- Segnale in Unix:
  - meccanismo con cui il S.O. avvisa un processo P che si è verificato un evento:
- 1. asincrono cioè scorrelato con ciò che P (in esecuzione o non) sta facendo, ad esempio:
  - a) l'utente ha battuto un carattere di interruzione
  - b) un altro processo Q ha effettuato una chiamata di sistema per inviare il segnale a P – anche se questo non è il meccanismo migliore per la sincronizzazione fra processi
  - c) è suonata una sveglia "caricata" in precedenza da P (granularità: secondi)
- 2. sincrono: causato da ciò che P, in esecuzione, ha cercato di fare (es. eseguire una istruzione illegale, o accedere ad un indirizzo a lui non permesso)

Segnali: tipologie

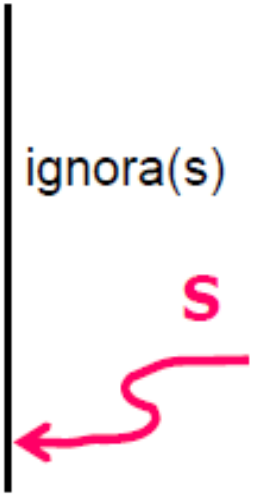
- I diversi tipi di eventi, e quindi i tipi di segnali, in C sono identificati da un valore numerico e una corrispondente costante simbolica, es:
  - 2 SIGINT carattere di interruzione (da tastiera)
  - 4 SIGILL istruzione hw illegale
  - 9 SIGKILL terminazione (comportamento non modificabile)
  - 11 SIGSEGV accesso alla memoria non valido
  - 14 SIGALRM sveglia
  - 15 SIGTERM terminazione
  - SIGUSR1/SIGUSR2 segnali definibili dall'utente
- L'elenco completo sul man (man -s7 signal)
- I valori numerici corrispondenti possono variare da versione a versione quindi meglio usare le opportune costanti
- NB: è possibile per un processo P inviare a un altro processo Q qualunque segnale, anche SIGSEGV, ma solo se P è abilitato a inviare segnali a Q

Segnali: ciclo di vita

- Il segnale è generato/spedito ad un processo quando un particolare evento causa l'occorrenza del segnale:
  - Eccezioni hw, es: divisione per 0
  - Condizioni sw, es: attivazione allarme
- Il S.O. consegna (delivers) il segnale al processo. Nel periodo dalla spedizione alla consegna il segnale è detto in attesa (pendig). Poi che succede?
- Per ogni tipo di segnale c'è un comportamento per default che può essere:
  - ignorare il segnale
  - terminare (variante: terminare salvando un file "core" che può servire per il debugging)
  - sospendersi – poi si può farlo ripartire
- Esempi di comportamento per default:
  - SIGINT terminare
  - SIGSEGV terminare con salvataggio del "core"
  - SIGSTOP sospendersi

Reazione al segnale

- Un processo P può chiedere al sistema operativo di modificare il proprio comportamento in caso di consegna di un segnale di tipo S rispetto al comportamento di default. In particolare può:
  - ignorare il segnale
  - eseguire una funzione ("catturare" il segnale – signal catching), che può servire ad es. Per:
    - eseguire "ultime volontà" (la funzione svolge qualche operazione, ad es. rimuove dei file temporanei, e poi chiama exit)
    - rileggere file di configurazione
  - ripristinare il comportamento di default



- Il comportamento si può modificare solo se S non è SIGKILL (in quel caso si termina sempre) o SIGSTOP che serve a "sospendere" il processo

L'interfaccia affidabile

- Come si chiede di modificare il comportamento? Esiste un modo “non affidabile” e uno “affidabile” (reliable signals)
- L’interfaccia “affidabile” (reliable signals) è leggermente meno semplice da programmare:
- 1) si deve riempire la seguente struttura:

```
struct sigaction {
void (*sa_handler)(int);  /* addr of signal handler, */
                          /* or SIG_IGN to ignore isignal, or SIG_DFL
                             to reset default behavior*/
sigset_t sa_mask;         /* additional signals to block */
int sa_flags;             /* signal options */
….
};
```

  dove `*sa_handler` è un puntatore alla funzione da eseguire alla ricezione del segnale oppure `SIG_IGN` (ignoro segnale) o `SIG_DFL` (ristabilisco comportamento standard)

