

Modulo 5: Programmazione di sistema

- - -

Parte E: Segnali e pipe

Laboratorio di Sistemi Operativi I
Anno Accademico 2006- 2007

Francesco Pedullà
(Tecnologie Informatiche)

Massimo Verola
(Informatica)

Copyright © 2005-2006 Francesco Pedullà, Massimo Verola

Copyright © 2001-2005 Renzo Davoli (Università di Bologna), Alberto Montresor (Università di Bologna)

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation;

with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license can be found at: <http://www.gnu.org/licenses/fdl.html#TOC1>

Sommario

- ◆ segnali
- ◆ pipe
- ◆ named pipe

Introduzione ai segnali - I

- ♦ **I segnali sono interrupt software a livello di processo**
 - ♦ Permetto la gestione di eventi asincroni che interrompono il normale funzionamento di un processo
- ♦ **Segnali – breve storia**
 - ♦ Versione "non affidabile" introdotti dalle prime versioni di Unix
 - ♦ I segnali potevano essere persi
 - ♦ Unix 4.3BSD e SVR3 introducono segnali affidabili
 - ♦ Si evita la possibilità che un segnale vada perso
 - ♦ POSIX.1 standardizza la gestione dei segnali

Introduzione ai segnali - II

- ♦ **Caratteristiche dei segnali**
 - ♦ Ogni segnale ha un identificatore
 - ♦ Identificatori di segnali iniziano con i tre caratteri SIG
 - ♦ Es. `SIGABRT` è il segnale di abort
 - ♦ Numero segnali: 15- 40, a seconda della versione di UNIX
 - ♦ POSIX: 18
 - ♦ Linux: 38
 - ♦ I nomi simbolici corrispondono ad un intero positivo
 - ♦ Definizioni di costanti in `bits/signum.h`
 - ♦ Il numero 0 è utilizzato per un caso particolare
- ♦ **I segnali sono eventi asincroni**
 - ♦ La gestione avviene tramite *signal handler*

Condizioni che possono generare segnali - I

- ◆ **Pressione di tasti speciali sul terminale**
 - ◆ Es: Premere il tasto `Ctrl-C` genera il segnale `SIGINT`
- ◆ **Eccezioni hardware**
 - ◆ Divisione per 0 (`SIGFPE`)
 - ◆ Riferimento non valido a memoria (`SIGSEGV`)
 - ◆ L'interrupt viene generato dall'hardware, e catturato dal kernel; questi invia il segnale al processo in esecuzione
- ◆ **System call `kill`**
 - ◆ Permette di spedire un segnale ad un altro processo
 - ◆ Limitazione: uid del processo che esegue `kill` deve essere lo stesso del processo a cui si spedisce il segnale, oppure 0 (root)

Condizioni che possono generare segnali - II

- ◆ **Comando `kill`**
 - ◆ Interfaccia shell alla system call `kill`
- ◆ **Condizioni software**
 - ◆ Eventi asincroni generati dal software del sistema operativo, non dall'hardware della macchina
 - ◆ Esempi:
 - ◆ terminazione di un child (`SIGCHLD`)
 - ◆ generazione di un alarm (`SIGALRM`)

Azioni associate ai segnali - I

- ◆ Ignorare il segnale

- ◆ Alcuni segnali che non possono essere ignorati: **SIGKILL** e **SIGSTOP**
 - ◆ Motivo: permettere al superutente di terminare processi
 - ◆ Segnali hardware: comportamento non definito in POSIX se ignorati

- ◆ Esecuzione dell'azione di default

- ◆ Per molti segnali "critici", l'azione di default consiste nel terminare il processo
- ◆ Può essere generato un file di core, eccetto quando:
 - ◆ bit set-user-id e set-group-id settati e uid/gid diversi da owner/group;
 - ◆ mancanza di permessi in scrittura per la directory;
 - ◆ core file troppo grande

Azioni associate ai segnali - II

- ◆ **Catturare ("catch") il segnale:**
 - ◆ Il kernel informa il processo chiamando una funzione specificata dal processo stesso (*signal handler*)
 - ◆ Il signal handler gestisce il problema nel modo più opportuno
- ◆ **Esempio:**
 - ◆ nel caso del segnale **SIGCHLD** (terminazione di un child)
 - possibile azione: eseguire `waitpid`
 - ◆ nel caso del segnale **SIGTERM** (terminazione standard)
 - possibili azioni: rimuovere file temporanei, salvare file

Alcuni dei segnali più importanti - I

- ♦ **SIGABRT (Terminazione, core)**
 - ♦ Generato da `system call abort()`; terminazione anormale
- ♦ **SIGALRM (Terminazione)**
 - ♦ Generato da un timer settato con la `system call alarm` o la funzione `setitimer`
- ♦ **SIGBUS (Non POSIX; terminazione, core)**
 - ♦ Indica un hardware fault (definito dal s.o.)
- ♦ **SIGCHLD (Default: ignore)**
 - ♦ Quando un processo termina, `SIGCHLD` viene spedito al processo parent
 - ♦ Il processo parent deve definire un signal handler che chiami `wait` o `waitpid`
- ♦ **SIGFPE (Terminazione, core)**
 - ♦ Eccezione aritmetica, come divisioni per 0
- ♦ **SIGHUP (Terminazione)**
 - ♦ Inviato ad un processo se il terminale viene disconnesso

Alcuni dei segnali più importanti - II

- ♦ **SIGILL (Terminazione, core)**
 - ♦ Generato quando un processo ha eseguito un'azione illegale
- ♦ **SIGINT (Terminazione)**
 - ♦ Generato quando un processo riceve un carattere di interruzione (Ctrl-C) dal terminale
- ♦ **SIGIO (Non POSIX; default: terminazione, ignore)**
 - ♦ Evento I/O asincrono
- ♦ **SIGKILL (Terminazione)**
 - ♦ Maniera sicura per uccidere un processo
- ♦ **SIGPIPE (Terminazione)**
 - ♦ Scrittura su pipe/ socket in cui il lettore ha terminato/ chiuso
- ♦ **SIGSEGV (Terminazione, core)**
 - ♦ Generato quando un processo esegue un riferimento di memoria non valido

Alcuni dei segnali più importanti - III

- ♦ **SIGSYS (Terminazione, core)**
 - ♦ Invocazione non valida di system call
 - ♦ Esempio: parametro errato
- ♦ **SIGTERM (Terminazione)**
 - ♦ Segnale di terminazione normalmente generato dal comando kill
- ♦ **SIGURG (Non POSIX; ignora)**
 - ♦ Segnala il processo che una condizione urgente è avvenuta (dati out-of-bound ricevuti da una connessione di rete)
- ♦ **SIGUSR1, SIGUSR2 (Terminazione)**
 - ♦ Segnali non definiti utilizzabili a livello utente
- ♦ **SIGSTP (Default: stop process)**
 - ♦ Generato quando un processo riceve un carattere di suspend (Ctrl-Z) dal terminale

