05 - Processi Segnali Pipe

Processi interagenti

Classificazione:

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

L'interazione puo' 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 puo' essere eseguita soltanto dopo l'istruzione j del processo P2

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

- **modello ad ambiente locale**: non c'e' divisione di variabili (processo pesante)
  - comunicazione avviene attraverso scambio di messaggi
  - sincronizzazione avviene mediante scambio di eventi (Segnali)
- **modello ad ambiente globale**: piu' processi possono condividere lo stesso spazio di indirizzamento --> possibilita' di condividere variabili (come nei thread)
  - variabili condivise e relativi strumenti di sincronizzazione (ad esempio, lock e semafori)

Processi interagenti mediante scambio di messaggi

Facciamo riferimento al modello ad ambiente locale:

- non vi e' memoria condivisa
- i processi possono interagire mediante scambio di messaggi: comunicazione
  Spesso SO offre meccanismi a supporto della comunicazione tra processi (Inter Process Communication - IPC)

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
- capacita'
- modalita' 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 prelevera' il messaggio
  send(Mailbox, msg)

Comunicazione diretta

Il canale e' 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 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 piu' processi, che funge da contenitore dei messaggi

Proprieta'

- il canale di comunicazione e' rappresentato dalla mailbox (non viene creato automaticamente)
- il canale puo' essere associato a piu' di due processi:
  - mailbox di sistema: molti-a-molti (come individuare il processo destinatario di un messaggio)
  - mailbox del processo destinatario: molti-a-uno
- canale bidirezionale:
  - p0: send(query, mbx)
  - p1: send(answ, mbx)
- per ogni coppia di processi possono esistere piu' canali (uno per ogni mailbox condivisa)

Buffering del canale

Ogni canale di comunicazione e' caratterizzato da una capacita': numero dei messaggi che e' in grado di gestire contemporaneamente.

Gestione usuale secondo politica FIFO:

- i messaggi vengono posti in una coda in attesa di essere ricevuti
- la lunghezza massima della coda rappresenta la capacita' del canale

Caso semplificato con capacita' nulla: non ci e' accomodamento perche' il canale non e' in grado di gestire messaggi in attesa

- processo mittente e destinatario devono sincronizzarsi all'atto di spedire (send)/ricevere (receive) il messaggio: comunicazione asincrona o rendez vous
- send e receive possono essere (solitamente sono) sospensive

Capacita' limitata: esiste un limite N alla dimensione della coda

- se la coda non e' piena, un nuovo messaggio viene posto in fondo
- se la coda e' piena: send e' sospensiva
- se la coda e' vuota: receive puo' essere sospensiva

Capacita' illimitata: lunghezza della coda teoricamente infinita. L'invio sul canale non e' sospensivo.

Sincronizzazione tra processi

Si e' visto che due processi possono interagire per

- **cooperare**: i due 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 imposti dei vincoli di accesso. Ad esempio:
  - gli accessi di due processi a una risorsa devono escludersi mutuamente nel tempo

- In entrambi i casi e' necessario disporre di **strumenti di sincronizzazione**

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 possibilita' 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 (di solito privi di contenuto informativo o con contenuto minimale) tra un processo ed altri
  - segnali UNIX

Sincronizzazione tra processi nel modello ad ambiente globale

Facciamo riferimento ai 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 rappresente da variabili condivise (ad esempio, puntatori a file)

In entrambi i casi e' necessario sincronizzare i processi per coordinarli nell'accesso alla memoria condivisa: **problema della mutua esclusione**.

Il problema della mutua esclusione

In caso di condivisione di risorse (variabili) puo' essere necessario impedire accessi concorrenti alla stessa risorsa.

- **Sezione critica**: sequenza di istruzioni mediante la quale un processo accede e puo' aggiornare variabili condivise
- **Mutua esclusione**: ogni processo esegue le proprie sezioni critiche in modo esclusivo rispetto agli altri processi

In generale per garantire la mutua esclusione nell'accesso a variabili condivise, ogni sezione critica e':

- 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

Esempio: produttore & consumatore

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

Problema: finche' 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);`

Per migliorare l'efficienza del sistema, in alcuni SO e' possibile utilizzare system call del tipo:

- **dormo****()** per sospendere il processo che la chiama (stato di waiting e spreco di CPU evitato)
- **sveglia(P)** per riattivare un processo P sospeso (se P non e' sospeso, non ha effetto e il segnale di risveglio viene perso)

Possibile soluzione: semafori (Dijkstra, 1965)

Definizione di semaforo

- Tipo di dato astratto condiviso fra piu' processi al quale sono applicabili due operazioni (system call a esecuzione non interrompibile): wait(s) singal(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 puo' essere condiviso da 2 o piu' processi per risolvere problemi di sincronizzazione (es. mutua esclusione)

wait()/signal()

- **wait()**
  - in caso di s.value=0, implica la **sospensione** del processo che la segue (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 puo' causare il **risveglio** di 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**!