- 2) Si passa poi come argomento alla funzione sigaction e da quel punto in poi le direttive avranno effetto

L'interfaccia affidabile

- Lo schema tipo di utilizzo è:

```
struct sigaction act,old;          /* nuova/vecchia azione */
act.sa_handler=f;                  /* oppure SIG_IGN o SIG_DFL */
sigemptyset(&act.sa_mask);         /* azzera maschera segnali
act.sa_flags = 0;
sigaction(s,&act,&old)             /* associa act, ricorda old */
```

- per default un tipo di gestione del segnale rimane finché non si chiede di cambiarla
- ogni processo ha una signal mask: insieme di segnali la cui consegna è bloccata mentre si sta eseguendo la routine di gestione di un altro segnale,
- `sigemptyset(&act.sa_mask)` assegna a act.sa_mask l'insieme vuoto, poi passato a sigaction come insieme di segnali da bloccare durante l'esecuzione dell'handler

Invio di segnali

- Come inviare un segnale:
  - con la chiamata di sistema

    kill(pid_destinatario, tipo_segnale)
  - con il comando kill che è realizzato con la chiamata di sistema

Il nome kill è fuorviante, poichè si può inviare qualunque tipo di segnale, non solo SIGKILL, e (come già visto) non tutti i segnali uccidono

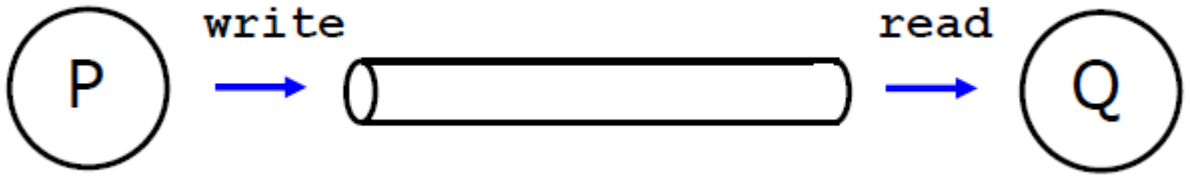
Chiamate di sistema interrotte

- Che succede se arriva un segnale «da catturare» durante una chiamata di sistema «lenta», es.:
  - lettura da tastiera
  - wait, waitpid
- In tal caso la chiamata viene interrotta: si esce con valore -1 ed errno=EINTR
- Con sigaction si può però chiedere che la chiamata di sistema venga invece fatta ripartire, per i dettagli vedere ad es. *Stevens & Rago, Advanced Programming in the Unix Environment*
- In questo corso non vedremo come far ripartire la chiamata di sistema, ma dovrete sapere come interpretare, ed eventualemente risolvere, un errore di EINTR

