Primitive di Comunicazione

Inter-process communication

- Nell'attività del sistema si rende necessario la cooperazione tra processi per realizzare le diverse funzionalità offerte all'utente.
- La cooperazione comporta una comunicazione reciproca con scambio di informazioni, i.e. ***interprocess communication***:
  - **Trasferimento dati**

    Invio di dati significativi tra processi sulla stessa macchina e non.
  - **Condivisione di dati**

    Processi che operano s porzioni di memoria condivisa. Le modifiche ai dati sono immediatamente visibili a tutti i processi.
  - **Notifica di eventi**

    Processi che notificano altri processi sul verificarsi una condizione specifica.
  - **Condivisione di risorse**

    Le risorse sono sempre limitate rispetto ai processi che le utilizzano. Si definiscono quindi meccanismi di politiche di accesso tra processi che sfruttano primitive del kernel.
  - **Controllo tra processi**

    Un processo debugger controlla totalmente l'esecuzione di altri processi.

Inter-process communication con segnali

- Abbiamo visto che i segnali sono utilizzati per la notifica di eventi tra processi.
- L'azione di default dipende dal tipo di segnale. In genere si ha la terminazione del processo che lo riceve.
- `signal()` permette di definire azioni di risposta con funzioni proprietarie:

  È possibile implementare rudimentali schemi di sincronizzazione e/o comunicazione.
- **Limitazioni**:
  - **Onerosi** dato che comportano la gestione di interrupts da pate del kernel.
  - **I dati inviati sono limitat**i all'informazione a cui il segnale è associato.
  - **Difficili da gestire** in schemi di comunicazione complessi.
  - La notifica avviene in maniera **asincrona**.

Inter-process communication: altre implementazioni

- **Oltre ai segnali ci sono svariati metodi per realizzare l'IPC:**
  - **Anonymous pipes**
  - **FIFOs** o **named pipes**
  - **Sockets**
  - Message queues
  - Shared memory
  - Semaphores

Inter-process communication con 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

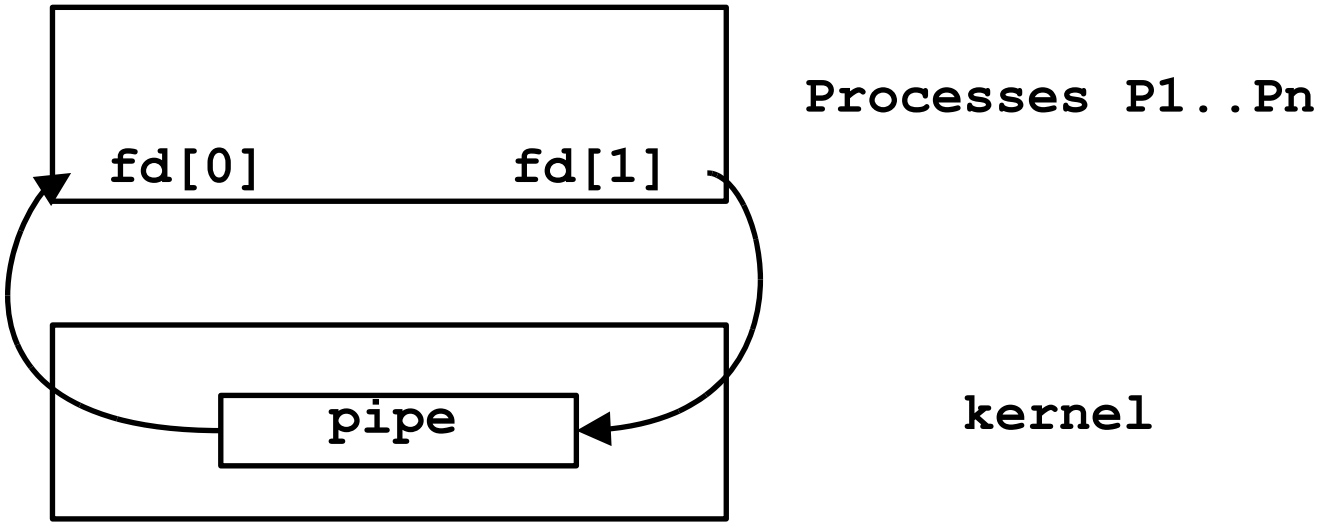
Pipe

- **Può essere vista come un vero e proprio 'tubo' in cui:**
  - Ad una estremità **si inviano solo dati:**
    - Se la pipe è piena, chi scrive deve attendere che qualche processo legga qualche dato.
  - All'altra estremità **si ricevono solo dati:**
    - I dati vengono letti e consumati (non sono più disponibili).
    - Il processo si blocca fino a quando non c'è qualche dato da leggere.
- **Caratteristiche:**
  - Il canale è **unidirezionale** in un solo senso (half-duplex).
  - I dati sono inviati come **first-in first-out** in forma non strutturata.
  - Il kernel mantiene la pipe per mezzo del file system.

