrica

Processo produttore P:

```
....  
main()  
{ msg M;  
  do  
  { produco(&M);  
    ...  
    send(C, M);  
  }while(!fine);  
}
```

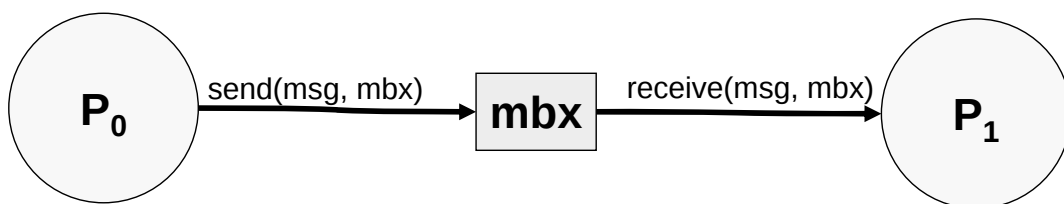
Processo consumatore C:

```
....  
main()  
{ msg M; pid id;  
  do  
  { receive(&id, &M);  
    ...  
    consumo(M);  
  }while(!fine);  
}
```

Comunicazione asimmetrica:

- destinatario non è obbligato a conoscere l'identificatore del mittente

Comunicazione indiretta



I processi cooperanti ***non sono tenuti a conoscersi*** reciprocamente e si scambiano messaggi depositandoli/prelevandoli da una ***mailbox condivisa***

- ***mailbox (o porta)*** come ***risorsa astratta condivisibile*** da più processi che funge da contenitore dei messaggi

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:
 - p0: send(query, mbx)
 - p1: send(answ, mbx)
- per ogni coppia di processi possono esistere più canali (uno per ogni mailbox condivisa)

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



Buffering del canale

Capacità nulla: non vi è accodamento perché 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 (solitamente sono) **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**: **send è sospensiva**
 - se la coda **è vuota**: **receive può essere sospensiva**
- **Capacità illimitata:** lunghezza della coda teoricamente infinita. L'invio sul canale non è sospensivo

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, ...) per la quale sono stati dei vincoli di accesso.
Ad esempio:
 - ✓ gli accessi di due processi a una risorsa **devono escludersi mutuamente nel tempo**

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

Sincronizzazione tra processi

Sincronizzazione permette di imporre vincoli temporali 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 locale

Mancando la possibilità di condividere memoria:

- ❑ Gli accessi alle risorse "condivise" vengono controllati e coordinati da SO
- ❑ La sincronizzazione avviene mediante meccanismi offerti da SO che consentono la **notifica di "eventi" asincroni** (privi di contenuto informativo) tra un processo ed altri
 - **segnali UNIX**

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

- ❑ **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

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

HP: a T_0 $M.num_msg=1$;

Processo mittente:

```
 $T_0$ : <crea messaggio m>  
 $T_1$ : if (M.num_msg < MAX)  
 $T_2$ : inserisci(M.mbx,m);
```

Processo destinatario:

```
.  
. .  
 $T_3$ : if (M.num_msg > 0)  
 $T_4$ : estrai(M.mbx,m);  
 $T_5$ : M.num_msg--;  
...
```

Sbagliato!
 $M.num_msg=0$

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

Il problema della mutua esclusione

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

Sezione critica

sequenza di istruzioni mediante la quale 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