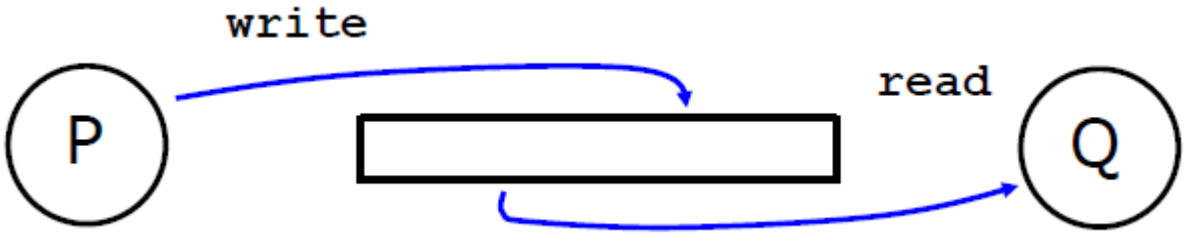
Esercizi

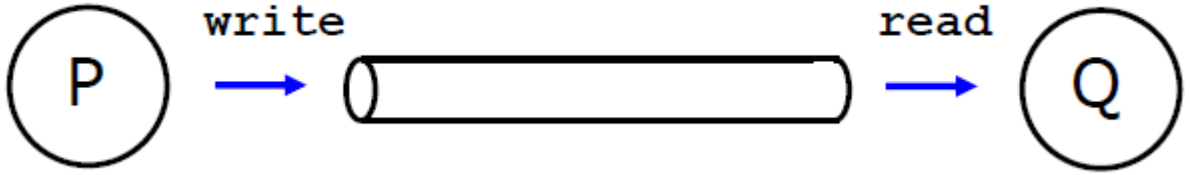
- Scaricare dal sito del corso il codice di prova su segnali e pipe (appunti4)
- ESERCIZIO 4.1: modificare il programma sig.c affichè stampi il PID del processo
- ESERCIZIO 4.2: inviare da terminale segnali con il comando kill al processo che esegue "sig.c": inviare il segnale di interruzione (SIGINT), quello di terminazione (SIGTERM) e quello di accesso non valido alla memoria (SIGSEGV).
- ESERCIZIO 4.3: modificare il programma in modo da ignorare il segnale di interruzione.
- ESERCIZIO 4.4: verificare, integrando gli esempi precedenti che utilizzano fork, exec e sigaction, se le disposizioni "ignorare il segnale" ed "eseguire una funzione" vengono:
  - "ereditate" da un processo figlio, qualora richieste dal processo padre prima della fork;
  - mantenute da un processo che effettua una system call exec.

- Le pipes (pipe = tubo, tubazione) sono un meccanismo di comunicazione tra processi che si appoggia alle chiamate di sistema per accedere ai files:

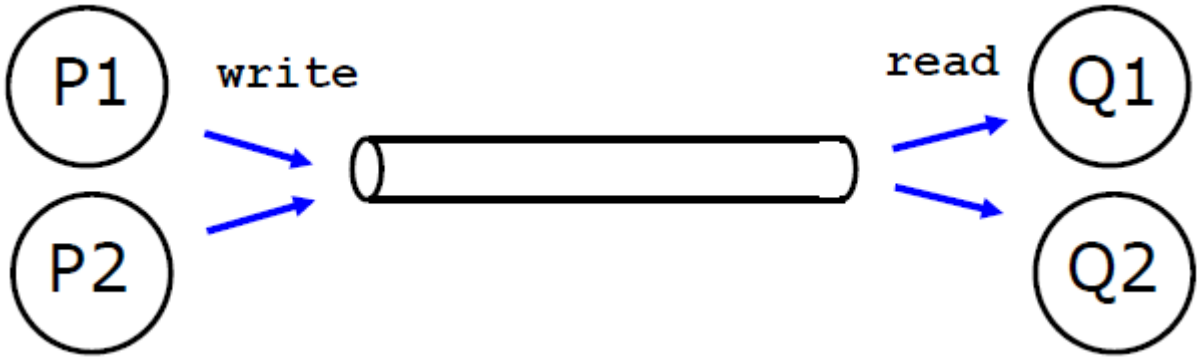


- In realtà i dati non si “muovono” nel “tubo”:

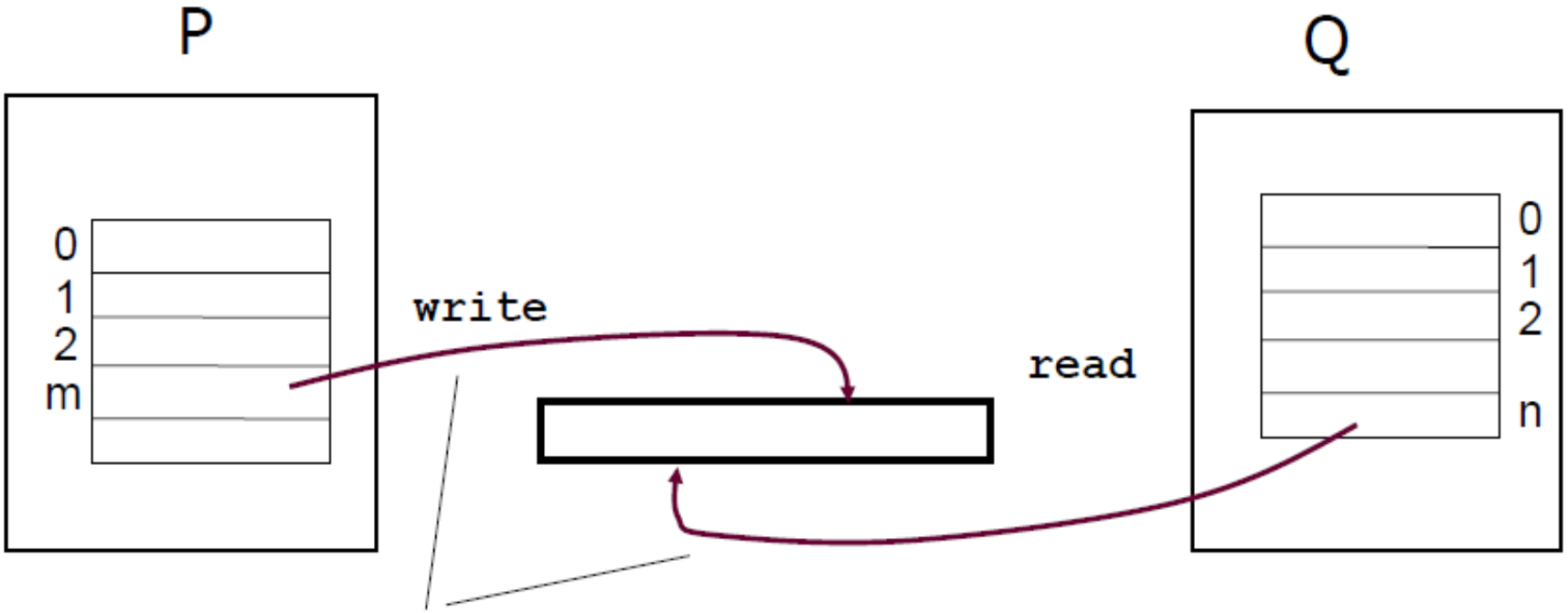




- Una read da una pipe può sospendere il processo chiamante (Q), se tutti i dati scritti nella pipe sono già stati letti (pipe “vuota”)
- Una write su una pipe può sospendere il processo chiamante (P), se i dati scritti e non ancora letti occupano tutto lo spazio allocato (pipe “piena”)
- Possono esserci più “produttori” e “consumatori”:



- NB: le strutture a cui ci si appoggia sono le stesse dei files: la tabella dei file aperti
- Proprio per questo si possono usare read e write