Pipe

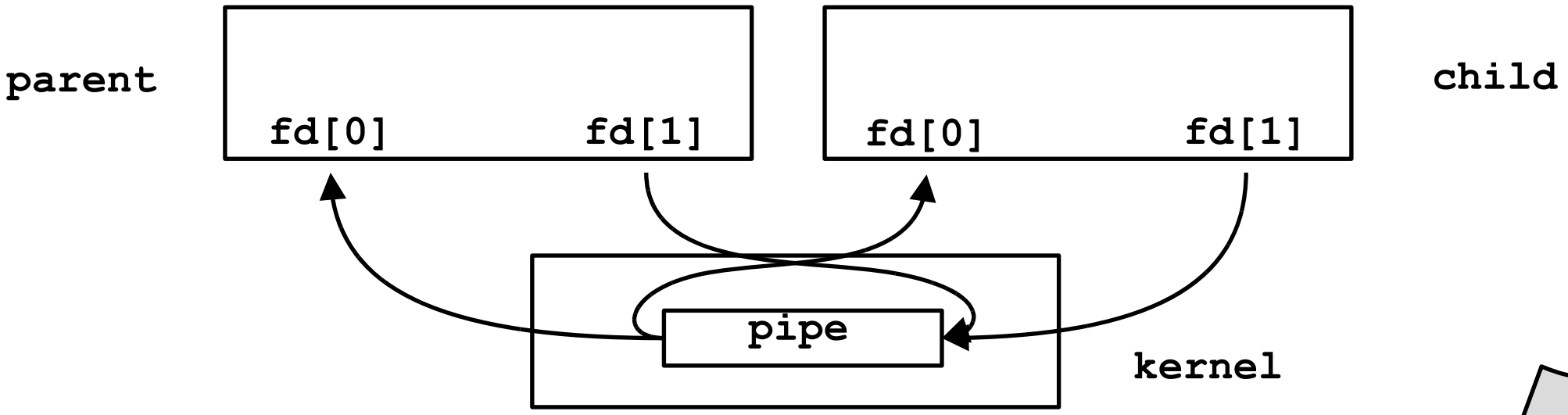
```
#include <unistd.h>

int pipe(int filedes[2]); // -1 for error and set errno
```

- 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]`



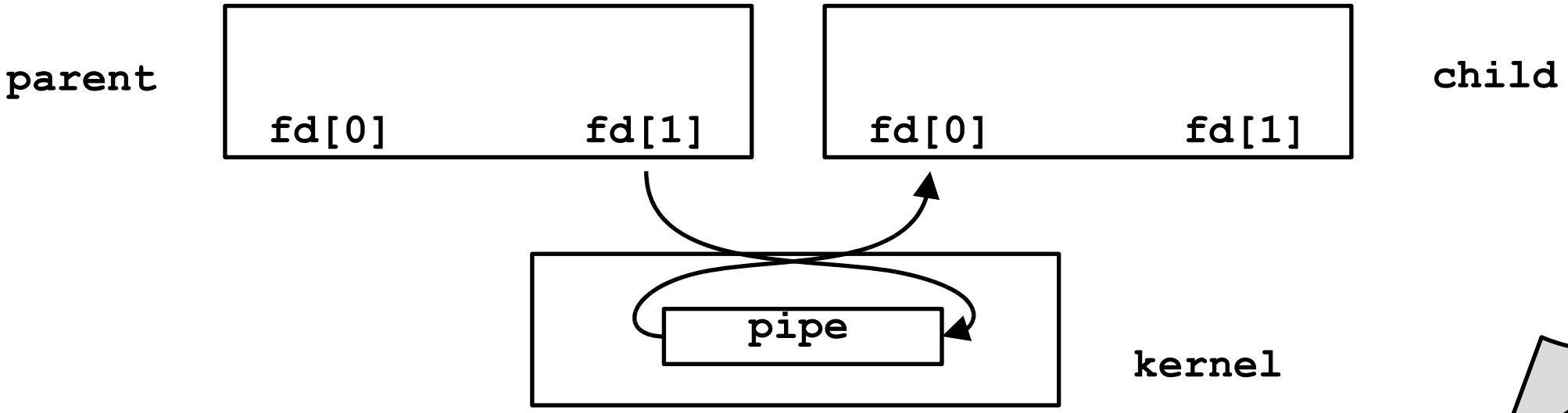
Pipe

- **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]);`)