System call signal - I

- ♦ `void (*signal(int signo, void (*func)(int)))(int);`
- ♦ **Descrizione:**
 - ♦ **signo**: l'identificatore del segnale che si vuole catturare
 - ♦ **func**: l'azione che vogliamo che sia eseguita
 - ♦ **SIG_IGN**: ignora il segnale (non applicabile a **SIGKILL** e **SIGSTOP**)
 - ♦ **SIG_DFL**: azione di default
 - ♦ l'indirizzo del signal handler: quando si vuole catturare il segnale (non applicabile a **SIGKILL** e **SIGSTOP**)
 - ♦ valore di ritorno:
 - ♦ il valore del precedente signal handler se ok
 - ♦ **SIG_ERR** in caso di errore

System call signal - II

- **Definizione alternativa:**

```
typedef void sighandler_t(int);
```

```
sighandler_t *signal(int, sighandler_t*);
```

- **Definizione delle costanti (tipica) in `signal.h`:**

- Queste costanti possono essere utilizzate come "puntatori a funzioni che prendono un intero e non ritornano nulla"
- I valori devono essere tali che non possano essere assegnati a signal handler

```
#define SIG_ERR (void (*)( ))-1
```

```
#define SIG_DFL (void (*)( ))0
```

```
#define SIG_IGN (void (*)( ))1
```

Generazione dei segnali - I

- ◆ **System call:** `int kill(pid_t pid, int signo);`
 - ◆ La system call `kill` spedisce un segnale ad un processo oppure ad un gruppo di processi
 - ◆ Argomento `pid`:
 - ◆ `pid > 0` spedito al processo identificato da `pid`
 - ◆ `pid == 0` spedito a tutti i processi appartenenti allo stesso gruppo del processo che invoca `kill`
 - ◆ `pid < -1` spedito al gruppo di processi identificati da `-pid`
 - ◆ `pid == -1` non definito
 - ◆ Argomento `signo`:
 - ◆ Numero di segnale spedito

Generazione dei segnali - II

- ♦ **System call:** `int kill(pid_t pid, int signo);`
 - ♦ Permessi:
 - ♦ Il superutente può spedire segnali a chiunque
 - ♦ Altrimenti, il real uid o l'effective uid della sorgente deve essere uguale al real uid o l'effective uid della destinazione
 - ♦ POSIX.1 definisce il segnale 0 come il *null signal*
 - ♦ Se il segnale spedito è null, `kill` esegue i normali meccanismi di controllo errore senza spedire segnali
 - ♦ Esempio: verifica dell'esistenza di un processo; spedizione del null signal al processo (nota: i process id vengono riciclati!)
- ♦ **System call:** `int raise(int signo);`
 - ♦ Spedisce il segnale al processo chiamante

Generazione dei segnali - III

- ◆ **System call:** `unsigned int alarm(unsigned int sec);`
 - ◆ Questa system call permette di creare un allarme che verrà generato dopo il numero specificato di secondi
 - ◆ Allo scadere del tempo, il segnale **SIGALRM** viene generato
 - ◆ Attenzione: *il sistema non è real-time*
 - ◆ Garantisce che la pausa sarà almeno di **sec** secondi
 - ◆ Il meccanismo di scheduling può ritardare l'esecuzione di un processo
 - ◆ Esiste un unico allarme per processo
 - ◆ Se un allarme è già settato, il numero di secondi rimasti prima dello scadere viene ritornato da `alarm`
 - ◆ Se **sec** è uguale a zero, l'allarme preesistente viene generato

Generazione dei segnali - IV

- ◆ **System call:** `unsigned int alarm(unsigned int sec);`
 - ◆ L'azione di default per `SIGALRM` è di terminare il processo
 - ◆ Ma normalmente viene definito un signal handler per il segnale
- ◆ **System call:**
 - ◆ `int getitimer(int which, struct itimerval *value);`
 - ◆ `int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);`
 - ◆ Permettono un controllo più completo
- ◆ **System call:** `int pause();`
 - ◆ Questa system call sospende il processo fino a quando un segnale non viene catturato (ritorna `-1` e setta `errno` a `EINTR`)

Esercizio

- ♦ **Scrivere sigusr.c:**
 - ♦ Cattura i segnali definiti dall'utente e stampa un messaggio di errore
- ♦ **Esempio di output:**

```
$ a.out &
```

```
[1]      235
```

```
$ kill -USR1 235      # spedisce segnale SIGUSR1 a 235
```

```
received SIGUSR1    # catturato
```

```
$ kill 235          # spedisce segnale SIGTERM
```

```
[1] + Terminated  a.out &
```

Soluzione sigusr.c - I

```
#include <signal.h>
```

```
#include <stdio.h>
```

```
#include <string.h>
```

```
void handler(int sig_number)
```

```
{
```

```
    char msg[]="Ricevuto segnale SIGUSR1!\n";
```

```
    write(1,msg,strlen(msg));
```

```
}
```

Soluzione sigusr.c - II

```
int main()  
{  
    void (*sh_addr)(int);  
    sh_addr=signal(SIGUSR1,handler);  
    while (1)  
        pause();  
}
```

Funzione sleep

- ♦ `unsigned int sleep(unsigned int seconds);`
 - ♦ questa system call causa la sospensione del processo fino a quando:
 - ♦ l'ammontare di tempo specificato trascorre
 - ♦ return value: 0
 - ♦ un segnale viene catturato e il signal handler effettua un return
 - ♦ return value: tempo rimasto prima del completamento della sleep
 - ♦ nota:
 - ♦ la `sleep` può concludersi dopo il tempo richiesto
 - ♦ la `sleep` può essere implementata utilizzando `alarm()`, ma spesso questo non accade per evitare conflitti

Funzione abort

- ♦ `void abort();`
 - ♦ questa funzione spedisce il segnale `SIGABRT` al processo
 - ♦ comportamento in caso di:
 - ♦ `SIG_DFL`: terminazione del processo
 - ♦ `SIG_IGN`: non ammesso
 - ♦ signal handler: il segnale viene catturato
 - ♦ nel caso che il segnale venga catturato, il signal handler:
 - ♦ può eseguire `return`
 - ♦ può invocare `exit` o `_exit`
 - ♦ in entrambi i casi, il processo viene terminato
 - ♦ motivazioni per il catching: cleanup

Startup

- **Quando un programma esegue una system call `fork`:**
 - I signal catcher settati nel parent vengono copiati nel figlio
- **Quando un programma viene eseguito tramite `exec`**
 - Se il signal catcher per un certo segnale è default o ignore, viene lasciato inalterato nel child
 - Se il signal catcher è settato ad una particolare funzione, viene cambiato a default nel child
 - Motivazione: la funzione può non esistere nel figlio
- **Casi particolari**
 - Quando un processo viene eseguito in background
 - Segnali `SIGINT` e `SIGQUIT` vengono settati a *ignore*
 - In che momento/ da chi questa operazione viene effettuata?