Atomicita' di wait() e signal()

Affinche' sia rispettato il vincolo della 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 puo' accedere al semaforo fino a che l'operazione non e' completa o bloccata (Sospensione nella corda)

SO che mette a disposizione le primitive di Dijkstra deve realizzare wait() e signal() come operazioni non interrompibili (system call)

Sincronizzazione di processi cooperanti

Mediante semafori possiamo anche imporre vincoli temporali sull'esecuzione di processi cooperanti.

Obiettivo: vogliamo imporre che l'esecuzione della fase A (in P1) preceda sempre l'esecuzione della fase B (in P2)
Soluzione: si introduce 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, puo' sbloccare P1, oppure portare il valore del semaforo a 1 (se P2 non e' 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 sinconizzazione
    - produttore non puo' scrivere nel buffer se pieno
        - semaforo vuoti, con valore iniziale a N; valore dell'interno associato a vuoto rappresenta il numero di elementi liberi nel buffer
    - consumatore non puo' leggere dal buffer se vuoto
        - semaforo pieno, con valore iniziale a 0; valore dell'interno associato a pieno rappresenta il numero di elementi occupati nel buffer

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 piu' alto livello (costrutti di linguaggi di programmazione)che eliminano a priori il problema della mutua esclusione sulle variabili condivise

Problema dei "dining-philosophers"

- Ci sono 5 filosofi seduti a una tavola rotonda
- Ognuno ha davanti a se un piatto e tra ogni piatto c'e' una bacchetta
- Per mangiare un filosofo usa due bacchette, quella alla sua sinistra e quella alla sua destra che pero' sono condivise con i filosofi vicini
- Di conseguenza due filosofi vicini non possono mangiare contemporaneamente
- I filosofi oltre a mangiare, naturalmente, pensano ma questa attivita' avviene in modo indipendente: l'unico momento in cui si devono sincronizzare e' quando mangiano

Risorse condivise:

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

Meccanismi alternativi di sincronizzazione: monitor

Coda di accesso regolata e disciplinata verso i dati condivisi, magari con priorita' differenziate

---

Sincronizzazione tra processi UNIX: i segnali

Sincronizzazione tra processi

Processi interagenti possono avere bisogno di meccanismi di sincronizzazione.
Ad esempio, abbiamo visto e rivedremo diffusamente il caso di processi pesanti UNIX che vogliano accedere allo stesso file in lettura/scrittura (sincronizzazione di produttore e consumatore)

UNIX: non c'e' condivisione alcuna di spazio di indirizzamento tra processi. Serve un meccanismo di sincronizzazione per modello ad ambiente locale --> segnali.

Segnali

Sono interruzioni software a un processo, che notifica un evento asincrono. Ad esempio segnali:

- generati da terminale (Es. CTRL+C)
- generati da altri processi
- generati dal kernel SO in seguito ad eccezioni HW (violazione dei limiti di memoria, divisione per 0, ...)

- generati da kernel SO in seguito a condizioni SW (time-out, scrittura su pipe chiusa come vedremo in seguito, ...)

Segnali UNIX

Un segnale puo' essere inviato

- dal kernel del SO a un processo
- da un processo utente ad altri processi utente (es. comando kill)

Quando un processo riceve un segnale, puo' comportarsi in tre modi diversi

1. gestire il segnale con una funzione handler definital dal programmatore
2. eseguire un'azione predefinita dal SO (azione di default)
3. ignorare il segnale (nessuna reazione)

Nei primi due casi, processo reagisce in modo asincrono al segnale

1. interruzione dell'esecuzione
2. esecuzione dell'azione associata (handler o default)
3. ritorno alla prossima istruzione del codice del processo interrotto

Per ogni versione di UNIX esistono vari tipi di segnale (in Linux 32 segnali), ognuno identificato da un intero. Ogni segnale e' associato a un particolare evento e prevede una specifica azione di default. E' possibile riferire i segnali con identificatori simbolici (SIGxxxx): SIGKILL, SIGSTOP, SIGUSR1. L'associazione tra nome simbolico e intero corrispondente (che dipende dalla versione di UNIX) e' specificata nell'header file <signal.h>