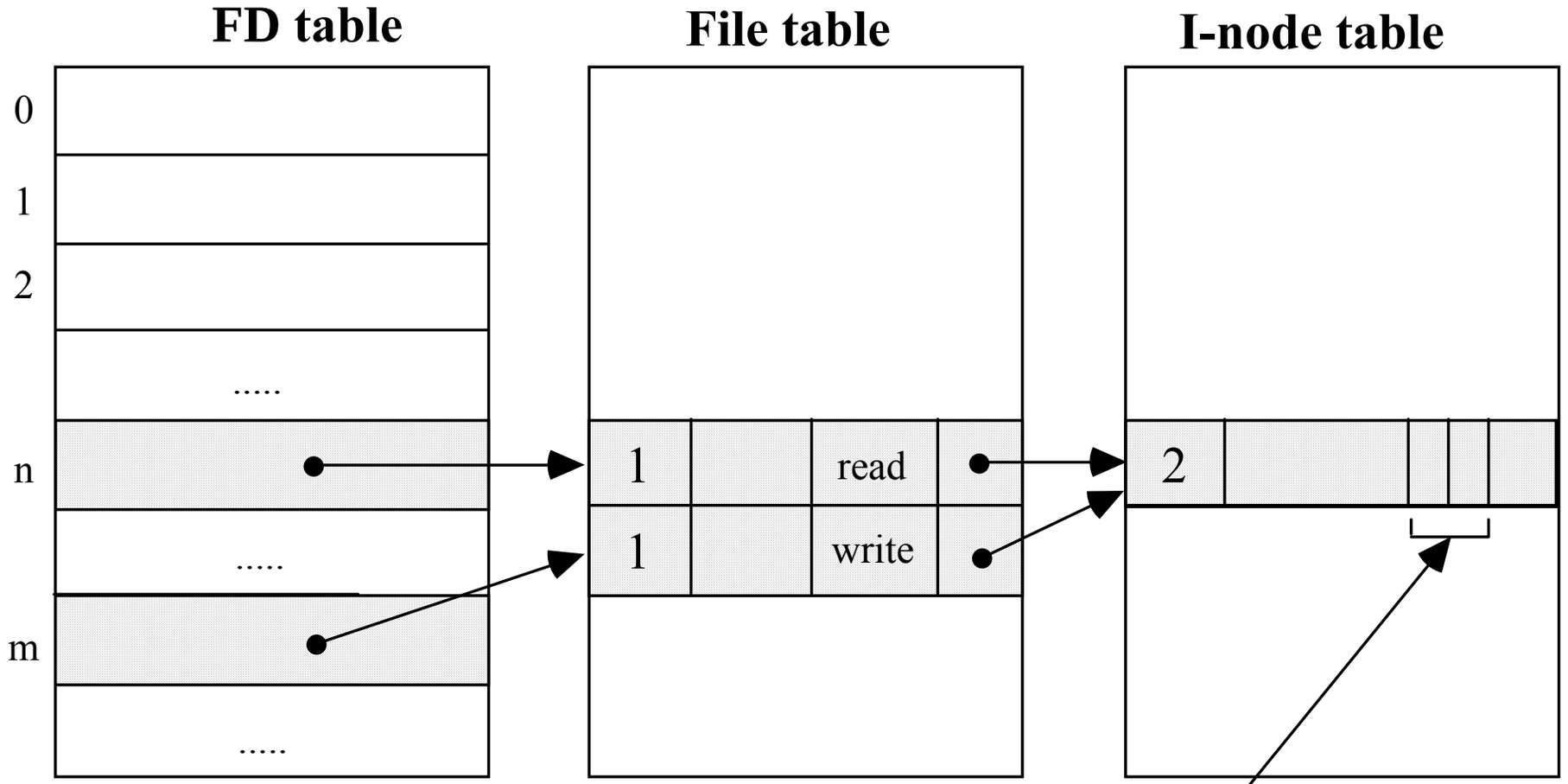
Pipe – esempio di invocazione

```
#include  <unistd.h>
…

int main(void)
{
  int   fd[2]; /* pipe's endpoints */
  pid_t pid;

  if (pipe(fd) < 0) /* errno filled */
    /* error code */

  if ( (pid = fork()) < 0)
    /* error code */

  if (pid > 0) {   /* parent */
    close(fd[0]);
    /* only send data from parent */

  } else {         /* child */
    close(fd[1]);
    /* only read data from child */
  }

  exit(0);
}
```