Funzioni *reentrant* - I

- ◆ **Quando un segnale viene catturato**
 - ◆ la normale sequenza di istruzioni viene interrotta
 - ◆ vengono eseguite le istruzioni del signal handler
 - ◆ quando il signal handler ritorna (invece di chiamare `exit`) la normale sequenza di istruzioni viene ripresa
- ◆ **Problemi:**
 - ◆ Cosa succede se un segnale viene catturato durante l'esecuzione di una `malloc` (che gestisce lo heap), e il signal handler invoca una chiamata a `malloc`?
 - ◆ In generale, può succedere di tutto...
 - ◆ Normalmente, ciò che accade è un segmentation fault...

Funzioni reentrant - II

- ♦ **POSIX.1 garantisce che un certo numero di funzioni siano reentrant**
 - ♦ `_exit`, `access`, `alarm`, `chdir`, `chmod`, `chown`, `close`, `creat`, `dup`, `dup2`, `execle`, `execve`, `exit`, `fcntl`, `fork`, `fstat`, `get*id`, `kill`, `link`, `lseek`, `mkdir`, `mkfifo`, `open`, `pathconf`, `pause`, `pipe`, `read`, `rename`, `rmdir`, `set*id`, `sig*`, `sleep`, `stat`, `sysconf`, `time`, `times`, `umask`, `uname`, `unlink`, `utime`, `wait`, `waitpid`, `write`
- ♦ **Se una funzione manca...**
 - ♦ perché utilizza strutture dati statiche
 - ♦ perché chiama `malloc` e `free`
 - ♦ perché fa parte della libreria standard di I/O

Funzioni reentrant - III

- ♦ **In ogni caso:**
 - ♦ Le funzioni reentrant listate in precedenza possono modificare la variabile `errno`
 - ♦ Un signal handler che chiama una di quelle funzioni dovrebbe salvare il valore di `errno` prima della funzione e ripristinarlo dopo
 - ♦ Evitare l'utilizzo di `printf` (o altre funzioni non rientranti) nel signal handler

Standard POSIX - I

- Nelle prime versioni di UNIX

- I segnali non erano affidabili
 - Potevano andare persi (un segnale viene lanciato senza che un processo ne sia al corrente)
 - Problema derivante in parte dal fatto che, una volta catturato, il signal catcher doveva essere ristabilito

```
signal(SIGINT, sig_int);  
void sig_int() {  
    signal(SIGINT, sig_int);  
    /* process the signal */  
}
```

- Race condition tra l'arrivo del segnale e la `signal` in `sig_int()`

Standard POSIX - II

- ♦ **Cosa succede se un processo riceve un segnale durante una system call?**
 - ♦ normalmente, l'eventuale azione associata viene eseguita solo dopo la terminazione della system call
 - ♦ in alcune system call "lente", le prime versioni di Unix potevano interrompere la system call, la quale ritornava `-1` come errore e la variabile `errno` veniva settata a `ENINTR`

Standard POSIX - III

- ♦ **Motivazioni:**
 - ♦ in assenza di interruzioni da segnali:
 - ♦ una lettura da terminale resta bloccata per lunghi periodi di tempo
 - ♦ un segnale di interruzione non verrebbe mai consegnato
 - ♦ poiché il processo ha catturato un segnale, c'è una buona probabilità che sia successo qualcosa di significativo

Standard POSIX - IV

- ◆ **System call "lente":**
 - ◆ operazioni `read` su file che possono bloccare il chiamante per un tempo indeterminato (terminali, pipe, connessioni di rete)
 - ◆ operazione `write` su file che possono bloccare il chiamante per un tempo indeterminato prima di accettare dati
 - ◆ `pause`, `wait`, `waitpid`
 - ◆ certe operazioni `ioctl`
 - ◆ alcune system call per la comunicazione tra processi
- ◆ **Problemi:**
 - ◆ bisognerebbe gestire esplicitamente l'errore dato dalle interruzioni

Standard POSIX - V

- ◆ **Esempio di gestione:**

```
while ( (n= read(fd, buff, BUFSIZE)) < 0 )  
    { if (errno != EINTR) break; }
```

- ◆ **Restart automatico di alcune system call:**

- ◆ Alcune system call possono ripartire in modo automatico:
 - ◆ per evitare la necessita' di gestire l'errore dovuto ad una interruzione (come nell'esempio sopra)
 - ◆ perché in alcuni casi non è dato sapere se il file su cui si opera può bloccarsi indefinitamente
- ◆ System call con restart: `ioctl`, `read`, `write`
 - ◆ solo quando operano su fd che possono bloccarsi indefinitamente
- ◆ System call senza restart: `wait`, `waitpid`
 - ◆ sempre

Standard POSIX - VI

- ♦ **POSIX e S.O. moderni:**
 - ♦ Capacità di interrompere le system call: standard
 - ♦ I signal handler rimangono installati: standard
 - ♦ Restart automatico delle system call: non specificato
 - ♦ In realtà, in molti S.O. moderni è possibile specificare se si desidera il restart automatico oppure no
- ♦ **POSIX specifica un meccanismo per segnali affidabili:**
 - ♦ E' possibile gestire ogni singolo dettaglio del meccanismo dei segnali
 - ♦ quali bloccare
 - ♦ quali gestire
 - ♦ come evitare di perderli, etc.

Segnali affidabili - I

♦ Alcune definizioni:

- ♦ Diciamo che un segnale è *generato* per un processo quando accade l'evento associato al segnale
 - ♦ Esempio: riferimento memoria non valido \Rightarrow **SIGSEGV**
 - ♦ Quando il segnale viene generato, viene settato un flag nel process control block del processo
- ♦ Diciamo che un segnale è *consegnato* ad un processo quando l'azione associata al segnale viene intrapresa
- ♦ Diciamo che un segnale è *pendente* nell'intervallo di tempo che intercorre tra la generazione del segnale e la consegna

Segnali affidabili - II

♦ Bloccare i segnali

- ♦ Un processo ha l'opzione di bloccare la consegna di un segnale per cui l'azione di default non è configurata a *ignore*
- ♦ Se un segnale bloccato viene generato per un processo, il segnale rimane pending fino a quando:
 - ♦ il processo sblocca il segnale
 - ♦ il processo cambia l'azione associata al segnale al valore *ignore*
- ♦ E' possibile ottenere la lista dei segnali pending tramite la funzione `sigpending`
- ♦ **Cosa succede se un segnale bloccato viene generato più volte prima che il processo sblocchi il segnale?**
 - ♦ POSIX non specifica se i segnali debbano essere accodati oppure se vengano consegnati una volta sola

Segnali affidabili - III

- ♦ **Cosa succede se segnali diversi sono pronti per essere consegnati ad un processo?**
 - ♦ POSIX non specifica l'ordine in cui devono essere consegnati
 - ♦ POSIX suggerisce che segnali importanti (come `SIGSEGV`) siano consegnati prima di altri
- ♦ **Maschera dei segnali:**
 - ♦ Ogni processo ha una maschera di segnali che specifica quali segnali sono attualmente bloccati
 - ♦ E' possibile pensare a questa maschera come ad un valore numerico con un bit per ognuno dei possibili segnali
 - ♦ E' possibile esaminare la propria maschera utilizzando la system call `sigprocmask`

