

System call: gestione dei segnali



Roberto De Virgilio

Sistemi operativi - 20 Dicembre 2017

Segnali - Introduzione

- I processi talvolta devono gestire eventi inaspettati o impredicibili
 - errori run-time (es. divisione per 0)
 - mancanza di corrente elettrica
 - terminazione di un processo figlio
 - richiesta di terminazione da parte di un utente (<Ctrl-c>)
 - richiesta di sospensione da parte di un utente (<Ctrl-z>)

Segnali - Introduzione

- Per la gestione di eventi **asincroni** di questo tipo Unix offre il meccanismo dei **segnali**.
 - I segnali sono *interrupt software* a livello di processo
 - Permettono 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

Segnali - Introduzione

- **Caratteristiche dei segnali**
 - Ogni segnale ha un **identificatore**
 - Identifieri di segnali sono *nomi simbolici* (**SIGxxx**) che 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 **signal.h**
 - Il numero **0** è utilizzato per un caso particolare
- **I segnali sono eventi asincroni**
 - La gestione avviene tramite **signal handler**
 - "*Se e quando avviene un segnale, esegui questa funzione*"

Condizioni che possono generare segnali

■ Cause di un segnale

- I segnali possono essere legati a diverse **cause**
 - Kernel
 - Altri processi
 - Interrupt esplicativi dell'utente

Condizioni che possono generare segnali

- **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 deve essere 0 (root)

Segnali - Gestione

- Il kernel verifica la presenza di segnali pendenti per un processo quando questo
 - passa da kernel a user mode (es. ritorno da sys call)
 - abbandona lo stato sleep o vi entra
- Debolezze dei segnali
 - un segnale può essere gestito con un certo **ritardo** (problema per applicazioni real-time)
 - non si tiene conto del *numero di segnali* dello **stesso tipo** pendenti (se vari **SIGINT** arrivano in successione rapida, può accadere che uno solo di questi venga rilevato).

Segnali - Azioni associate

- Quando un processo “ riceve” un segnale può reagire in uno dei seguenti modi
 - **Ignore**: il segnale viene ignorato e non ha alcun effetto (**SIGKILL** e **SIGSTOP** non possono essere ignorati questo permette al superuser di terminare processi)
 - **Catch**: esegue un ***signal handler***, una funzione che gestisce l'evento associato al segnale
 - ▶ sospende il flusso di esecuzione corrente
 - ▶ esegue il signal handler
 - ▶ riprende il flusso di controllo originario quando il signal handler termina.

Segnali - Azioni associate

- La reazione ad un segnale può essere definita dal programmatore o può essere quella di default stabilita dal kernel.
 - Ogni segnale ha un'azione di default, che può essere (dopo la ricezione):
 - Segnale scartato (ignore)
 - Terminazione processo
 - ▶ generando un file core (dump)
 - ▶ senza generare un file core (quit)
 - Sospensione processo (suspend)
 - Riprende l'esecuzione del processo (resume)

Alcuni segnali predefiniti

Macro	#	Default	Description
SIGHUP	1	quit	hang up
SIGINT	2	quit	interrupt
SIGQUIT	3	dump	quit
SIGILL	4	dump	illegal instruction
SIGTRAP	5	dump	trace trap
SIGABRT	6	dump	abort
SIGEMT	7	dump	emulator trap instruction
SIGFPE	8	dump	arithmetic exception
SIGKILL	9	quit	kill (cannot be caught, blocked or ignored)
SIGBUS	10	dump	bus error (bad format address)
SIGSEGV	11	dump	segmentation violation (out-of-range address)
SIGSYS	12	dump	bad argument to system call
SIGPIPE	13	quit	write on a pipe/socket with no one to read it
SIGALRM	14	quit	alarm clock
SIGTERM	15	quit	software termination signal

Alcuni segnali predefiniti

Macro	#	Default	Description
SIGUSR1	16	quit	user signal 1
SIGUSR2	17	quit	user signal 2
SIGCHLD	18	ignore	child status changed
SIGPWR	19	ignore	power fail or restart
SIGWINCH	20	ignore	window size change
SIGURG	21	ignore	urgent socket condition
SIGPOLL	22	exit	pollable event
SIGSTOP	23	quit	stopped (signal)
SIGSTP	24	quit	stopped (user)
SIGCONT	25	ignore	continued
SIGTTIN	26	quit	stopped (tty input)
SIGTTOU	27	quit	stopped (tty output)
SIGVTALRM	28	quit	virtual timer expired
SIGPROF	29	quit	profiling timer expired
SIGXCPU	30	dump	CPU time limit exceeded
SIGXFSZ	31	dump	file size limit exceeded