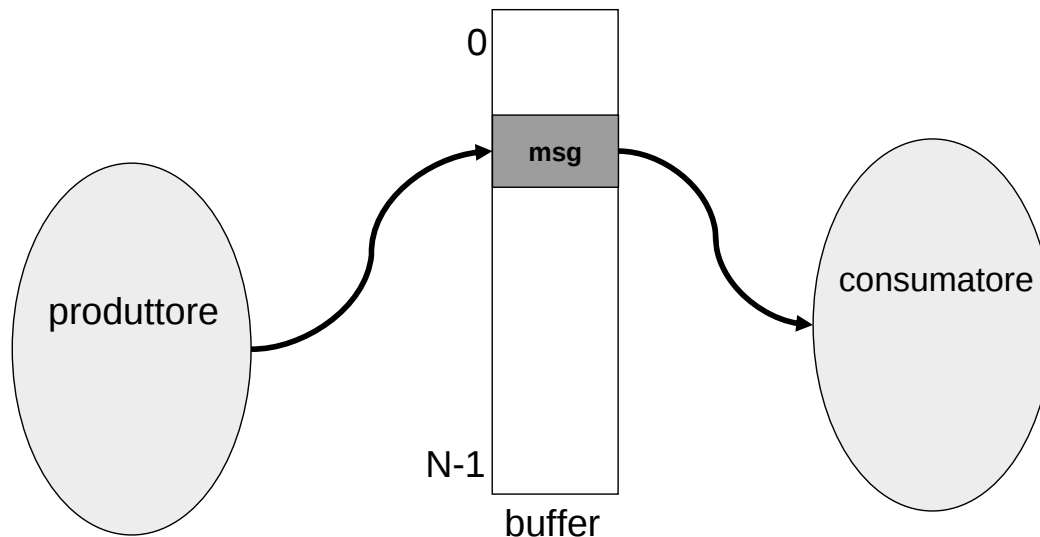
Mutua esclusione

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>
```

Esempio: produttore & consumatore



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

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

Produttore&consumatore

Problema: finché 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 SO è possibile utilizzare le system call:

- **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: 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: ad esempio, consideriamo la sequenza temporale:

1. `cont=0` (buffer vuoto)
2. **C** legge `cont`, poi viene *descheduled prima di sleep* (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 (deadlock)**

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

Possibile soluzione: semafori (Dijkstra, 1965)

Definizione di semaforo

- **Tipo di dato astratto** condiviso fra più processi al quale sono applicabili solo due operazioni (*system call*):
 - **wait (s)**
 - **signal (s)**
- A una variabile *s* di tipo *semaforo* sono associate:
 - una variabile intera *s.value* non negativa con valore iniziale ≥ 0
 - una coda di processi *s.queue*

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

System call sui semafori: definizione

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

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

wait() / signal()

- **Wait**

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**

- 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!**

Atomicità di wait() e signal()

Affinché 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 non è **completa** o **bloccata** (sospensione nella coda)

➤ SO realizza **wait()** e **signal()** come operazioni non interrompibili (*system call*)

Esempio di mutua esclusione con semafori

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**

Esempio di mutua esclusione con semafori

Soluzione: uso di **un semaforo (binario) mutex**, il cui valore è inizializzato a 1 (e può assumere soltanto due valori: 0 e 1)

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

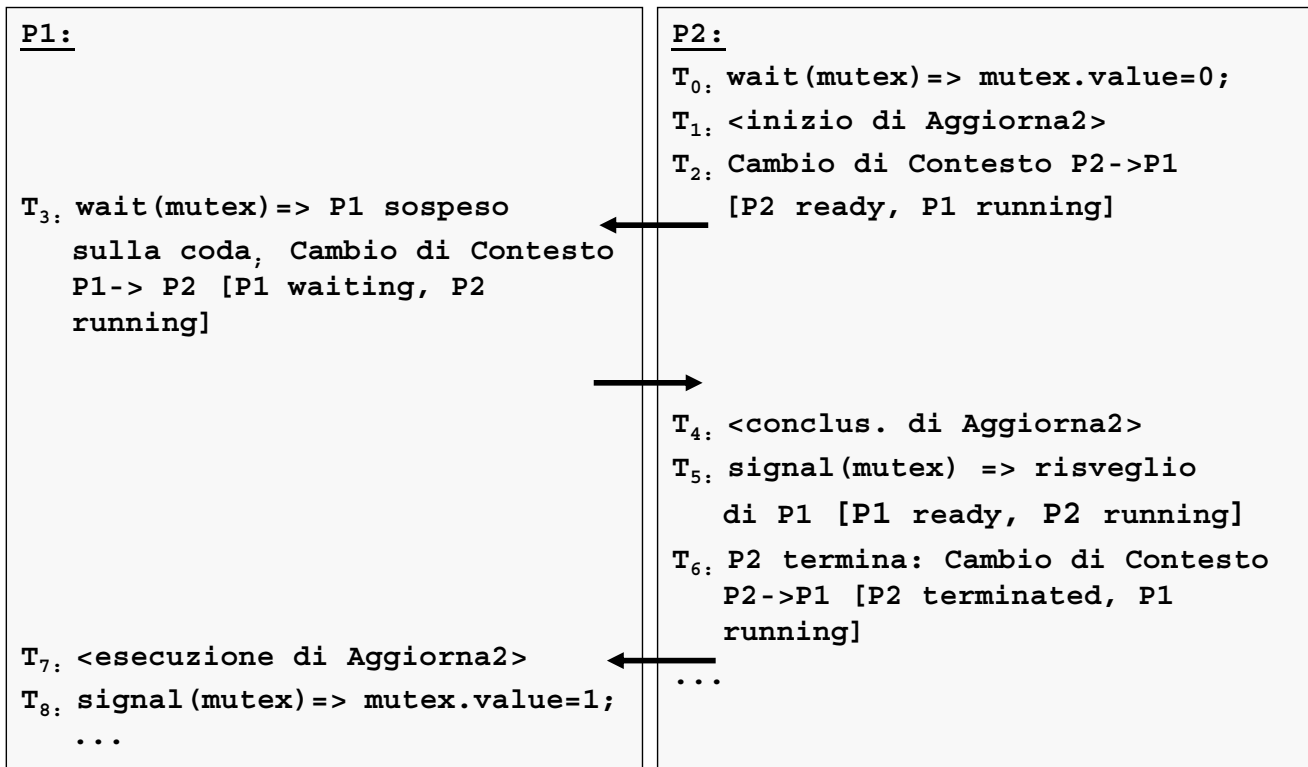
```
P1:
...
wait(mutex);
Aggiorna1(&D);
signal(mutex);
...
```