Pipe – implementazione nel kernel

- **La pipe viene gestita dal kernel per mezzo del file system:**
  - Crea una nuova entry nella tabella degli i-node
  - Crea due nuove entry nella tabella dei file:
    - Un link per puntare l'i-node in scrittura (`filedes[1]`)
    - Un link per puntare l'i-node in lettura (`filedes[0]`)
  - I due rispettivi descrittori sono inseriti nella tabella dei file aperti del processo
- **Utilizzo:**
  - Dato che una pipe è vista come un file, è possibile accederci con le consuete operazioni di `read` / `write`
  - Tuttavia, dato che l'offset è gestito *ad-hoc*, non sono possibili alcune operazioni tipo `lseek`

Pipe



pipe (n | m)

0 1

read e write offset (stanno qui e non nella file table, perché devono essere visti da entrambi i processi)

Pipe

- **La chiamata `read`**
  - Per leggere i dati si può utilizzare la comune chiamata a **`read`**:

```
ssize_t read(int filedes, void *buf, size_t nbyte);
```

- **Condizioni:**
  - se l'estremo di ouput 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 output è stato **chiuso**
    - restituisce i dati disponibili, ritornando il numero di byte
    - successive chiamate ritornano 0

Pipe

- **La chiamata `write`**
  - Per inviare i dati utilizziamo invece la **`write`**:

```
ssize_t write(int filedes, const void *buf, size_t nbyte);
```

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

Pipe – esempio (2) di invocazione

```
#include  <unistd.h>
…
#define MAXLINE 1024
int main(void)
{
  int   n, fd[2];                          /* pipe's endpoints */
  pid_t pid;
  char  line[MAXLINE];

if (pipe(fd) < 0)                          /* errno filled if error */
    /* error code */

  if ( (pid = fork()) < 0)
    /* error code */

  if (pid > 0) {                           /* parent */
    close(fd[0]);                          /* only send data from parent */
    write(fd[1], "hello world\n", 12);
  } else {                                 /* child */
    close(fd[1]);
    n = read(fd[0], line, MAXLINE);
    write(STDOUT_FILENO, line, n);
  }

  exit(0);
}
```

Pipe

La chiamata `fstat`:

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

Atomicità

- Quando si scrive su una pipe, la costante `PIPE_BUF` specifica la dimensione del buffer della 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.

Pipe - popen

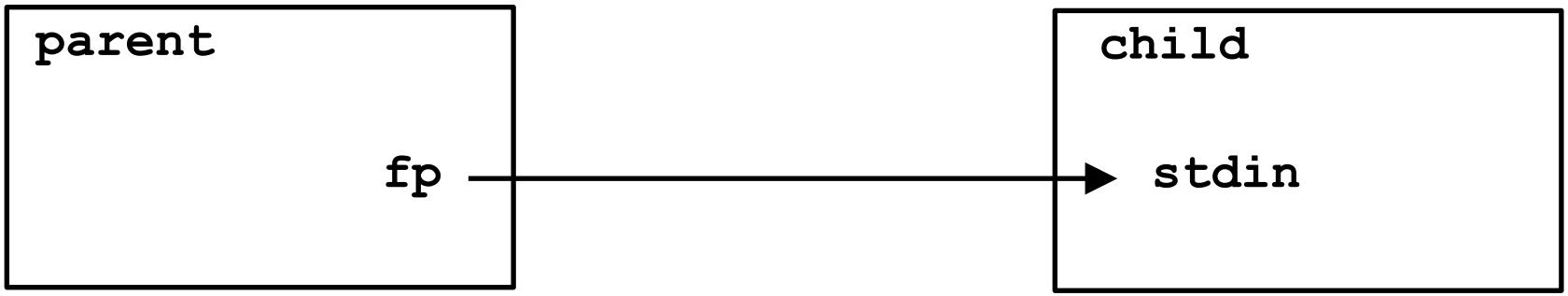
```
FILE *popen(char *command, char *type);
int pclose(FILE *fp);
```