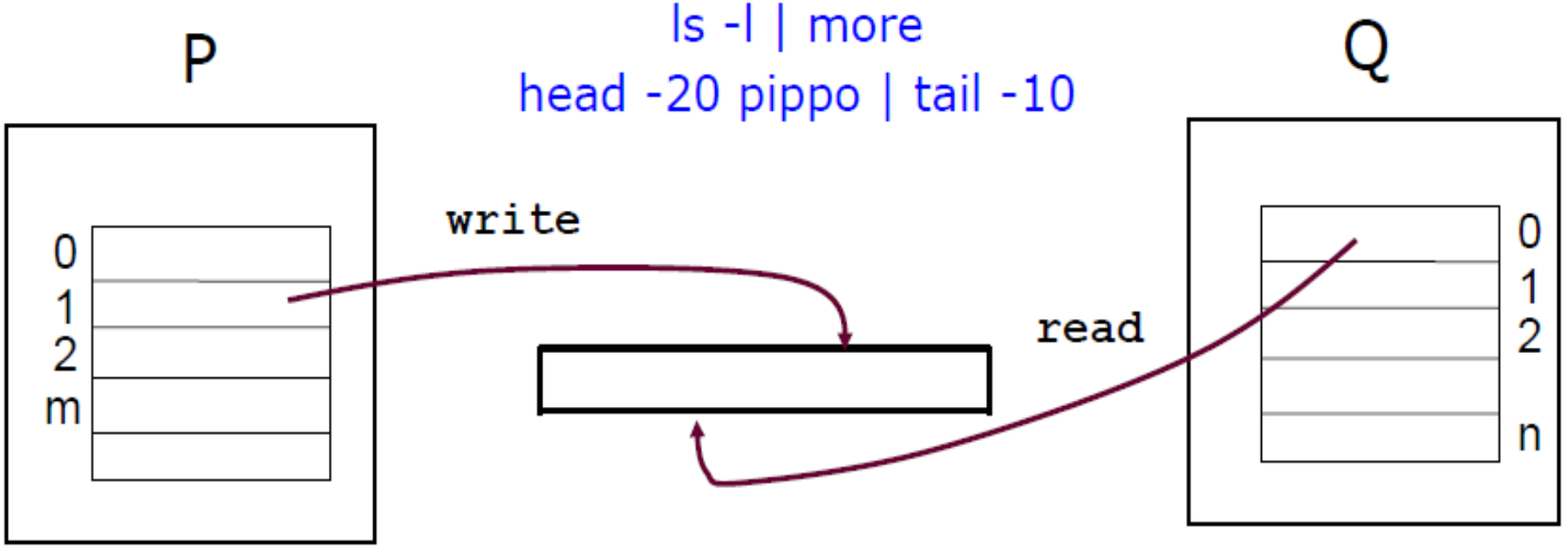


NB: su questa pagina le frecce non indicano il flusso di dati,
ma il punto in cui si scrive/legge

- Ma non solo... ci si può ridirigere lo standard input/output, per avere quello che negli interpreti di comandi si ottiene con:

comando1 | comando2

- lo standard output di comando1 diventa lo standard input di comando2, es.:

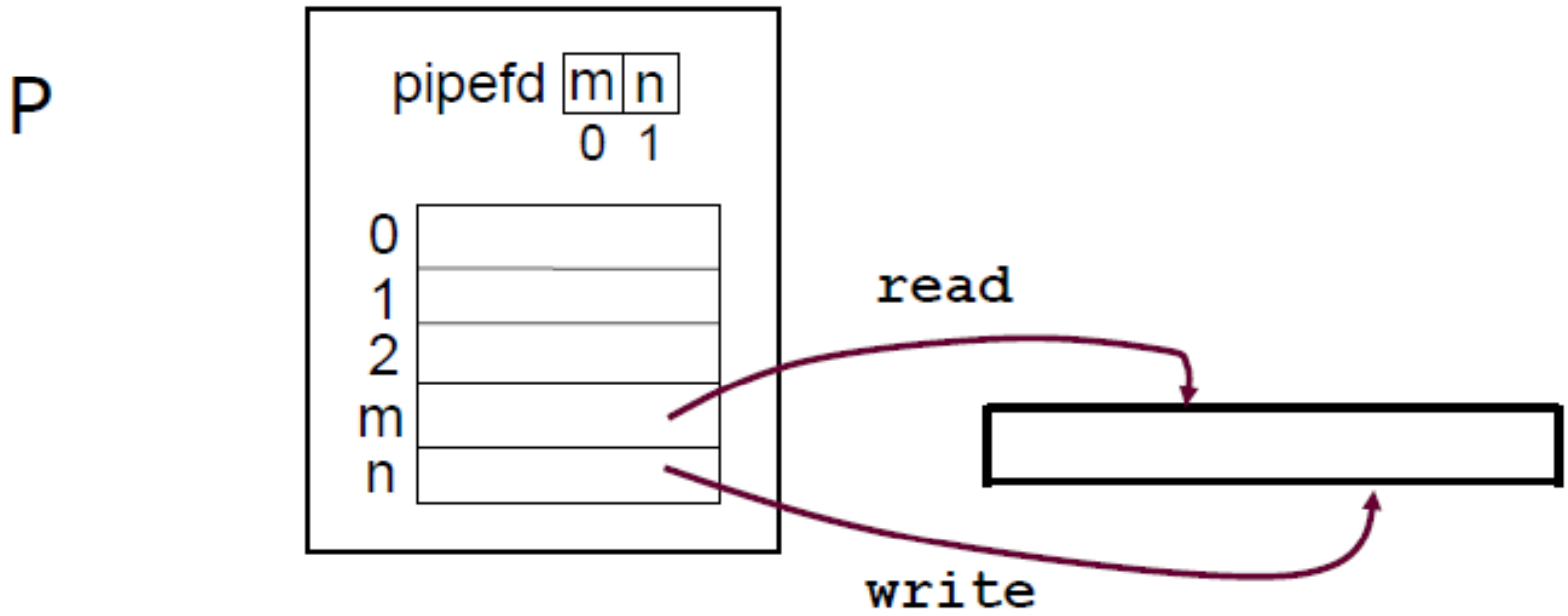


- Ma come si implementa ?