```
P2:
...
wait(mutex);
Aggiorna2(&D);
signal(mutex);
...
```

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

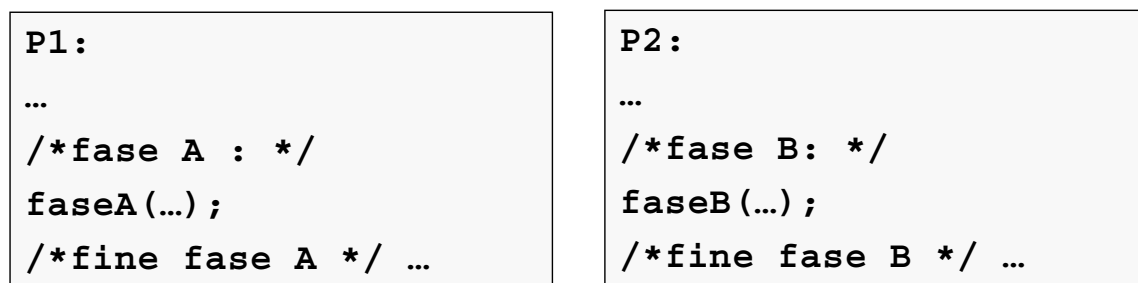
Mutua esclusione con semafori: esecuzione

Ad esempio, verifichiamo la seguente sequenza di esecuzione:



Sincronizzazione di processi cooperanti

Mediante semafori possiamo anche imporre **vincoli temporali** sull'esecuzione di processi cooperanti. Ad esempio:

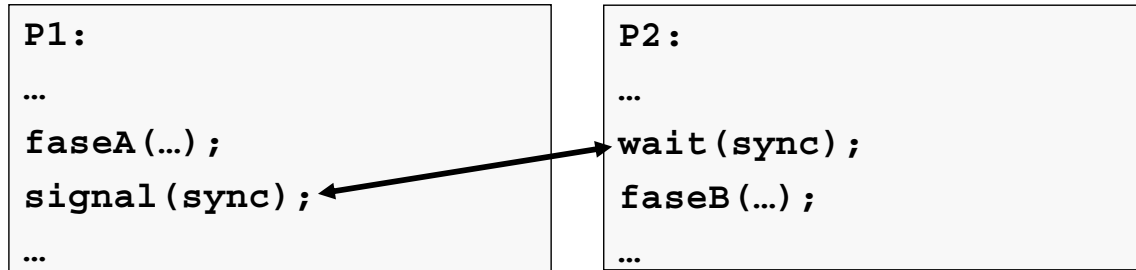


Obiettivo: vogliamo imporre che l'esecuzione della fase A (in P1) preceda sempre l'esecuzione della fase B (in P2)

Sincronizzazione di processi cooperanti

Soluzione: si introduce un **semaforo** *sync*, **inizializzato a 0**