- **Descrizione di `popen`:**
  - Apre una pipe verso un programma esterno puntato da **`command`**
  - Chiude automaticamente le parti non usate dei pipe
  - A seconda del valore di type ridireziona:
    - lo standard output del nuovo processo sul pipe → **`type = "r"`**
    - lo standard input dal nuovo processo sul pipe → **`type = "w"`**
- **Descrizione di `pclose`**
  - Attende la terminazione di **`command`** e ne restituisce l'exit status
  - Oppure in caso anomalo **`-1`** con **`errno`** settato a **`ECHILD`**

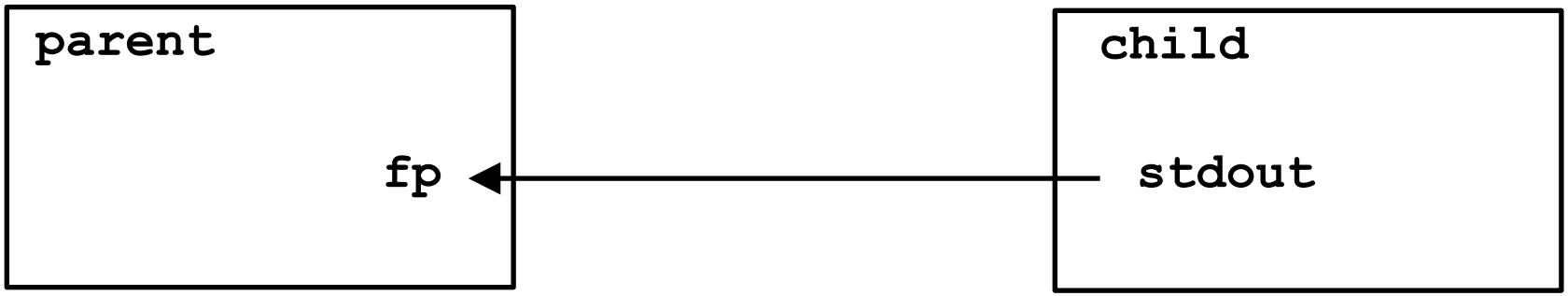
  `command` è visto come child dato che all'invocazione di **`popen`** il kernel esegue: **`fork()`** del padre; **`execve()`** sul figlio creato.

Pipe - popen

- `type = "w"`



- `type = "r"`



Pipe e named pipe

- **Stream pipe:**
  - Unix di oggi permettono di definire **stream pipe** (pipe bidirezionale)
  - I processi possono sia leggere che scrivere:
    - Vengono ritornati sempre due descrittore `filedes[0]` e `filedes[1]`
    - Entrambi supportano `read` e `write`
- **Tuttavia nelle pipe "normali":**
  - Utilizzabili solo da processi che hanno un "antenato" in comune
  - **motivo**: unico modo per ereditare descrittori di file
- **Named pipe**
  - permette a processi non collegati di comunicare
  - utilizza il file system per "dare un nome" al pipe
  - La creazione è simile alla procedura per creare file

Named pipe - FIFO

- **System call:**

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

  - crea un FIFO dal **pathname** specificato
  - la specifica dell'argomento **mode** è identica a quella di **open**
    O_RDONLY, **O_WRONLY**, **O_RDWR**, … (vedi **umask**)

- **Come funziona un FIFO?**
  - Le normali chiamate **open**, **read**, **write**, **close**, sono concesse
  - In pipe la rimozione è implicita mentre un FIFO è persistente
  - La rimozione deve essere esplicitata con **unlink**
  - I diritti di accesso sono regolati come in un file normale

Named pipe - FIFO

- **La chiamata ad `open`:**
  - File aperto senza flag **O_NONBLOCK**
    - Se il file è **aperto in lettura:**

      Si blocca fino a quando un altro processo non apre il FIFO in scrittura
    - Se il file è **aperto in scrittura:**

      Si blocca fino a quando un altro processo non apre il FIFO in lettura
  - File aperto con flag **O_NONBLOCK**
    - Se il file è **aperto in lettura:**