Gestione segnali - I

- ◆ **System call:** `int sigpending(sigset_t *set);`
- ◆ **Descrizione:**
 - ◆ Ritorna l'insieme di segnali che sono attualmente pending per il processo corrente
- ◆ **Esempio:**

```
void pr_mask() {
    sigset_t  sigset;
    int  errno_save = errno;
    if (sigpending(&sigset) < 0)
        perror("sigpending error");
    if (sigismember(&sigset, SIGINT))  printf("SIGINT ");
    if (sigismember(&sigset, SIGQUIT)) printf("SIGQUIT ");
    /* remaining signals can go here */
    printf("\n");
    errno = errno_save;
}
```

Gestione segnali - II

- ◆ **System call**

```
int sigaction(int signo, struct sigaction
              *newact, struct sigaction *oldact);
```

- ◆ Permette di esaminare e/ o modificare l'azione associata ad un segnale
- ◆ Nei sistemi moderni, `signal` è implementata utilizzando `sigaction`
- ◆ Argomenti:
 - ◆ `signo` segnale considerato
 - ◆ `newact` se diverso da `NULL`, struttura dati contenente informazioni sulla nuova azione
 - ◆ `oldact` se diverso da `NULL`, struttura dati contenente informazioni sulla vecchia azione

Gestione segnali - III

- **Struttura sigaction:**

```
struct sigaction {  
    void (*sa_handler)();    /* signal handler */  
    sigset_t sa_mask;        /* addit.block mask */  
    int sa_flags;            /* options */  
}
```

- **Descrizione:**

- `sa_handler` è il puntatore all'azione per il segnale (un signal handler, `SIG_IGN` o `SIG_DFL`)
- `sa_mask` è un insieme addizionale di segnali da bloccare quando un segnale viene catturato da un signal handler
- `sa_flags` descrive flag addizionali per vincolare il comportamento del sistema

Gestione segnali - IV

- ◆ **Utilizzo di `sa_mask`**
 - ◆ All'inizio dell'esecuzione di un signal handler:
 - ◆ il valore corrente della `procmask` viene salvato
 - ◆ alla `procmask` vengono aggiunti
 - ◆ i segnali specificati in `sa_mask`
 - ◆ il segnale specificato da `signo`
 - ◆ Al termine dell'esecuzione di un signal handler:
 - ◆ la `procmask` viene ripristinata al valore salvato
- ◆ **Alcuni valori per `sa_flag` (non standard POSIX):**
 - ◆ `SA_RESTART` forza automatic restart per system call interrotte da questo segnale
 - ◆ `SA_INTERRUPT` elimina automatic restart per system call interrotte da questo segnale

Gestione segnali - V

- ◆ **System call:**

```
int sigsuspend(sigset_t *sigmask);
```

- ◆ La procmask viene posta uguale al valore puntato da **sigmask**
- ◆ Il processo è sospeso fino a quando:
 - ◆ un segnale viene catturato
 - ◆ un segnale causa la terminazione di un processo
- ◆ Ritorna sempre **-1** con **errno** uguale a **EINTR**

Esercizio: Sincronizzazione tramite segnali - I

♦ Descrizione

- ♦ realizzare un meccanismo basato su segnali per risolvere il problema produttore / consumatore
 - ♦ processo padre: produttore
 - ♦ produce numeri interi consecutivi
 - ♦ stampa il proprio pid e il numero prodotto
 - ♦ processo figlio: consumatore,
 - ♦ consuma i numeri prodotti dal padre
 - ♦ stampa il proprio pid e il numero consumato
- ♦ il processo padre e figlio utilizzano un buffer di dimensione 1, realizzato con un file condiviso

Esercizio: Sincronizzazione tramite segnali - II

♦ Descrizione

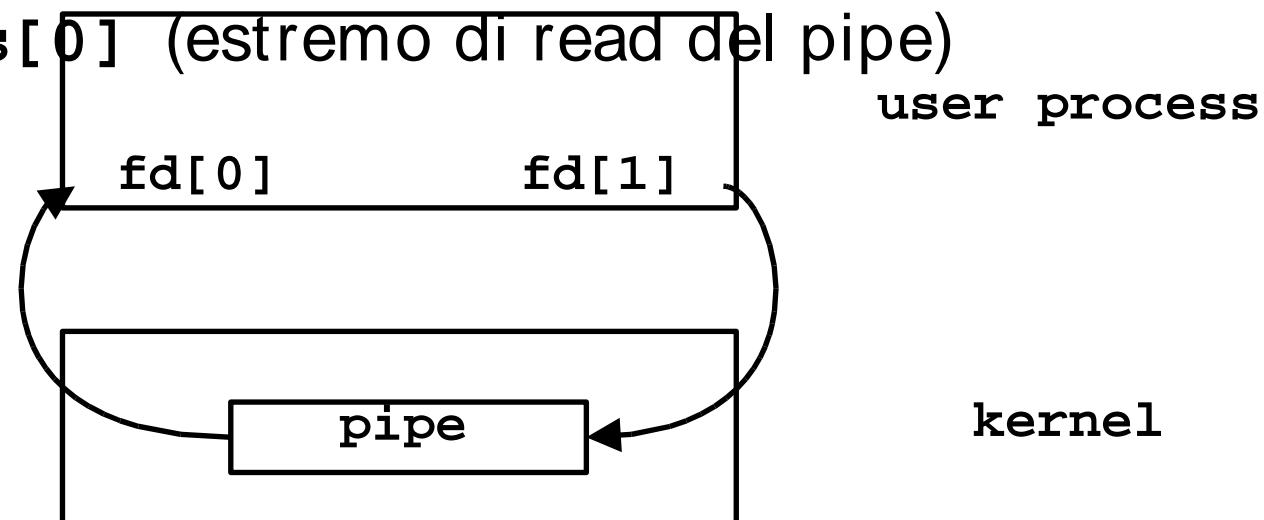
- ♦ processo padre:
 - ♦ apre il file condiviso in scrittura
 - ♦ utilizza `lseek` per scrivere sempre all'inizio
 - ♦ scrive i 4 byte che rappresentano un int
- ♦ processo figlio:
 - ♦ apre il file condiviso in lettura
 - ♦ utilizza `lseek` per leggere sempre all'inizio
 - ♦ legge i 4 byte che rappresentano un int

Definizione e caratteristiche di un pipe

- ◆ **Cos'è un pipe?**
 - È un canale di comunicazione che unisce due processi
- ◆ **Caratteristiche:**
 - La più vecchia e la più usata forma di *interprocess communication* utilizzata in Unix
 - Limitazioni
 - Sono half- duplex (comunicazione in un solo senso)
 - Utilizzabili solo tra processi con un "antennato" in comune
 - Come superare queste limitazioni?
 - Gli *stream pipe* sono full- duplex
 - *FIFO (named pipe)* possono essere utilizzati tra più processi
 - *named stream pipe* = stream pipe + FIFO