```
semaphore sync=0;  
sync.value=0
```



- se P2 esegue la `wait()` prima della terminazione della fase A, P2 viene sospeso;
- quando P1 termina la fase A, può sbloccare P2, oppure portare il valore del semaforo a 1 (se P2 non è ancora arrivato alla `wait`)

Produttore & consumatore con semafori

- Problema di **mutua esclusione**
 - produttore e consumatore non possono accedere contemporaneamente al buffer
 - semaforo **binario mutex**, con valore iniziale a 1
- Problema di **sincronizzazione**
 - produttore non può scrivere nel buffer se pieno
 - semaforo **vuoto**, con valore iniziale a **N**;
valore dell'intero associato a **vuoto** rappresenta il numero di elementi liberi nel buffer
 - consumatore non può leggere dal buffer se vuoto
 - semaforo **pieno**, con valore iniziale a **0**;
valore dell'intero associato a **pieno** rappresenta il numero di elementi occupati nel buffer

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**, anche complesse, tra processi
- correttezza della realizzazione **completamente a carico del programmatore**

Alternative: esistono **strumenti di più alto livello** (costrutti di linguaggi di programmazione) che eliminano a priori il problema della mutua esclusione sulle variabili condivise

- Variabili condizione
- Regioni critiche
- Monitor
- **synchronized** in Java
- ...

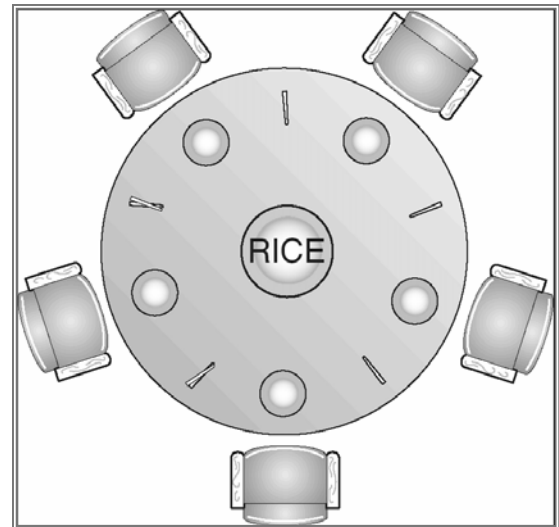
Problema dei dining-philosophers

Problema molto noto in letteratura

Risorse condivise:

- ❑ Ciotola di riso (data set)
- ❑ Semafori bastoncini[5]
inizializzati a 1

Provare a pensare a soluzioni di sincronizzazione mediante il solo uso di semafori e possibili problemi

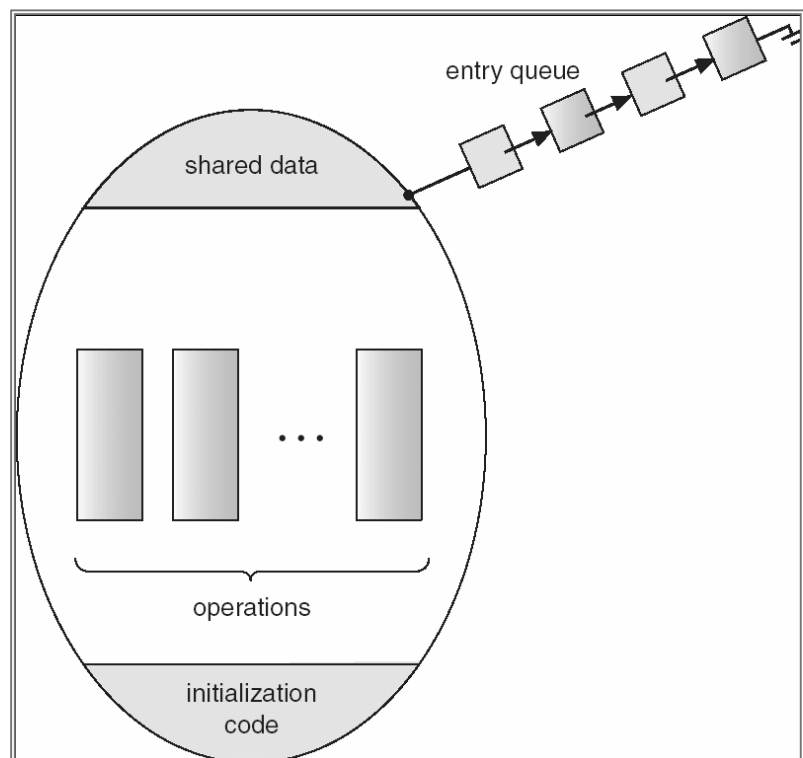


Si possono verificare situazioni di blocco indefinito?

Meccanismi alternativi di sincronizzazione: monitor

Ve ne occuperete in corsi successivi

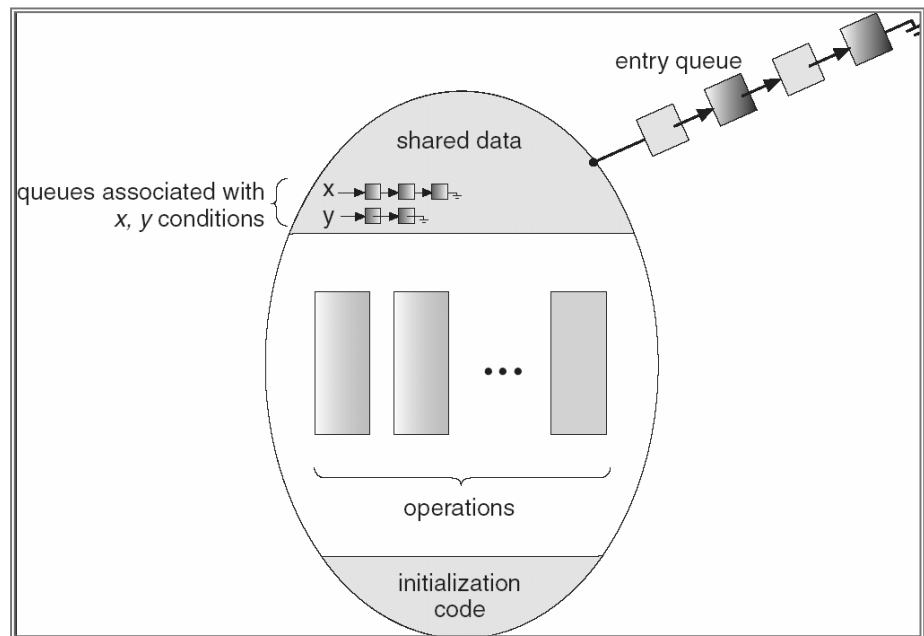
Coda di accesso
regolata e disciplinata verso i