Alcuni dei segnali più importanti

- **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

- **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

- **SIGSYS (Terminazione, core)**
 - Invocazione non valida di system call
 - Esempio: parametro non corretto
- **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

Segnali da Tastiera

- È possibile inviare un segnale ad un processo in foreground premendo <Ctrl-c> o <Ctrl-z> dalla tastiera:
 - Quando il driver di un terminale riconosce che è stato premuto <Ctrl-c> (<Ctrl-z>) invia un segnale **SIGINT (SIGSTP)** a tutti i processi nel gruppo del processo in foreground.
 - Alcuni segnali collegati
 - **SIGCHLD**: inviato al padre da un figlio che termina;
 - **SIGSTOP**: sospensione da dentro un programma;
 - **SIGCONT**: riprende l'esecuzione di un programma dopo una sospensione

Richiedere un segnale di allarme

- **Funzione**

```
unsigned int alarm (unsigned int count)
```

- **Istruisce il Kernel a spedire il segnale SIGALRM al processo invocante dopo count secondi:**

- Un eventuale **alarm()** già schedulato viene soprascritta col nuovo
- Se count è **0**, non schedula nessun nuovo **alarm()** (e cancella quello eventualmente già schedulato).
- Restituisce il numero di secondi rimanenti prima dell'invio dell'allarme, oppure **0** se non è schedulato nessun **alarm()**

Nota: l'allarme è inviato dopo almeno **count** secondi, ma il meccanismo di scheduling può ritardare ulteriormente la ricezione del segnale.

Richiedere un segnale di allarme

- **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 funzione sospende il processo fino a quando un segnale non viene catturato
 - Ritorna `-1` e setta `errno` a `EINTR`

Segnali – Funzione signal

- **Funzione**

```
void (*signal(int signum, void (*handler)(int)))(int)
```

- **Definizione alternativa:**

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

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

- **Descrizione:**

- Installa un nuovo signal handler, **handler**, per il segnale con numero **signum**
- Restituisce il precedente signal handler associato a **signum**, se ha successo; altrimenti, **-1**.

Segnali – Funzione signal

- **Handler può essere**
 - **Indirizzo** di una funzione handler definita dall'utente
 - ▶ La funzione handler ha un argomento intero che rappresenta il numero del segnale. Questo permette di utilizzare lo stesso handler per segnali differenti.
 - Uno dei seguenti valori
 - ▶ **SIGN_IGN**: indica che il segnale dev'essere ignorato
 - ▶ **SIGN_DFL**: indica che deve essere usato l'handler di default fornito dal nucleo

Nota: Il nome di una funzione è un valore puntatore a funzione. Una funzione in una dichiarazione di parametro viene interpretata dal compilatore come un puntatore.
Quindi nel prototipo di signal() si può togliere "*" ...

Segnali – Funzione signal

- I segnali **SIGKILL** e **SIGSTP** non sono riprogrammabili
 - Con la creazione di nuovi processi
 - ▶ Dopo una **fork()**, il processo figlio eredita le politiche di gestione dei segnali del padre
 - ▶ Se il figlio esegue una **exec()**
 - i segnali precedentemente ignorati continuano ad essere ignorati
 - i segnali i cui handler erano stati ridefiniti sono ora gestiti dagli handler di default.
 - Ad eccezione di **SIGCHLD**, i segnali non sono accodati
 - ▶ quando più segnali dello stesso tipo arrivano “contemporaneamente”, solo uno viene trattato

Segnali - Generazione

```
/* catch_ctrl.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void catch_int(int sig_num) {
    printf("\nDon't do that\n"); /* print the message */
    fflush(stdout);
}

int main (void) {

    /* set the INT (Ctrl-C) signal handler to 'catch_int' */
    signal(SIGINT, catch_int);
    for ( ;; )
        pause();
}
```

Segnali - Generazione

- **Output**

```
$ gcc catch_ctrl.c -o test
```

```
$ ./test
```

^C

Don't do that

^C

Don't do that

^C

Don't do that

...