System call *pipe* e file descriptor

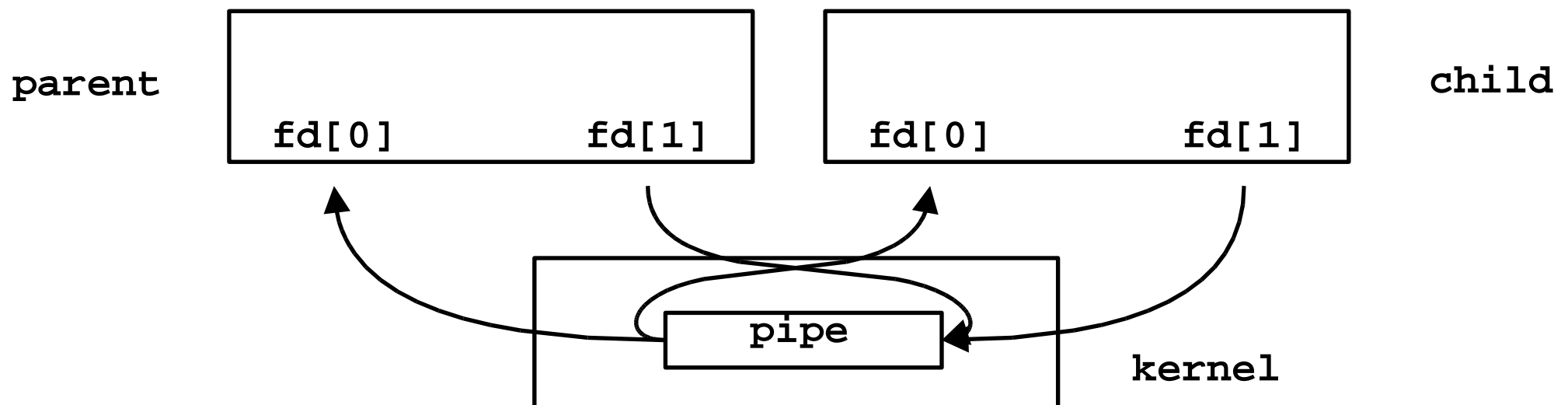
- **System call:** `int pipe(int fildes[2]);`
 - Ritorna due descrittori di file attraverso l'argomento `fildes`
 - `fildes[0]` è aperto in lettura
 - `fildes[1]` è aperto in scrittura
 - L'output di `fildes[1]` (estremo di write del pipe) è l'input di `fildes[0]` (estremo di read del pipe)



Utilizzo di pipe - I

◆ Come utilizzare i pipe?

- I pipe in un singolo processo sono completamente inutili
- Normalmente:
 - il processo che chiama pipe chiama `fork`
 - i descrittori vengono duplicati e creano un canale di comunicazione, **allocato nel kernel**, tra parent e child o viceversa



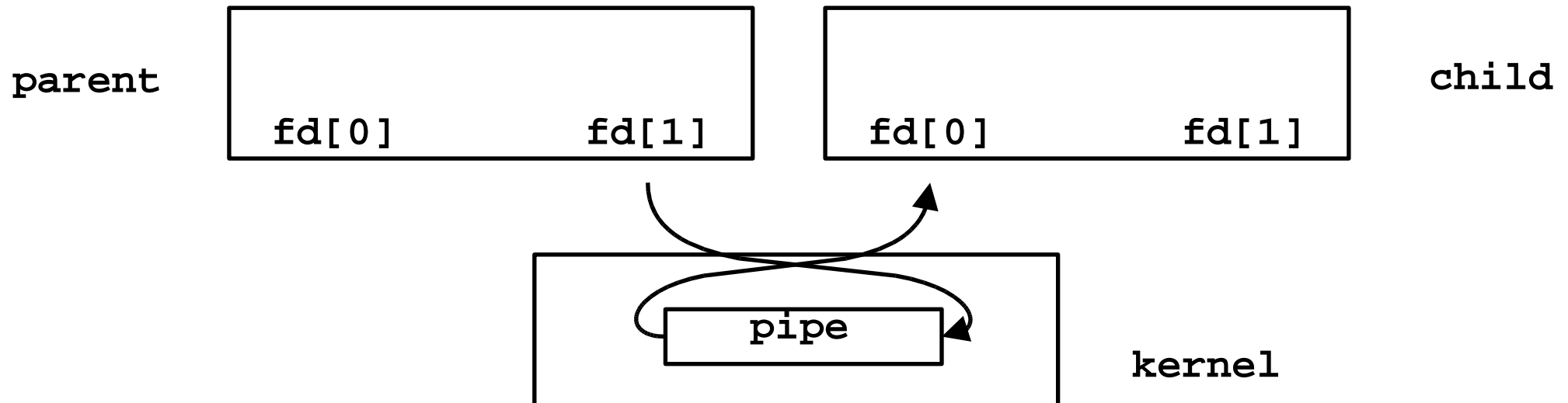
Utilizzo di pipe - II

◆ Come utilizzare i pipe?

- Cosa succede dopo la `fork` dipende dalla direzione dei dati
- I canali non utilizzati vanno chiusi

◆ Esempio: parent → child

- Il parent chiude l'estremo di read (`close(fd[0]);`)
- Il child chiude l'estremo di write (`close(fd[1]);`)



Utilizzo di pipe - III

♦ Come utilizzare i pipe?

- Una volta creati, è possibile utilizzare le normali chiamate `read/write` sugli estremi

♦ La chiamata `read`

- se l'estremo di `write` è aperto
 - restituisce i dati disponibili, ritornando il numero di byte
 - successive chiamate si bloccano fino a quando nuovi dati non saranno disponibili
- se l'estremo di `write` è stato chiuso
 - restituisce i dati disponibili, ritornando il numero di byte
 - successive chiamate ritornano 0, per indicare la fine del file

Utilizzo di pipe - IV

♦ La chiamata `write`

- se l'estremo di read è aperto
 - i dati in scrittura vengono bufferizzati fino a quando non saranno letti dall'altro processo
- se l'estremo di read è stato chiuso
 - viene generato un segnale **SIGPIPE**
 - ignorato/ catturato: `write` ritorna `-1` e `errno= EPIPE`
 - azione di default: terminazione

♦ **Esercizio:**

- Due processi: parent e child
- Il processo parent comunica al figlio una stringa, e questi provvede a stamparla

Utilizzo di pipe - V

♦ Chiamata `fstat`

- Se utilizziamo `fstat` su un descrittore aperto su un pipe, il tipo del file sarà descritto come fifo (macro `S_ISFIFO`)

♦ Atomicità

- Quando si scrive su un pipe, la costante `PIPE_BUF` specifica la dimensione del buffer del pipe
- Chiamate `write` di dimensione inferiore a `PIPE_BUF` vengono eseguite in modo atomico
- Chiamate `write` di dimensione superiore a `PIPE_BUF` possono essere eseguite in modo non atomico
 - La presenza di scrittori multipli può causare interleaving tra chiamate `write` distinte

Esercizio

- ◆ **Consideriamo un programma che vuole mostrare il proprio output una pagina alla volta**
 - Aggiungere al programma la funzionalità di `more`, `less`
 - Scrivere uno script che metta il programma in pipe con `more`
 - Scrivere un programma che crei un pipe con `more`