dati condivisi



Meccanismi alternativi di sincronizzazione: monitor con variabili condizione

Ve ne occuperete in
corsi successivi

Anche **code
differenziate**
associate a
**condizioni di
controllo**
verificate a
runtime



Comunicazione tra processi UNIX

Interazione tra processi UNIX

Processi UNIX non possono condividere memoria
(*modello ad ambiente locale*)

Interazione tra processi può avvenire

- mediante la **condivisione di file**
 - complessità: realizzazione della sincronizzazione tra i processi
- attraverso **specifici strumenti di Inter Process Communication**:
 - tra processi **sulla stessa macchina**
 - ✓ **pipe** (tra processi della stessa gerarchia)
 - ✓ **fifo** (qualunque insieme di processi)
 - tra processi in nodi diversi della stessa **rete**:
 - ✓ **socket**

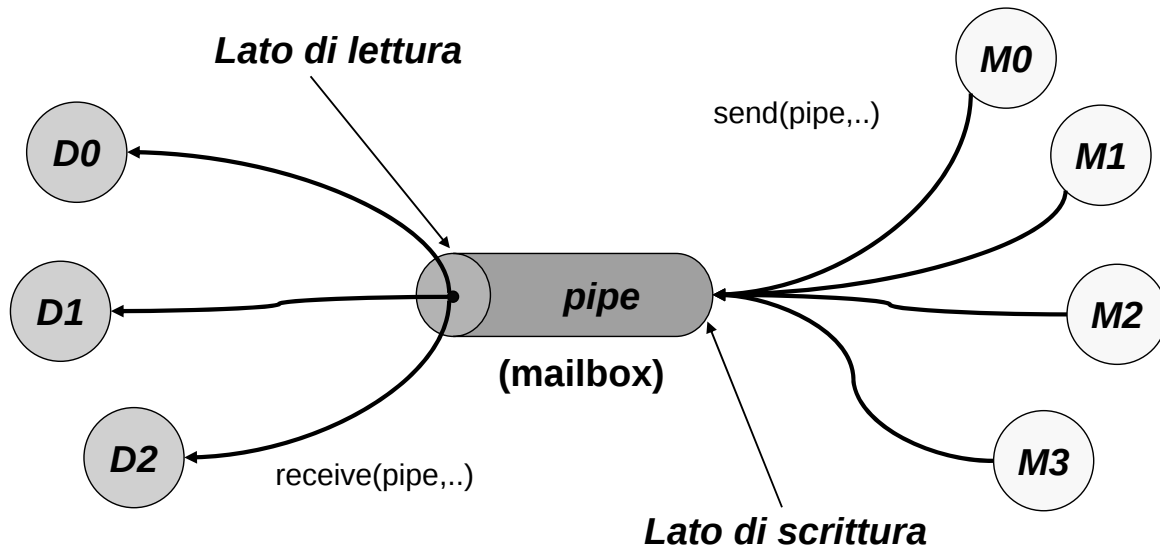
pipe

La pipe è un **canale di comunicazione** tra processi:

- **unidirezionale**: accessibile mediante due estremi distinti, uno di lettura e uno di scrittura
- (teoricamente) **molti-a-molti**:
 - più processi possono **spedire messaggi** attraverso la stessa pipe
 - più processi possono **ricevere messaggi** attraverso la stessa pipe
- **capacità limitata**:
 - in grado di gestire **l'accodamento di un numero limitato di messaggi**, gestiti in modo FIFO. Limite stabilito dalla **dimensione della pipe** (es. 4096B)

Comunicazione attraverso pipe

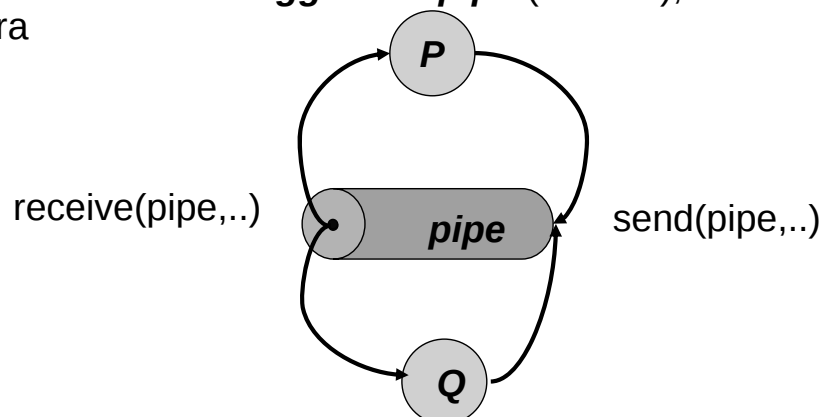
Mediante la pipe, la comunicazione tra processi è **indiretta** (senza naming esplicito): **mailbox**



Pipe: unidirezionalità/bidirezionalità

Uno stesso processo può:

- sia **depositare messaggi nella pipe** (*send*), mediante il lato di scrittura
- sia **prelevare messaggi dalla pipe** (*receive*), mediante il lato di lettura



la pipe può anche consentire una **comunicazione** **"bidirezionale"** tra P e Q (ma va rigidamente disciplinata)

System call `pipe`

Per creare una pipe:

```
int pipe(int fd[2]);
```

`fd` è il puntatore a un **vettore di 2 file descriptor**, che verranno inizializzati dalla system call in caso di successo:

- `fd[0]` rappresenta il lato di lettura della pipe
- `fd[1]` è il lato di scrittura della pipe

la system call `pipe()` restituisce:

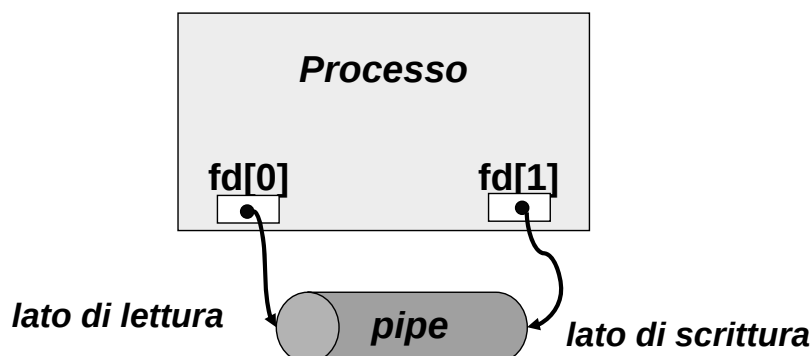
- un valore negativo, in caso di fallimento
- 0, se ha successo

Creazione di una pipe

Se `pipe(fd)` ha successo:

vengono allocati due nuovi elementi nella tabella dei file aperti del processo e i rispettivi file descriptor vengono assegnati a `fd[0]` e `fd[1]`

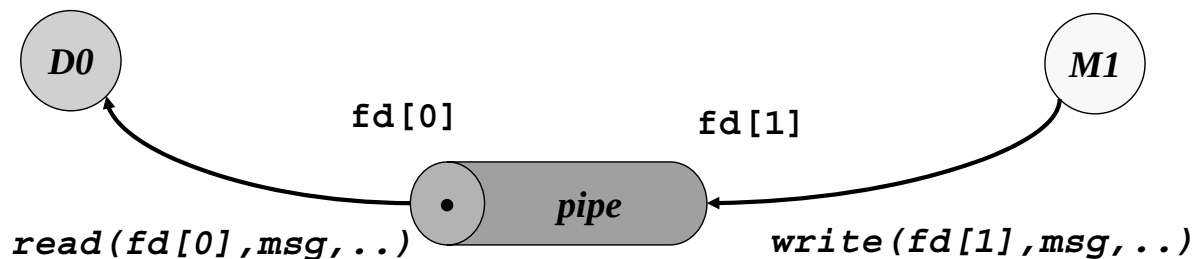
- `fd[0]`: lato di lettura (*receive*) della pipe
- `fd[1]`: lato di scrittura (*send*) della pipe



Omogeneità con i file

Ogni lato di accesso alla pipe è visto dal processo in **modo omogeneo al file** (file descriptor)

- si può accedere alla pipe mediante le system call di lettura/scrittura su file `read()`, `write()`



- `read()`: operazione di ricezione
- `write()`: operazione di invio

Sincronizzazione processi comunicanti

Il canale (**pipe**) ha **capacità limitata**. Come nel caso di produttore/consumatore è necessario sincronizzare i processi:

- se la **pipe** è **vuota**: un processo che **legge si blocca**
 - se la **pipe** è **piena**: un processo che **scrive si blocca**
- **Sincronizzazione automatica**: `read()` e `write()` sono implementate **in modo sospensivo** dal SO

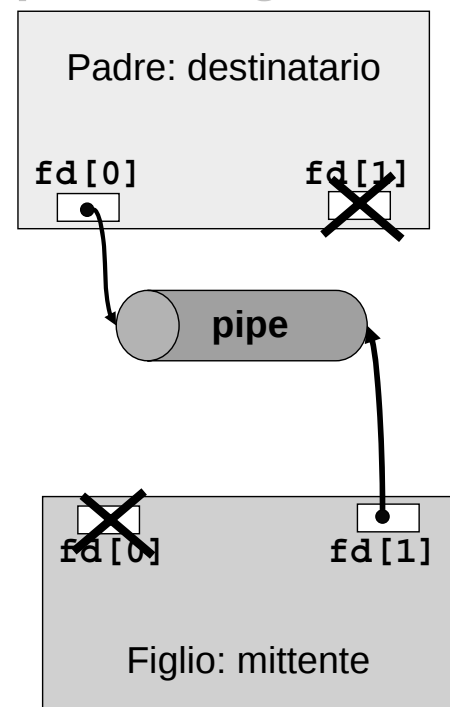
Quali processi possono comunicare mediante pipe?

Per mittente e destinatario *il riferimento al canale di comunicazione è un array di file descriptor*:

- Soltanto i **processi appartenenti a una stessa gerarchia** (cioè, che hanno un **antenato** in comune) possono scambiarsi messaggi mediante pipe. Ad esempio, possibilità di comunicazione:
 - tra **processi fratelli** (che **ereditano** la pipe dal processo padre)
 - tra un processo **padre** e un processo **figlio**;
 - tra **nonno** e **nipote**
 - ...

Esempio: comunicazione tra padre e figlio