Gestione dei segnali UNIX

Quando un processo riceve un segnale, puo' gestirlo in 3 modi diversi:

- gestire il segnale con una funzione handler definita dal programmatore
- eseguire un'azione predefinita dal SO (azione di default)
- ignorare il segnale

NB: non tutti i segnalo possono essere gestiti in modalita' scelta esplicitamente dai processi: SIGKILL e SIGSTOP non sono ne intercettabili, ne ignorabili.
--> qualunque processo, alla ricezione di SIGKILL o SIGSTOP esegue sempre l'azione di default.

System call `signal`

Ogni processo puo' gestire esplicitamente un segnale utilizzando la system call signal():

```
typedef void (*handler_t)(int);
handler_t signal(int sig, handler_t handler);
```

- sig e' l'intero (o il nome simbolico) che individua il segnale da gestire
- il parametro handler e' un puntatore a una funzione che indica l'azione da associare al segnale. handler() puo':
  - puntare alla routine di gestione dell'interruzione (handler)
  - valere SIG_IGN (nel caso di segnale ignorato)
  - valere SIG_DFL (nel caso di azioni di default)
- ritorna un puntatore a funzione:
  - al precedente gestore del segnale
  - SIG_ERR(-1), nel caso di errore

Routine di gestione del segnale (handler)

Caratteristiche:

- handler prevede sempre un parametro formale di tipo int che rappresenta il numero del segnale eddettivamente ricevuto
- handler no restituisce alcun risultato

Esempio: gestore del SIGCHLD

SIGCHLD e' il segnale che il kernel del SO invia a un processo padre quando uno dei suoi figli termina.

Tramite l'uso di segnali e' possibile svincolare il padre da un'attesa esplicita della terminazione del figlio, mediante un'apposita funzione handler per la gestione di SIGCHLD:

- la funzione handler verra' attivata in modo asincrono alla ricezione del segnale
- handler chiamera' wait() con cui il padre porta' raccoglier ed eventualmente gestire lo stato di terminazione del figlio

Segnali & fork()

Le associazioni segnali-azioni vengono registrate in User Structure del processo

Siccome:

- `fork()` copia User Structure del padre in quella del figlio
- padre e figlio condividono lo stesso codice, quindi
- il figlio eredita dal padre le informazioni relative alla gestione dei segnai:
  - ignora gli stessi segnali ignorati dal padre
  - gestisce con le stesse funzioni gli stessi segnali gestiti dal padre
  - segnali a default del figlio sono gli stessi del padre

--> ovviamente signal() del figlio successive alla fork() non hanno effetto sulla gestione dei segnali del padre

Segnali & exec()

Sappiamo che

- `exec()` sostituisce codice e dati del processo invocante
- User Structure viene mantenuta, tranne le informazioni legate al codice del processo (ad esempio, le funzioni di gestione dei segnali, che dopo exec() non sono piu' visibili)

quindi

- dopo exec(), un processo:
  - ignora gli stessi segnali ignorati prima di exec()
  - i segnali a default rimangono a default ma
  - i segnali che prima erano festiti, vengono riportati a default

System call `kill()`

I processi possono inviare segnali ad altri processi invocando la system call `kill()`

```
int kill(int pid, int sig);
```

- sig e' l'intero (o il nome simbolico) che individua il segnale da inviare
- il parametro pid specifica il destinatario del segnale;
  - pid>0: l'intero e' il pid dell'unico processo destinatario
  - pid = 0: il segnale e' spedito a tutti i processi appartenenti al gruppo de mittente
  - pid < -1: il segnale e' spedito a tutti i processi con groupld uguale al valore assoluto di pid
  - pid == -1: vari comportamenti possibili (Posix non specifica)

Segnali: altre system call

`sleep()`

```
unsigned int sleep(unsigned int N)
```

- Provoca la sospensione del processo per N secondi (al massimo)
- se il processo riceve un segnale durante il periodo di sospensione, viene risvegliato prematuramente
  - restituisce 0 se la sospensione non e' stata interrotta da segnali
  - se il risveglio e' stato causato da un segnale al tempo x, sleep() restituisce il numero di secondi non utilizzati dell'intervallo di sospensione (N-x)