Segnali – altro esempio

- Il programma **critical1.c** suggerisce come si possono proteggere pezzi di codice “critici” da interruzioni dovute a <Ctrl-c> (segnale **SIGINT**) o ad altri segnali simili che possono essere ignorati.
- Procede nel modo seguente
 - ▶ salva il precedente valore dell'handler, indicando che il segnale deve essere ignorato
 - ▶ esegue la regione di codice protetta
 - ▶ ripristina il valore originale dell'handler

Segnali - altro esempio

```
/* critical1.c */
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int main (void) {
    void (*oldHandler) (int); /* To hold old handler value */
    printf ("I can be Control-C'ed\n");
    sleep (3);
    oldHandler = signal (SIGINT, SIG_IGN); /* Ignore Control-C */
    printf ("I'm protected from Control-C now\n");
    sleep (3);
    signal (SIGINT, oldHandler); /* Restore old handler */
    printf ("I can be Control-C'ed again\n");
    sleep (3);
    printf ("Bye!\n");
    return 0;
}
```

Segnali - altro esempio

- **Output**

```
$ gcc critical1.c -o test  
$ ./test  
I can be Control-C'ed  
I'm protected from Control-C now  
I can be Control-C'ed again  
Bye!  
$
```

Segnali - Generazione

- **System call:** `int kill(pid_t pid, int signo);`
 - La funzione `kill` spedisce un segnale ad un processo oppure a 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`
 - * se il mittente ha per proprietario il superuser, invia il segnale a tutti i processi, mittente incluso;
 - * se il mittente non ha per proprietario un superuser, invia il segnale a tutti i processi nello stesso gruppo del mittente, con esclusione del mittente;
 - Argomento `signo`:
 - Numero di segnale spedito

Nota: la funzione `kill` restituisce valore `0`, se invia con successo almeno un segnale; altrimenti, restituisce `-1`.

Segnali - Generazione

- **System call: `int kill(pid_t pid, int signo);`**
 - Permessi:
 - Il superuser può spedire segnali a chiunque
 - i processi mittente e destinatario hanno lo stesso proprietario; più precisamente real o effective uid del mittente coincide con real o effective uid del destinatario
 - il processo mittente ha come proprietario il superuser
 - 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

Segnali - Esempio

- **Esempio: sigusr.c**

- Cattura i segnali definiti dall'utente e stampa un messaggio di errore

```
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

static void sig_usr(int); /* one handler for both signals */

int main(void) {
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        perror("can't catch SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        perror("can't catch SIGUSR2");

    for ( ; ; )
        pause();
}
```

Segnali - Esempio

■ Esempio: sigusr.c

- Cattura i segnali definiti dall'utente e stampa un messaggio di errore

```
static void sig_usr(int signo) {    /* argument is signal number */
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else {
        printf("received signal %d\n", signo);
        abort();
        exit(1);
    }
    return;
}
```

Segnali - Esempio

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

```
$ gcc sigusr.c -o test
$ ./test &
[1]    9126
$ kill -USR1 9126 # spedisci segnale SIGUSR1 a 235
received SIGUSR1 # catturato
$ kill -USR2 9126 # spedisci segnale SIGUSR1 a 235
received SIGUSR2 # catturato
$ kill 9126      # spedisci segnale SIGTERM
[1] + Terminated: 15      ./test
```

Segnali – altro esempio

- Il programma **limit.c** permette all'utente di limitare il tempo impiegato da un comando per l'esecuzione. Ha la seguente interfaccia

limit nsec cmd args

- esegue il comando **cmd** con gli argomenti **args** indicati, dedicandovi al massimo **nsec** secondi
- Il programma definisce un handler per il segnale **SIGCHLD**

Segnali - altro Esempio

```
/* limit.c */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int delay;

/* death-of-child handler*/
void childHandler (int sig) { /* Executed if the child dies */
    int childPid, childStat; /* before the parent */
    childPid = wait (&childStat); /* Accept child's termination code */
    printf ("Child %d terminated within %d seconds\n", childPid, delay);
    exit (0); /* EXITSUCCESS */
}
```

Segnali - altro Esempio

```
int main (int argc, char *argv[]) {
    int pid;
    signal (SIGCHLD, childHandler); /* Install death-of-child handler */
    pid = fork (); /* Duplicate */
    if (pid == 0) { /* Child */
        execvp (argv[2], &argv[2]); /* Execute command */
        perror ("limit"); /* Should never execute */
    }
    else { /* Parent */
        sscanf (argv[1],
                "%d",
                &delay); /* Read delay from command line */
        sleep (delay); /* Sleep for the specified number of seconds */
        printf ("Child %d exceeded limit and is being killed\n"
                , pid);
        kill (pid, SIGINT); /* Kill the child */
    }
    return 0;
}
```

Segnali - altro Esempio

- **Output:**

```
$ gcc limit.c -o limit
$ ./limit 3 find / -name filechenonce
find: /.DocumentRevisions-V100: Permission denied
find: /.fsevents.d: Permission denied
find: /.Spotlight-V100: Permission denied
find: /.Trashes: Permission denied
Child 11534 exceeded limit and is being killed
$ ./limit 3 pwd
/Users/rdevirgilio
Child 11545 terminated within 3 seconds
```