```
main()
{int pid;
 char msg[]="ciao babbo";
 int fd[2];
 pipe(fd);
 pid=fork();
 if (pid==0)
 {/* figlio */
  close(fd[0]);
  write(fd[1], msg, 10);
  ...}
 else /* padre */
 { close(fd[1]);
  read(fd[0], msg, 10);
  ...
 }}
```



Ogni processo chiude il lato della pipe che non usa

Chiusura di pipe

Ogni processo può chiudere ***un estremo della pipe*** con la system call `close()`

- la comunicazione non è più possibile su di un estremo della pipe ***quando tutti i processi che avevano visibilità di quell'estremo*** hanno compiuto una `close()`

Se un processo P tenta:

- ***lettura da una pipe vuota il cui lato di scrittura è effettivamente chiuso***: `read` ritorna 0
- ***scrittura da una pipe il cui lato di lettura è effettivamente chiuso***: `write` ritorna -1, e il segnale ***SIGPIPE*** viene inviato a P (*broken pipe*)

Esempio

```
/* Sintassi: progr N
padre(destinatario) e figlio(mittente) si scambiano una
sequenza di messaggi di dimensione (DIM) costante; la
lunghezza della sequenza non è nota a priori;
destinatario interrompe sequenza di scambi di messaggi
dopo N secondi */
```

```
#include <stdio.h>
#include <signal.h>
#define DIM 10

int fd[2];
void fine(int signo);
void timeout(int signo);
```

Esempio

```
main(int argc, char **argv)
{int pid, N; char messaggio[DIM]="ciao ciao ";
  if (argc!=2)
  {   printf("Errore di sintassi\n");
      exit(1);}
  N=atoi(argv[1]);
  pipe(fd);
  pid=fork();
  if (pid==0) /* figlio */
  {   signal(SIGPIPE, fine);
      close(fd[0]);
      for(;;)
          write(fd[1], messaggio, DIM);
  }
```

Esempio

```
else if (pid>0) /* padre */
{
    signal(SIGALRM, timeout);
    close(fd[1]);
    alarm(N);
    for(;;)
    {
        read(fd[0], messaggio, DIM);
        write(1, messaggio, DIM);
    }
}
}/* fine main */
```

Esempio

```
/* definizione degli handler dei segnali */
void timeout(int signo)
{ int stato;
  close(fd[0]); /* chiusura effettiva del lato di lettura*/
  wait(&stato);
  if ((char)stato!=0)
    printf("Termin invol figlio (segnale %d)\n",
          (char)stato);
  else printf("Termin volont Figlio (stato %d)\n",
             stato>>8);
  exit(0);
}

void fine(int signo)
{ close(fd[1]);
  exit(0);
}
```

System call dup

Per *duplicare un elemento della tabella dei file aperti di processo*:

`int dup(int fd)`

□ `fd` è il file descriptor del file da duplicare

L'effetto di `dup()` è copiare l'elemento `fd` della tabella dei file aperti nella **prima posizione libera** (quella con l'indice minimo tra quelle disponibili)

- Restituisce il **nuovo file descriptor** (del file aperto copiato), oppure -1 (in caso di errore)

Esempio: mediante `dup()` ridirigere `stdout` su pipe