- Una pipe si apre con la chiamata di sistema pipe:

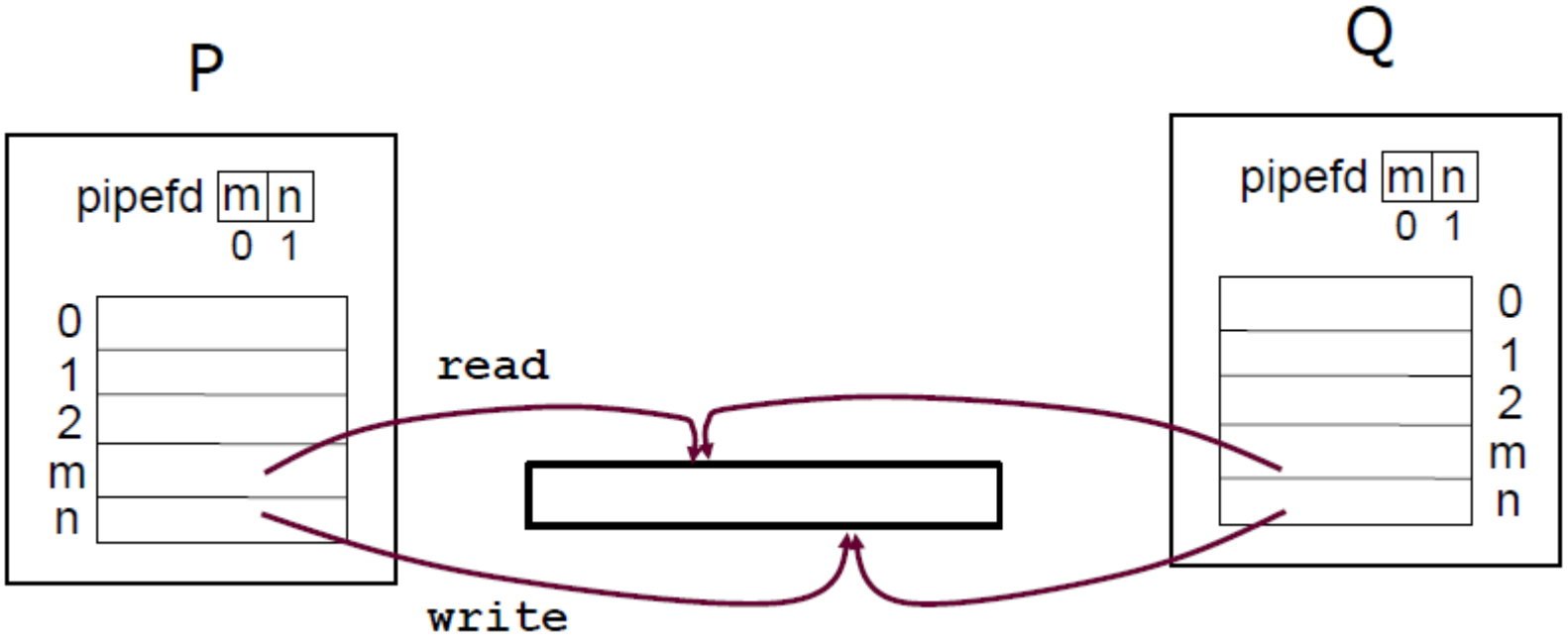
```
int pipefd[2];
pipe(pipefd);
```

- che alloca le strutture necessarie per gestire la pipe (es. buffer per parcheggiare i dati) e mette nell'array due "descrittori di file" (indici nella tabella dei file aperti) da usare per leggere/scrivere sulla pipe