Segnali – altro esempio (sospensione e ripresa)

- Il programma **pulse.c** crea due figli che entrano in un ciclo infinito e mostrano un messaggio ogni secondo.
- Il padre aspetta 2 secondi e quindi sospende il primo figlio, mentre il secondo figlio continua l'esecuzione
- Dopo altri 2 secondi il padre riattiva il primo figlio, aspetta altri 2 secondi e quindi termina entrambi i figli

Segnali - altro Esempio (sospensione e ripresa)

```
/* pulse.c */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int main (void) {
    int pid1, pid2;
    pid1 = fork ();
    if (pid1 == 0) { /* First child */
        while (1) { /* Infinite loop */
            printf ("pid1 is alive\n");
            sleep (1);
        }
    }
    pid2 = fork ();
    if (pid2 == 0) { /* Second child */
        while (1) { /* Infinite loop */
            printf ("pid2 is alive\n");
            sleep (1);
        }
    }
}
```

Segnali - altro Esempio (sospensione e ripresa)

```
sleep (2);
kill (pid1, SIGSTOP); /* Suspend first child */
sleep (2);
kill (pid1, SIGCONT); /* Resume first child */
sleep (2);
kill (pid1, SIGINT); /* Kill first child */
kill (pid2, SIGINT); /* Kill second child */
return 0;
}
```

Segnali - altro Esempio (sospensione e ripresa)

- Output:

```
$ gcc pulse.c -o pulse
$ ./pulse
pid1 is alive
pid2 is alive
pid1 is alive
pid2 is alive
pid1 is alive
pid2 is alive
pid2 is alive ← Suspend first child
pid2 is alive
pid2 is alive
pid1 is alive ← Resume first child
pid1 is alive
pid2 is alive
$
```

Segnali – System call

- **System call:** `void abort();`
 - questa system call 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

Segnali – Funzioni reentrant

- **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...

Segnali – Funzioni reentrant

- **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

Segnali – Standard POSIX

- **POSIX e S.O. moderni:**
 - Capacità di bloccare 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

■ Alcune definizioni:

- Diciamo che un segnale è *generato* per un processo quando accade l'evento associato al segnale
 - Esempio: riferimento memoria non valido ⇒ **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

■ Bloccare i segnali

- Un processo ha l'opzione di bloccare la consegna di un segnale per cui l'azione di default non è 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 ad 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

- **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**

Segnali affidabili – Segnali multipli

▪ Signal set

- Abbiamo bisogno di un tipo di dati per rappresentare segnali multipli
 - necessario in funzioni tipo **sigprocmask**
 - implementazione dipendente dal s.o., e in particolare dal numero di segnali definiti
- Il tipo **sigset_t** è definito in **signal.h**

▪ Operazioni su **sigset_t**:

- **int sigemptyset(sigset_t *set);**
 - Inizializza la struttura dati puntata da **set** ad un insieme vuoto
- **int sigfillset(sigset_t *set);**
 - Inizializza la struttura dati puntata da **set** ad un insieme che contiene tutti i segnali

Segnali affidabili – Segnali multipli

- **Operazioni su `sigset_t`:**
 - **`int sigaddset(sigset_t *set, int signo);`**
 - Aggiunge il segnale `signo` all'insieme puntata da `set`
 - **`int sigdelset(sigset_t *set, int signo);`**
 - Rimuove il segnale `signo` dalla struttura dati puntata da `set`
 - **`int sigismember(sigset_t *set, int signo);`**
 - Ritorna true se il segnale `signo` appartiene alla struttura dati puntata da `set`

Mascherare segnali

- **System call**

```
int sigprocmask(int how, sigset_t *set, sigset_t *oset);
```

- **Descrizione:**

- Argomento **oset**:
 - Se è diverso da **NULL**, al termine della chiamata questa struttura dati conterrà la maschera precedente
- Argomento **set**:
 - Se è diverso da **NULL**, allora l'argomento **how** descrive come la maschera viene modificata
- Argomento **how**
 - **SIG_BLOCK**: blocca i segnali compresi in **set** (bitwise or)
 - **SIG_UNBLOCK**: sblocca i segnali compresi in **set**
 - **SIG_SETMASK**: **set** è la nuova maschera

Segnali affidabili – Gestione segnali

- **Esempio:**

- Stampa parte del contenuto della maschera dei segnali
- Può essere richiamato da un signal handler (salvataggio `errno`)