```
main()
{ int pid, fd[2]; char msg[3]="bye";
  pipe(fd);
  pid=fork();
  if (!pid) /* processo figlio */
  {   close(fd[0]); close(1);
      dup(fd[1]); /* ridirigo stdout sulla pipe */
      close(fd[1]);
      write(1,msg, sizeof(msg)); /*scrivo su pipe*/
      close(1);
  }else /*processo padre
  {   close(fd[1]);
      read(fd[0], msg, 3);
      close(fd[0]);}}
```

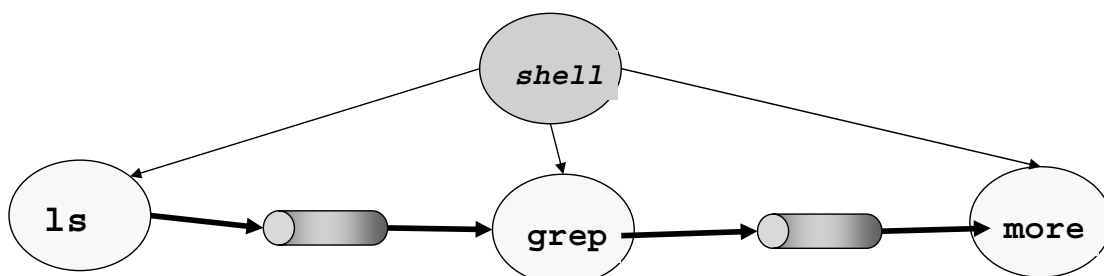
`dup()` & piping

Tramite `dup()` si può realizzare il piping di comandi. Ad esempio:

`ls -lR | grep Jun | more`

➔ Vengono creati 3 processi (uno per ogni comando), in modo che:

- ❑ `stdout` di `ls` sia ridiretto nello `stdin` di `grep`
- ❑ `stdout` di `grep` sia ridiretto nello `stdin` di `more`



Esempio: piping di 2 comandi senza argomenti

```
/* sintassi: programma com1 com2  significa:
   com1 | com2  */

main(int argc, char **argv)
{ int pid1, pid2, fd[2], i, status;
  pipe(fd);
  pid1=fork();
  if (!pid1) /* primo processo figlio: com2 */
  {   close(fd[1]);
      close(0);
      dup(fd[0]); /* ridirigo stdin sulla pipe */
      close(fd[0]);
      execlp(argv[2], argv[2], (char *)0);
      exit(-1);
  }
}
```

```
else /*processo padre
{ pid2=fork();
  if (!pid2) /* secondo figlio: com1 */
  {   close(fd[0]);
      close(1);
      dup(fd[1]);
      close(fd[1]);
      execlp(argv[1], argv[1], (char *)0);
      exit(-1);
  }
  for (i=0; i<2;i++)
  {   wait(&status);
      if((char)status!=0)
          printf("figlio terminato per segnale%d\n",
              (char)status);
  }
  exit(0);
}
```

Pipe: possibili svantaggi

Il meccanismo delle pipe ha **due svantaggi**:

- ❑ consente la comunicazione **solo tra processi in relazione di parentela**
- ❑ **non è persistente**: pipe viene distrutta quando terminano tutti i processi che hanno accesso ai suoi estremi

Per realizzare la comunicazione tra una coppia di **processi non appartenenti alla stessa gerarchia**?



fifo

È una **pipe con nome** nel file system:

- ❑ canale **unidirezionale** del tipo **first-in-first-out**
- ❑ è **rappresentata da un file** nel file system:
persistenza, visibilità potenzialmente globale
- ❑ ha un proprietario, un insieme di diritti ed una lunghezza
- ❑ è creata dalla system call **mkfifo()**
- ❑ è aperta e acceduta con le stesse system call dei file

Per creare una fifo (pipe con nome):

```
int mkfifo(char* pathname, int mode);
```

- ❑ **pathname** è il nome della fifo
- ❑ **mode** esprime i permessi

restituisce 0, in caso di successo, un valore negativo, in caso contrario

Apertura/chiusura di fifo

Una volta creata, ***fifo può essere aperta*** (come tutti i file) mediante **`open()`** . Ad esempio, un processo destinatario di messaggi:

```
int fd;  
fd=open("myfifo", O_RDONLY);
```

Per chiudere una fifo, si usa **`close()`**:

```
close(fd);
```

Per eliminare una fifo, si usa **`unlink()`**:

```
unlink("myfifo");
```

Accesso a fifo

Una volta aperta, fifo può essere acceduta (come tutti i file) mediante **`read()`** / **`write()`** . Ad esempio, un processo destinatario di messaggi:

```
int fd;  
char msg[10];  
fd=open("myfifo", O_RDONLY);  
read(fd, msg, 10);
```