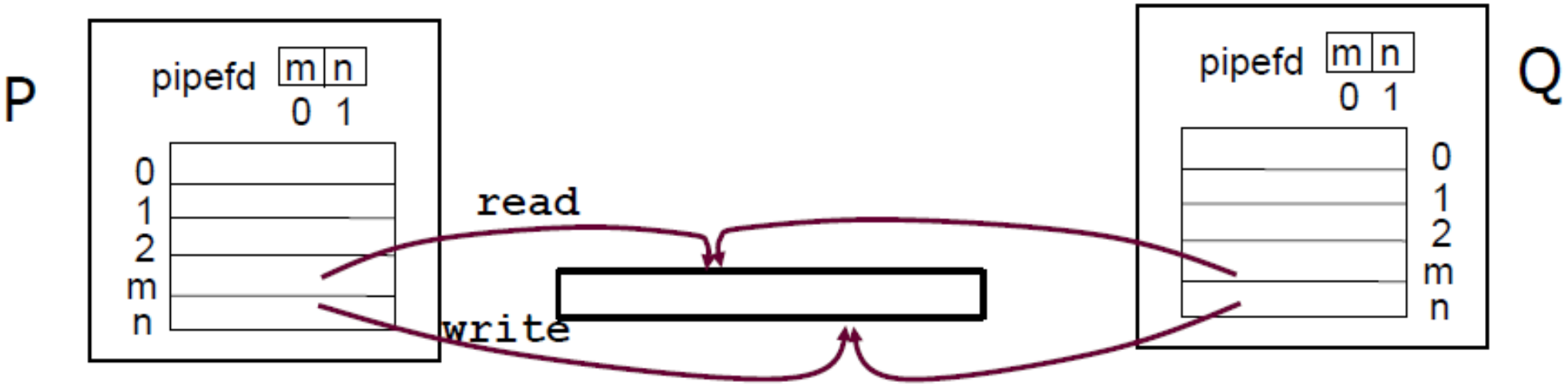


- Che se ne fa un processo di tutto questo? Poco, se è uno solo

- Però dopo una fork() abbiamo:

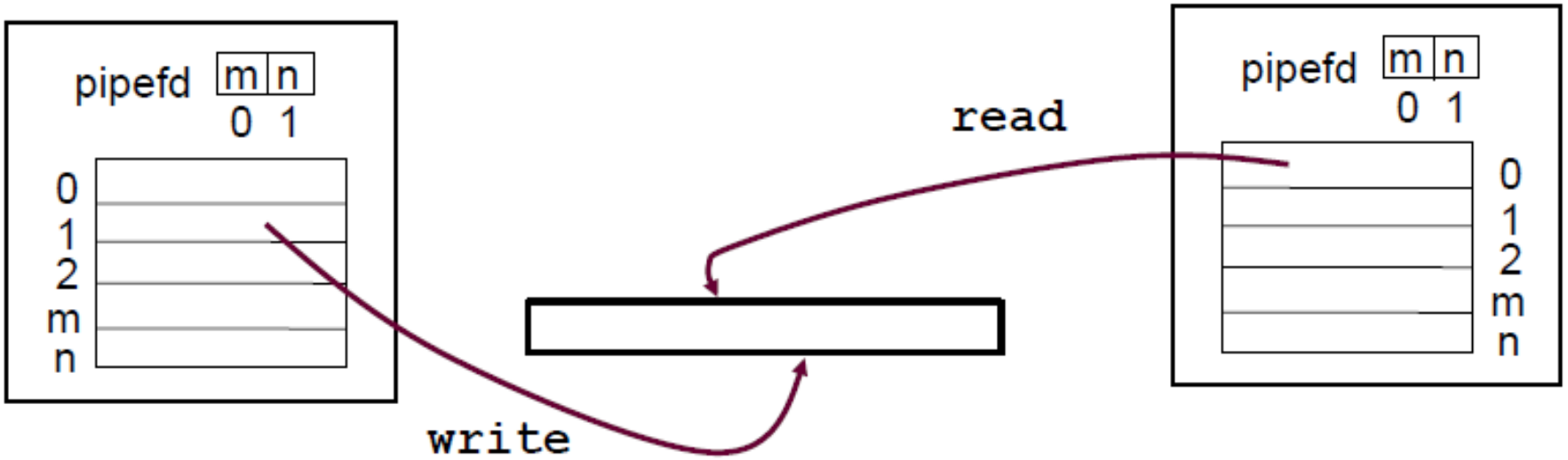


- e con opportune dup e close si ottiene quanto desiderato




```
dup2(pipefd[1],1); // copia la riga n nella 1
close(pipefd[0]);  // a P non serve leggere
close(pipefd[1]);  // non serve più
```

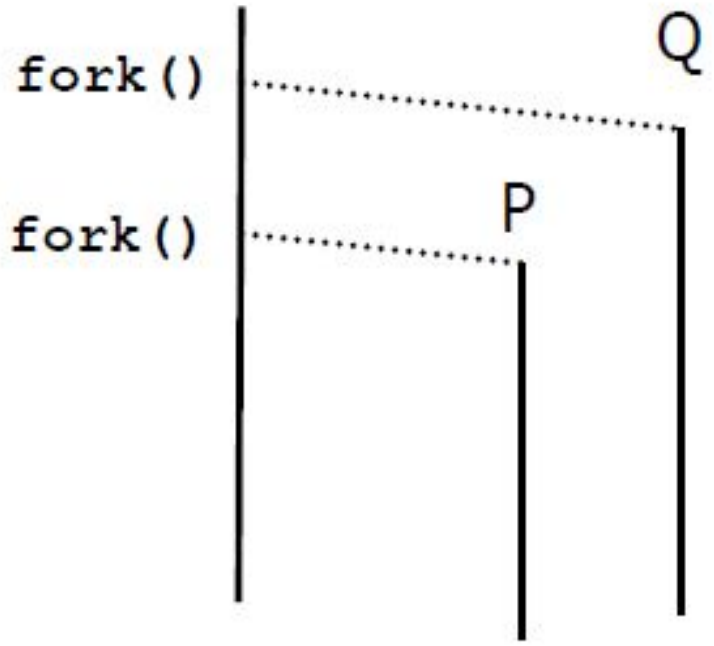
```
dup2(pipefd[0],0);
close(pipefd[0]);
close(pipefd[1]);
```



- In realtà nell'esempio che vediamo in laboratorio, che riproduce:

  comando1 | comando2

- dopo la chiamata di pipe vengono generati, come succede nella shell, due processi figli che ereditano copia della tabella dei file aperti
- Al processo padre (analogo dell'interprete shell) la pipe non serve, e quindi deve solo chiudere i descrittori dopo le fork



- Perché è importante chiudere i descrittori?
  - 1. (sempre, quando un file aperto non serve) per non tenere inutilmente occupate righe della tabella, di dimensione prefissata
  - 2. (per le pipe) Come fa il sistema operativo a dire “end of file” a un processo che legge da una pipe?
    - NB1: avere “end of file” è essenziale, es. per “while not EOF”
    - NB2: il processo può non sapere che sta leggendo da una pipe, ad es. se legge dal proprio standard input che è stato ridiretto
  - Risposta: se la pipe è “vuota” e non ci sono processi che hanno la pipe aperta in scrittura

- ESERCIZIO 4.5: in pipes.c verificare che cosa accade se nel padre la close(pipefd[1]) viene tolta oppure messa dopo le wait, perchè?
- ESERCIZIO 4.6: Realizzare analogo programma che riproduce, invece del comando "ls -l | wc -l", il comando "head -20 pipes.c | tail -10" che produce in output le righe da 11 a 20 del file "pipes.c"
- ESERCIZIO 4.7: realizzare una pipe a due produttori e un consumatore. Il programma prende come argomenti le stringhe <file1> e <file2>, poi:

  un produttore invoca cat <file1>, un secondo produttore cat <file2> e il consumatore fa il more

  - Poichè non usiamo funzioni di sincronizzazione, l'output finale risulterà un mix confuso di <file1> e <file2>