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

Segnali affidabili – Gestione segnali

- **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;  
}
```

Segnali affidabili – Gestione segnali

- **Esempio: critical2.c;**
 - viene bloccato **SIGINT** (<Ctrl-C>) , salvando la maschera corrente
 - si esegue **sleep** per 5 secondi
 - al termine dello **sleep**, si verifica se **SIGINT** è pendente
 - sblocca **SIGINT** tornando alla maschera precedente
 - nella gestione di **SIGINT**:
 - ripristina l'azione di default

Segnali affidabili – Gestione segnali

- Esempio: critical2.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void sig_int(int signo) {
    printf("caught SIGINT\n");

    if (signal(SIGINT, SIG_DFL) == SIG_ERR)
        perror("can't reset SIGINT");
    return;
}
```

Segnali affidabili – Gestione segnali

- Esempio: critical2.c

```
int main(void) {
    sigset_t newmask, oldmask, pendmask;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        perror("can't catch SIGINT");

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);
        /* block SIGINT and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        perror("SIG_BLOCK error");

    sleep(5);    /* SIGINT here will remain pending */
```

Segnali affidabili – Gestione segnali

- Esempio: critical2.c

```
if (sigpending(&pendmask) < 0)
    perror("sigpending error");
if (sigismember(&pendmask, SIGINT))
    printf("\nSIGINT pending\n");

/* reset signal mask which unblocks SIGINT */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    perror("SIG_SETMASK error");

printf("SIGINT unblocked\n");

sleep(5); /* SIGINT here will terminate with core file */

exit(0);
}
```

Segnali affidabili – Gestione segnali

- **Esempio: critical2.c;**
 - viene bloccato **SIGINT**, salvando la maschera corrente
 - si esegue **sleep** per 5 secondi
 - al termine dello **sleep**, si verifica se **SIGINT** è pendente
 - sblocca **SIGINT** tornando alla maschera precedente
 - nella gestione di **SIGINT**:
 - ripristina l'azione di default

■ Output (1):

```
$ ./critical2
^C
SIGINT pending
caught SIGINT
SIGINT unblocked
^C
$
```

Output (2):

```
$ ./critical2
^C ^C ^C ^C ^C
SIGINT pending
caught SIGINT
SIGINT unblocked
^C
$
```

Segnali affidabili – Gestione segnali

- **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**

Segnali affidabili – Gestione segnali

■ Esempio: suspend2.c

- Attesa che un signal handler cambi il valore di una variabile globale
 - Cattura sia **SIGINT** che **SIGQUIT**
 - Termina solo con **SIGQUIT**
- Si noti:
 - L'utilizzo del modificatore **volatile** per indicare che la variabile non deve essere mantenuta in un registro
 - L'utilizzo del tipo **sig_atomic_t** per la variabile **quitflag**
 - Non necessario in POSIX
 - In ANSI C, garantisce che l'aggiornamento della variabile verrà eseguito in modo atomico

Segnali affidabili – Gestione segnali

- **Esempio: suspend2.c**

```
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

volatile sig_atomic_t quitflag; /* set nonzero by signal handler */

void err_sys(const char* t){
    perror(t);
    exit(0);
}

void sig_int(int signo)/* one signal handler for SIGINT and SIGQUIT */
{
    if (signo == SIGINT)
        printf("\ninterrupt\n");
    else if (signo == SIGQUIT)
        quitflag = 1; /* set flag for main loop */
    return;
}
```

Segnali affidabili – Gestione segnali

■ Esempio: suspend2.c

```
int main(void) {
    sigset_t newmask, oldmask, zeromask;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_int) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");

    sigemptyset(&zeromask);

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
        /* block SIGQUIT and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    while (quitflag == 0)
        sigsuspend(&zeromask);

    /* SIGQUIT has been caught and is now blocked; do whatever */
    quitflag = 0;
        /* reset signal mask which unblocks SIGQUIT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

    exit(0);
}
```

Segnali affidabili – Gestione segnali

- **Esempio: suspend2.c**

- Attesa che un signal handler cambi il valore di una variabile globale
 - Cattura sia **SIGINT** (<Ctrl-C>) che **SIGQUIT** (<Ctrl-\>)
 - Termina solo con **SIGQUIT**

- **Output**

```
$ ./suspend2
^C
interrupt
^C
interrupt
^C
interrupt
^\
$
```

Sincronizzazione tra processi

■ Esempio: tellwait1.c

- Un programma che stampa due stringhe, ad opera di due processi diversi
- L'output dipende dall'ordine in cui i processi vengono eseguiti

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

void err_sys(const char* t){
    perror(t);
    exit(0);
}

int main(void){
    pid_t pid;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {
        printf("output from child\n");
    } else {
        printf("output from parent\n");
    }
    exit(0);
}
```

■ Output