Procedura

- si crea un pipe
- si chiama `fork`
- si chiudono nel padre e nel figlio gli opportuni descrittori
- il figlio usa `dup2` per duplicare l'estremo di input del pipe sullo standard input
 - attenzione; bisogna verificare che il descrittore non abbia già il valore scelto
- Nota: cerca in una variabile di ambiente `PAGER`

popen - I

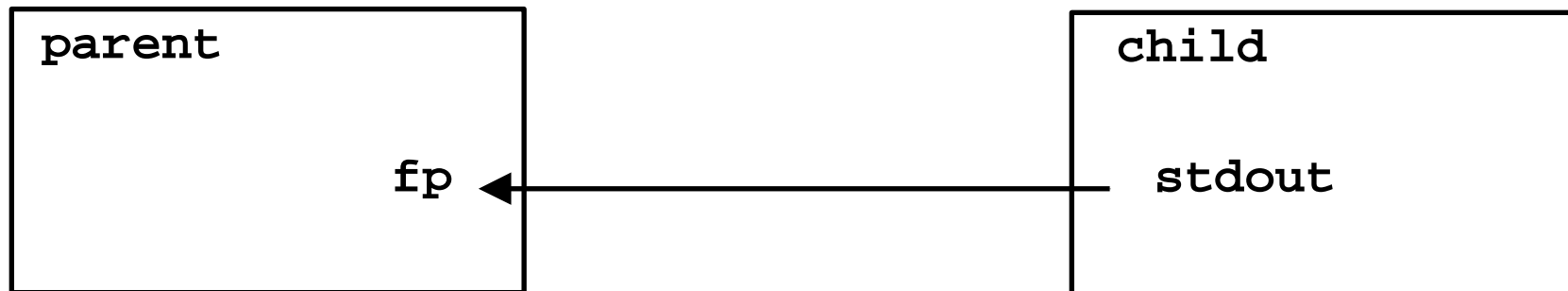
- ◆ `FILE *popen(char *cmdstring, char *type);`
- ◆ **Descrizione di popen:**
 - crea un pipe
 - crea mediante `fork` un processo child
 - chiude gli estremi non utilizzati del pipe
 - esegue mediante `exec` una shell (`sh -c cmdstring`) per eseguire il comando `cmdstring`
 - ritorna uno standard I/O file pointer:
 - se si specifica `type="r"` il file pointer (usato in lettura) e' collegato allo standard output del processo child `cmdstring`
 - se si specifica `type="w"` il file pointer (usato in scrittura) e' collegato allo standard output del processo child `cmdstring`

popen - ll

- ♦ `type = "w"`



- ♦ `type = "r"`



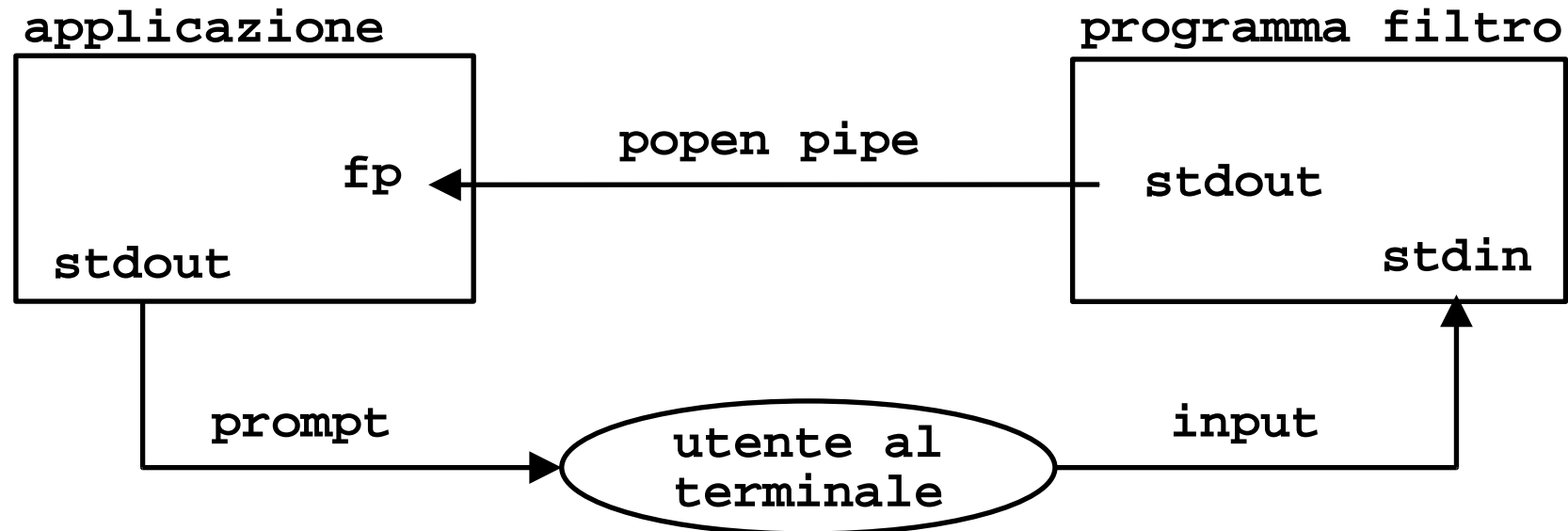
Nota: `cmdstring` è eseguita tramite `"/bin/sh -c"`

pclose - esempio

- ◆ `int pclose(FILE *fp);`
- ◆ **Descrizione di pclose**
 - chiude lo standard I/O file pointer ritornato da `popen`
 - attende mediante `wait` la terminazione del comando
 - ritorna il termination status della shell invocata per eseguire il comando

popen - esempio di utilizzo

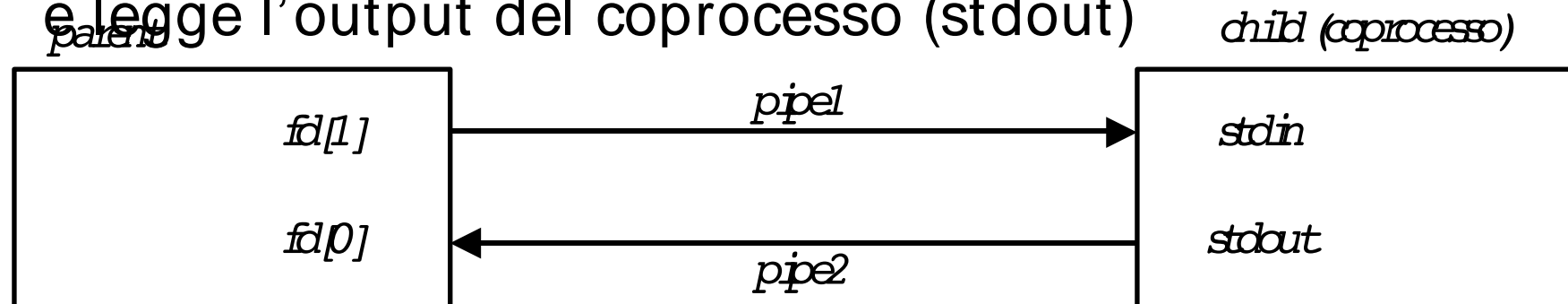
- Si consideri un'applicazione che scrive un prompt su standard output e legge una linea da standard input
- Mediante popen e' possibile inserire programma ("filtro") tra l'input e l'applicazione, cosi' da trasformare l'input prima che venga letto dall'applicazione
- La trasformazione potrebbe essere l'implementazione della *pathname expansion* o del meccanismo di *history*



Coprocessi

• Cos'è un coprocesso?