`alarm()`

```
unsigned int alarm(unsigned int N)
```

- imposta un timer che dopo N secondi inviera' allo stesso processo il segnale SIGALRM
- ritorna:
  - 0, se non vi erano time-out impostati in precedenza
  - il numero di secondi mancante allo scadere del time-out precedente

NB: comportamento di default associato a ricezione di SIGALRM e' la terminazione

`pause()`

```
int pause(void)
```

- sospende il processo fino alla ricezione di un qualunque segnale
- ritorna -1 (errno = EINTR)

Gestione durevole di segnali con handler

Non sempre l'associazione segnale/handler e' durevole:

- alcune implementazioni di UNIX (BSD, SystemV r3 e seguenti), prevedono che l'azione rimanga installata anche dopo la ricezione del segnale
- in alcune realizzazioni (SystemV prime release), dopo l'attivazione, handler ripristina automatiamente l'azione di default. In questi casi, occorre riagganciare il segnale handler.

Modello affidabile dei segnali

Aspetti:

1. se il gestore non rimane installato, e' comunque possibile reinstallare il gestore all'interno dell'handler
2. che cosa succede se arriva il segnale durante l'esecuzione dell'handler?
   - innestamento della routine di gestione?
   - perdita del segnale?
   - accomodamento dei segnali (segnali reliable, BSD 4.2)

`sigaction()`

- La primitiva signal() non e' portabile perche' ha una semantica diversa in diverse versioni di Unix. Per ovviare a questo problema POSIX.1 introduce la sigaction()
- La sigaction() permette di esaminare e/o modificare l'azione associata con un particolare segnale. Si noti che POSIX.1 richiede che un segnale rimanga installato (fino a una modifica esplicita del comportamento)
- Con la sigaction() e' anche possibile specificare il restart automatico delle system call interrotte da un segnale.

signal() vs. sigaction() in sintesi

- signal() con semantica variabile reliable/unreliable
  - unreliable in alcune versioni di UNIX/Linux
  - segnali da reinstallare ogni volta, corsa critica tra inizio handler e reinstallazione handler come prima istruzione dell'handler
  - possibile esecuzione innestata dell'handler se ricezione dello stesso segnale quando siamo ancora nell'handler
- sigaction() invece e' sempre reliable
  -
    - semantica ben definita, identica in ogni versione di UNIX/Linux
      - non c'e' bisogno di reinstallare l'handler
      - non perdiamo segnali: il segnale che ha causato l'attivazione dell'handler e' automaticamente bloccato fino alla fine dell'esecuzione dell'handler stesso

Note sul codice dei gestori segnali

1. Il codice dei gestiori dei segnali non dovrebbe mai contenere chiamate a primitive di I/O, che sono lente e potenzialmente bloccanti.
2. Il modello di comunicazione tra il codice di gestione dei segnali e il codice principale dell'applicazione attraverso l'uso di una variabile statica rappresenta invece una "best practice". Tuttavia, in questi casi e' importante che la variabile utilizzata sia di tipo sig_atomic_t e che essa venga dichiarata con la keyword volatile

Dettagli du sig_atomic_t e volatile

- Infatti, lo standard ISO C definisce sig_atomi_t come un tipo di dato che puo' essere acceduto senza interruzioni. Ovverosia, nessuna lettura da o scrittura su una variabile di tipo sig_atomic_t sara' interrrotta, per esempio dall'occorrenza di un nuovo segnale.
- La keyword volatile invece da istruzione al compilatore di non ottimizzare l'accesso alla variabile corrispondente, che in questo caso produrrebbe un comportamento non corretto del nostro programma
- Se non usassimo sig_atomic_t e volatile nella definizione della variabile, sarebbe molto probabile che alcuni cambiamenti allo stato di una variabile nel codice del gestore del segnale non venissero

notati dal codice dell'applicazione a causa della ricezione di ulteriori segnali o di ottimizzazioni del compilatore.

Comunicazione tra processi UNIX

Comunicazione tra processi UNIX

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

Interazione tra processi puo' avvenire

- mediante la condivisione di file
  - complessita': realizzazione della sincronizzazione tra i processi
- attraverso specifici strumenti di Inter Process Communication (IPC):
  - tra processi sulla stessa macchina
    - pipe (tra processi della stessa gerarchia)
    - fifo (qualunque insieme di processi)
  - tra ptocessi in noodi diversi della stessa rete:
    - socket

pipe

La pipe e' un canale di comunicazione tra processi

- unidirezionale: accessibile mediante due estremi distinti, uno di lettura e uno di scrittura
- (teoricamente) molti-a-molti:
  - piu' processi possono spedire messaggi attraverso la stessa pipe
  - piu' processi possono ricevere messaggi attraverso la stessa pipe
- capacita' limitata
  - in grado di gestire l'accomodamento 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 e' indiretta (senza naming esplicito): modello mailbox