```
$ ./tellwait1
output from child
output from parent
$ ./tellwait1
output from parent
output from child
$
```

Sincronizzazione tra processi

- **Esempio: tellwait1.c**
 - Un programma che stampa due stringhe, ad opera di due processi diversi
 - L'output dipende dall'ordine in cui i processi vengono eseguiti
- **Vogliamo sincronizzare i due processi:**
 - Prima stampa il padre, poi il figlio
- **Esempio: tellwait2.c**
 - Funzione TELL_WAIT:
 - inizializza strutture dati per la sincronizzazione
 - Funzione WAIT_PARENT:
 - attendi sincronizzazione dal parent
 - Funzione TELL_CHILD
 - invia segnale di sincronizzazione al client

Sincronizzazione tra processi

- Esempio: tellwait2.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include "tellwait.c"

int main(void){
    pid_t pid;

    TELL_WAIT();

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {
        WAIT_PARENT(); /* parent goes first getppid()*/
        printf("output from child\n");
    } else {
        printf("output from parent\n");
        TELL_CHILD(pid);
    }
    exit(0);
}
```

- Output

```
$ ./tellwait2
output from parent
output from child
$
```

Sincronizzazione tra processi

■ Esempio: tellwait.c

- Implementazione delle funzioni di sincronizzazione utilizzate nel programma precedente
- Basate sullo scambio di segnali **SIGUSR1** e **SIGUSR2**
- Funzione **TELL_WAIT** inizializza il processo, bloccando la ricezione di **SIGUSR1** e **SIGUSR2**
- Le funzioni **TELL_**
 - utilizzano la system call **kill**
- Le funzioni **WAIT_**
 - utilizzano le system call **sigsuspend**

Note ai Segnali

- Handler corti
- Mascherare correttamente
- Attenzione coi segnali di “fault” (**SIGBUS, SIGSEGV, SIGFPE**)
- Attenzione coi timer
- I segnali non sono pensati per realizzare ambienti “event driven”

Pipe

- **Cos'è un pipe?**

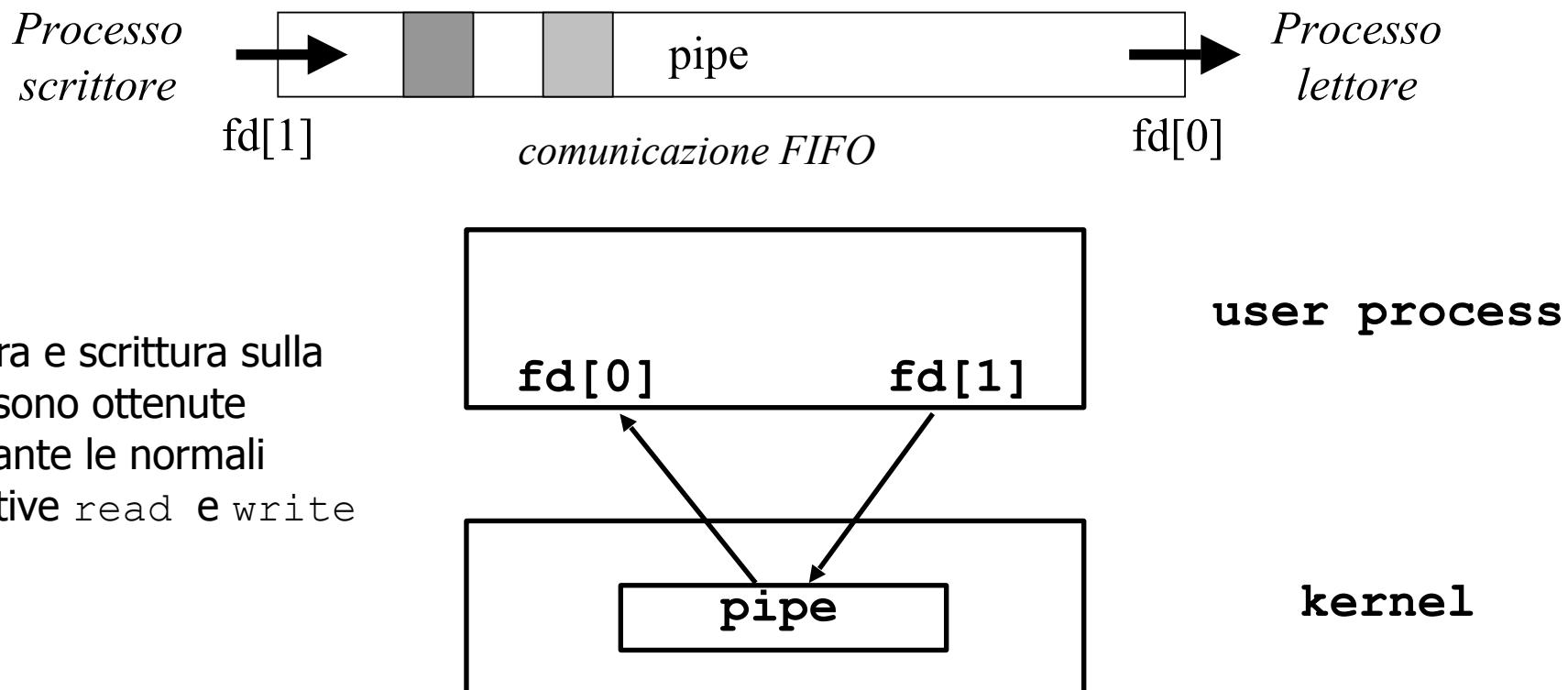
- E' un canale di comunicazione che unisce due processi