- Un filtro UNIX è un processo che legge da stdin e scrive su stdout
- Normalmente i filtri UNIX sono connessi linearmente mediante la pipeline della shell
- Un filtro si definisce coprocesso quando è collegato ad un altro processo, il quale genera l'input del coprocesso (stdin) e legge l'output del coprocesso (stdout)



Pipe e named pipe

◆ Pipe "normali"

- possono essere utilizzate solo da processi che hanno un "antenato" in comune, poiché questo è l'unico modo per ereditare descrittori di file

◆ Named pipe o FIFO

- permettono a processi non collegati di comunicare
- sebbene siano dei canali di comunicazione **allocati nel kernel** come le pipe normali, utilizzano il file system per "dare un nome" ai pipe (**i dati NON vengono scritti su disco!**)
- un FIFO è un tipo di file speciale, infatti utilizzando le chiamate **stat**, **lstat** sul pathname che corrisponde ad un FIFO, la macro **S_ISFIFO** restituirà **true**
- la procedura per creare un FIFO è simile alla procedura per creare un file

FIFO - I

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

- crea un FIFO dal `pathname` specificato
- la specifica dell'argomento `mode` è identica a quella di `open`, `creat` (`mode` codifica i permessi di accesso al file mediante un numero ottale, ad esempio `0644 = rw-r--r--`)
- ♦ **Come funziona un FIFO?**
 - una volta creato un FIFO, le normali chiamate `open`, `read`, `write`, `close`, possono essere utilizzate per leggere il FIFO
 - il FIFO può essere rimosso utilizzando `unlink`
 - le regole per i diritti di accesso si applicano come se fosse un file normale

Leggi: `man 4 fifo` per ulteriori informazioni e descrizione del comportamento specifico delle varie system call

FIFO - II

- ◆ **Chiamata open:**
 - File aperto senza flag `O_NONBLOCK`
 - Se il FIFO è aperto in sola lettura, la chiamata **si blocca** fino a quando un altro processo non apre il FIFO in scrittura
 - Se il FIFO è aperto in sola scrittura, la chiamata **si blocca** fino a quando un altro processo non apre il FIFO in lettura
 - File aperto con flag `O_NONBLOCK`
 - Se il FIFO è aperto in sola lettura, la chiamata **ritorna immediatamente**
 - Se il FIFO è aperto in sola scrittura, e nessun altro processo lo ha aperto in lettura, la chiamata **ritorna un**

messaggio di errore

FIFO - III

♦ Chiamata `write`

- se nessun processo ha aperto il file in lettura viene generato un segnale `SIGPIPE`:
 - ignorato/ catturato: `write` ritorna `-1` e `errno= EPIPE`
 - azione di default: terminazione

♦ Atomicità

- Quando si scrive su un pipe, la costante `PIPE_BUF` (in genere pari a 4096, vedi `/usr/include/linux/limits.h`) specifica la dimensione del buffer del pipe
- Chiamate `write` di dimensione inferiore a `PIPE_BUF` vengono eseguite in modo atomico
- Chiamate `write` di dimensione superiore a `PIPE_BUF` possono essere eseguite in modo non atomico
- La presenza di più scrittori può causare interleaving tra chiamate `write` distinte

FIFO - IV

Tabella riassuntiva sull'effetto del flag `O_NONBLOCK` su pipe e

FIFO

CONDIZIONE	COMPORAMENTO DI DEFAULT	COMPORAMENTO CON <code>O_NONBLOCK</code>
open di FIFO read-only senza che altri processi abbiano il FIFO aperto in scrittura	attesa finche' un processo apre FIFO in scrittura	ritorno immediato senza errore
open di FIFO write-only senza che altri processi abbiano il FIFO aperto in lettura	attesa finche' un processo apre FIFO in lettura	ritorno immediato con errore, errore pari a <code>ENXIO</code>
read da pipe o FIFO che non contiene dati	attesa finche' non vi siano dati in FIFO, o finche' nessun processo abbia piu' FIFO aperto in scrittura	ritorno immediato, valore di ritorno pari a 0
write in pipe o FIFO pieni	attesa finche' non vi sia spazio per scrivere,	ritorno immediato, valore di ritorno pari a 0

FIFO - V

- **Utilizzo dei FIFO**

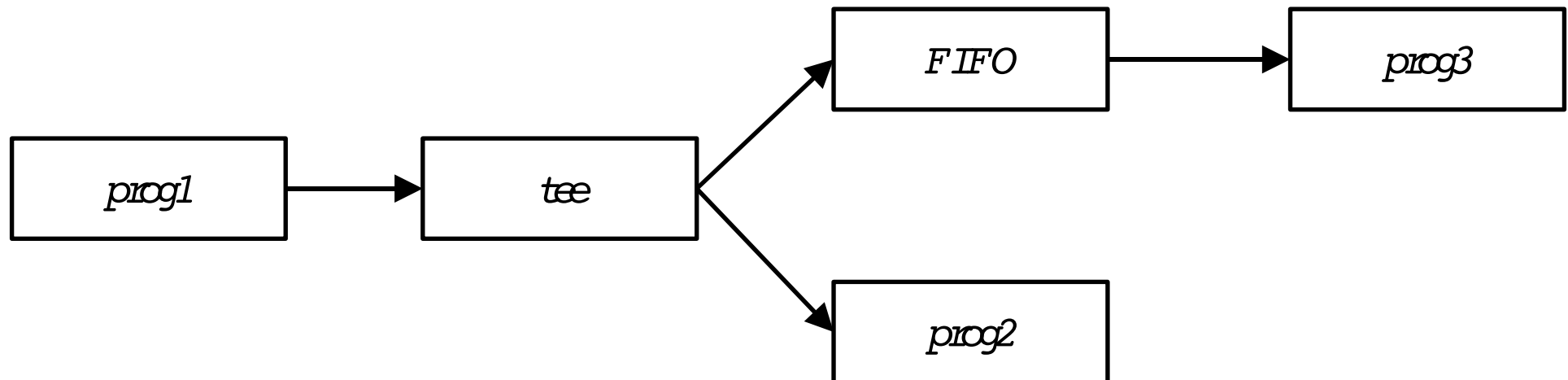
- Utilizzati dai comandi shell per passare dati da una shell pipeline ad un'altra, senza passare creare file intermedi

- **Esempio:**

```
mkfifo fifo1
```

```
prog3 < fifo1 &
```

```
prog1 | tee fifo1 | prog2
```