Pipe: unidirezionalita'/bidirezionalita'

Uno stesso processo puo':

- sia depositare messaggi nella pipe (send), mediante il lato di scrittura

- sia prelevare messaggi dalla pipe (receive), mediante il lato di lettura

la pipe puo' anche consentire una comunicazione "bidirezionale" tra Pe Q (ma il programmatore deve rigidamente disciplinarne l'uso per l'utilizzo corretto)

System call pipe

Per creare una pipe: `int pipe(int fd[2]);`

fd e' 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] e' 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

Omogeneita' con i file

Ogni lato di accesso alla pipe e' visto dal processo in modo omogeneo a qualunque altro file (file descriptor). Si puo' accedere alla pipe mediante la system call di lettura/scrittura su file read(), write()

Sincronizzazione automatica della pipe

Il canale (pipe) ha capacita' limitata. Come nel caso di produttore/consumatore e' necessario sincronizzare i processi. Sincronizzazione automatica in UNIX:

- se la pipe e' vuota: un processo che legge si blocca
- se la pipe e' piena: un processo che scrive si blocca

Sincronizzazione automatica: read() e write() sono implementate in modo sospensivo dal SO UNIX

Quali processi possono comunicare mediante pipe?

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

- soltanto processi appartenenti a una stessa gerarchia (cioe', che hanno un antenato in comune) possono scambiarsi messaggi mediante pipe. Ad esemio, possibilita' di comunicazione:
  - tra **processi fratelli** (che ereditano la pipe dal processo padre)
  - tra un processo **padre** e un processo **figlio**
  - tra **nonno** e **nipote**

Chiusura pipe

Ogni processo puo' chiudere un estremo della pipe con la system call `close()`. La comunicazione non e' piu' possibile su di un estremo della pipe quando tutti i processi che avevano visibilita' di quell'estremo hanno compiuto una close().

Se un processo P tenta:

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

System call dup

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

```
int dup(int fd)
```

- fd e' il file descriptor del file da duplicare
- L'effetto di dup() e' copiare l'elemento fd nella 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)

Stdin, stdout, stderr

Per convenzione, per ogni processo vengono aperti automaticamente 3 descrittori di file associati ai primi tre elementi della tabella

- stdin (fd 0) --> tastiera
- stdout (fd 1) --> video
- stderr (fd 2) --> video

dup() & piping

Tramite dup() si puo' realizzare il pipin dei comandi.

Ad esempio: `ls -lR | grep Jun | more`

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

- stdout di ls sia diretto nello stdin di grep
- stdout di grep sia ridiretto nello stdin di more

Pipe: possibili svantaggi

Il meccanismo della pipe ha due svantaggi:

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

Per realizzare la comunicazione persistente tra una coppia di processi non appartenenti alla stessa gerarchia? --> FIFO

fifo

E' una pipe con nome nel fils system:

- Esattamente come le pipe normali, canale unidirezionale del tipo first-in-first-out
- e' rappresentabile da un file nel file system: persistenza, visibilita' potenzialmente globale
- ha un proprietario, un insieme di diritti e una lunghezza
- e' creata dalla system call `mkfifo()`
- e' aperta e acceduta con le stesse system call dei file

Per creare una fifo (pipe con nome):

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

- pathname e' il none 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 puo' essere aperta (come tutti i file) mediante `open()`. Ad esempio, un processo destinatario di messaggi:

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

Per chiudere la fifo, si usa `close(fd)`.

Per eliminare la fifo, si usa `unlink("myfifo")`

Accesso a fifo

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

```
int fd
char msg[100]
fd=open("myfifo", O_RDONLY)
read(fd, msg, 10)
```

Tabella file/pipe/socket descriptor

- Tabella associata ad ogni processo utente e costituita da un elemento (riga) per ogni dile dal processo
- Indice della tabella = descrittore del file (fd)
- Per convenzione, per ogni processo vengono aperti automaticamente 3 descrittori di file associati ai primi tre elementi della tabella stdin (0), stdout (1), stderr (2) associati rispettivamente alla tastiera (0) e al video (1 e 2)
- Ogni entry (riga) della tabella contiene un puntatore o indice della riga della tabella globale dei file aperti relativa ai file