- **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 "**antenato**" 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

Pipe

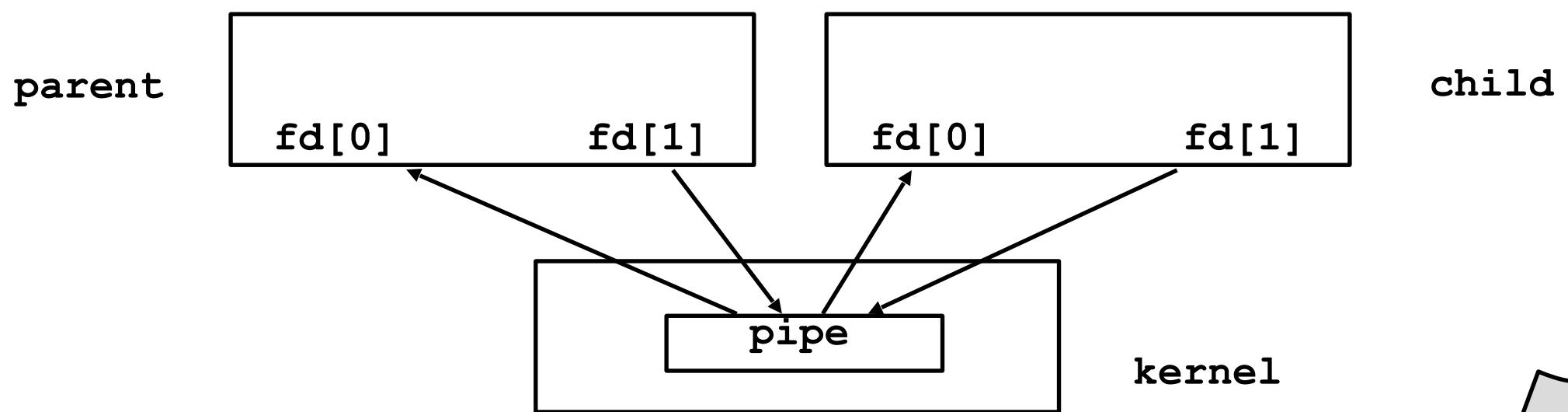
- System call: `int pipe(int filedes[2]);`
 - Ritorna due descrittori di file attraverso l'argomento `filedes`
 - `filedes[0]` è aperto in lettura
 - `filedes[1]` è aperto in scrittura
 - L'output di `filedes[1]` è l'input di `filedes[0]`



Pipe

■ 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 tra parent e child o viceversa