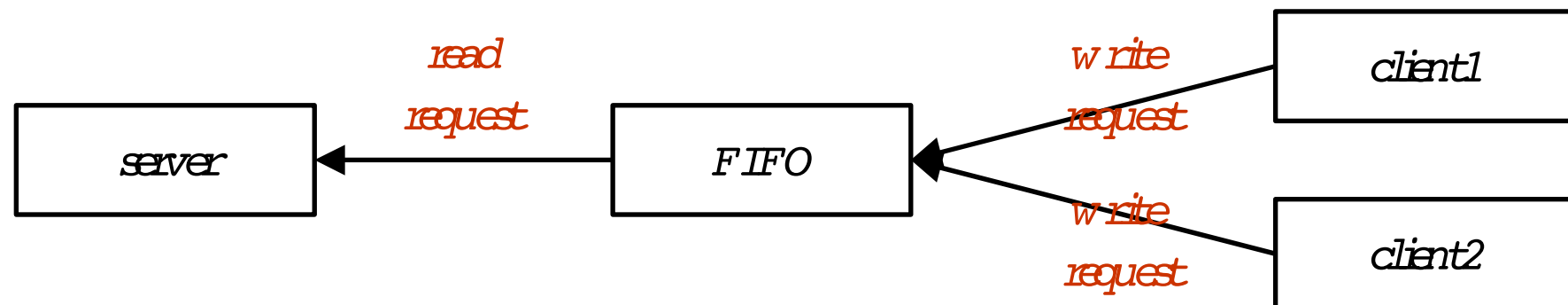
FIFO - VI

♦ Utilizzo dei FIFO

- Utilizzati nelle applicazioni client- server per comunicare

♦ Esempio:

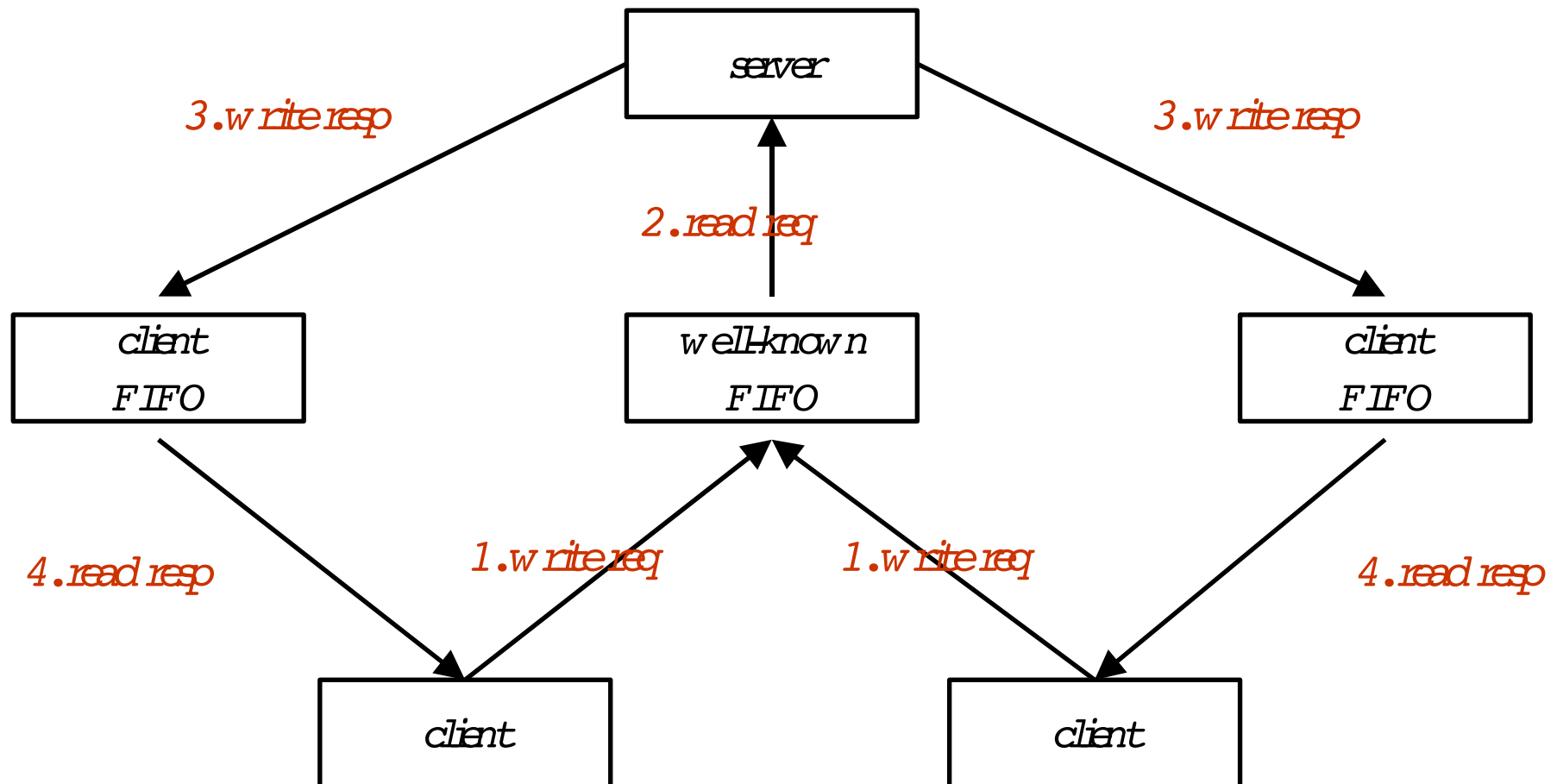
- Comunicazioni client → server
 - il server crea un FIFO
 - il pathname di questo FIFO deve essere "well-known" (ovvero, noto a tutti i client)
 - i client scrivono le proprie richieste sul FIFO
 - il server legge le richieste dal FIFO



FIFO - VII

- ◆ **Problema: come rispondere ai client?**
 - Non è possibile utilizzare il "well-known" FIFO
 - I client non saprebbero quando leggere le proprie risposte
 - Soluzione:
 - i client spediscono il proprio process id al server
 - ogni client crea un proprio FIFO (*client FIFO*) per la risposta, il cui nome contiene il process ID (in modo tale che il server puo' ricostruirlo), e lo apre in lettura
 - il server apre in scrittura il *client FIFO*
 - il server scrive sul *client FIFO* la risposta alla richiesta del client
 - Suggestimenti:
 - Il server dovrebbe catturare SIGPIPE, in quanto il client potrebbe terminare o chiudere il FIFO prima di leggere la risposta (altrimenti il SIGPIPE provocherebbe la terminazione del server)
 - Il server dovrebbe aprire in lettura e scrittura il "well-known" FIFO, altrimenti, quando l'ultimo client termina, il server leggerà EOF, invece di rimanere bloccato sulla read, in attesa che un nuovo client si connetta sulla "well-known FIFO"

FIFO - VIII



Esercizio

♦ Descrizione

- riscrivere l'esercizio produttore- consumatore utilizzando una named pipe
- utilizzare la pipe come buffer
- il produttore scrive interi sulla pipe
- il consumatore li stampa
- utilizzare più produttori