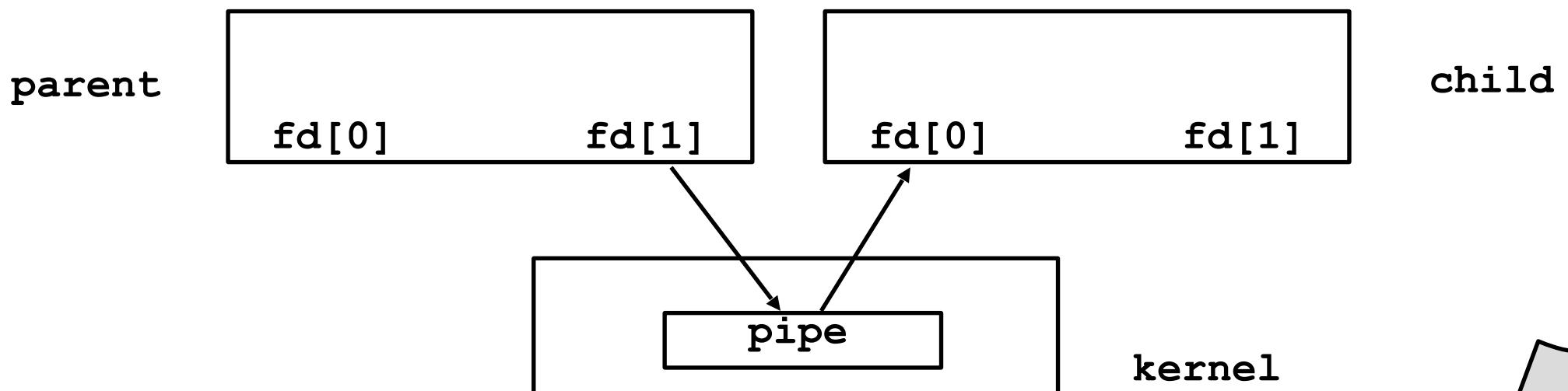
Pipe

- **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 input (**close(fd[0])** ;)
- Il child chiude l'estremo di output (**close(fd[1])** ;)



Pipe

■ Esempio: pipe1.c

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

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

#define MAXLINE 1024

void err_sys(const char* t){
    perror(t);
    exit(0);
}
```

Pipe

- Esempio: pipe1.c

```
int main(void) {
    int    n, fd[2]; pid_t pid; char line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");

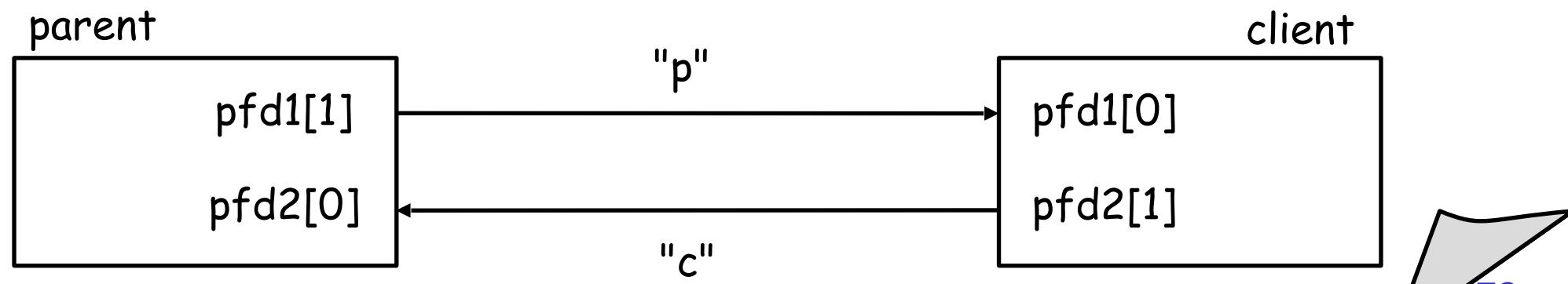
    else if (pid > 0) { /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);

    } else {           /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n); // printf("%s", line);
    }

    exit(0);
}
```

Pipe per sincronizzazione

- Esempio già visto: **tellwait2.c**
 - Un programma che stampa due stringhe, ad opera di due processi diversi
 - Basato su funzioni tell, wait
- Implementazione basata su pipe: **tellwaitpipe.c**
 - Vengono aperte due pipe (TELL_WAIT)
 - Funzioni TELL: emettono un carattere "p", "c"
 - Funzione WAIT: consumano un carattere "p", "c"





```
Current conditions at Pescara, Italy (IBP) 42.26N 014.12E 11M (IBP)
Last updated Feb 10, 2012 - 02:50 PM EST / 2012-02-10 1950 UTC
Temperature: 1 C
Relative Humidity: 80%
Wind: from the W (270 degrees) at 15 MPH (13 KT) gusting to 45 KPH
Weather: light snow grains
Sky conditions: overcast
Su Mo Tu We Th Fr Sa      Su Mo Tu We Th Fr Sa
feb 29 30 31 01 02 03 04    mar 04 05 06 07 08 09 10
  05 06 07 08 09 10 11    11 12 13 14 15 16 17
  12 13 14 15 16 17 18    18 19 20 21 22 23 24
  19 20 21 22 23 24 25    25 26 27 28 29 30 31
mar 26 27 28 29 01 02 03    apr 01 02 03 04 05 06 07
silvio@Stain ~ $ cd Video
silvio@Stain ~ /Video $ movgrab http://vimeo.com/27998081
Formats available for this Movie: flv
Selected format: flv
Progress: 61.47% 15.4M of 25.1M 693.6K/s
```