      Ritorna immediatamente
    - Se il file è **aperto in scrittura:**

      nessun altro processo è aperto in lettura, ritorna un messaggio di errore

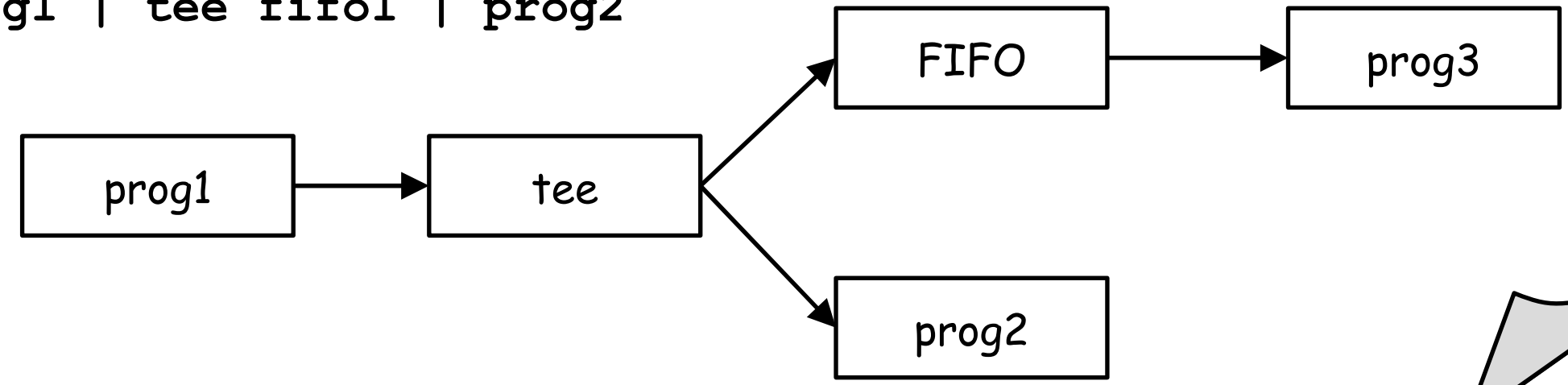
Named pipe - FIFO

- **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`** 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

Named pipe

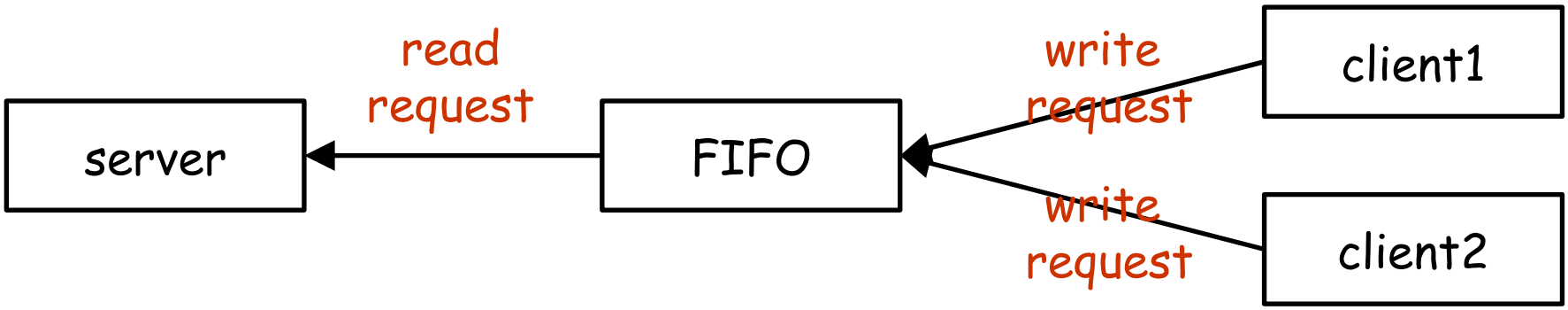
- **Utilizzazioni dei FIFO**
  - Utilizzati dai comandi shell per passare dati da una shell pipeline ad un'altra, senza passare creare file intermedi
  - "|"è il comando di pipe di Unix. Es. `prog1 | prog2` intende:
    - Lo standard di output di `prog1` è lo standard di input di `prog2`
- **Esempio:**

```
mkfifo fifo1
prog3 < fifo1 &
prog1 | tee fifo1 | prog2
```



Named pipe

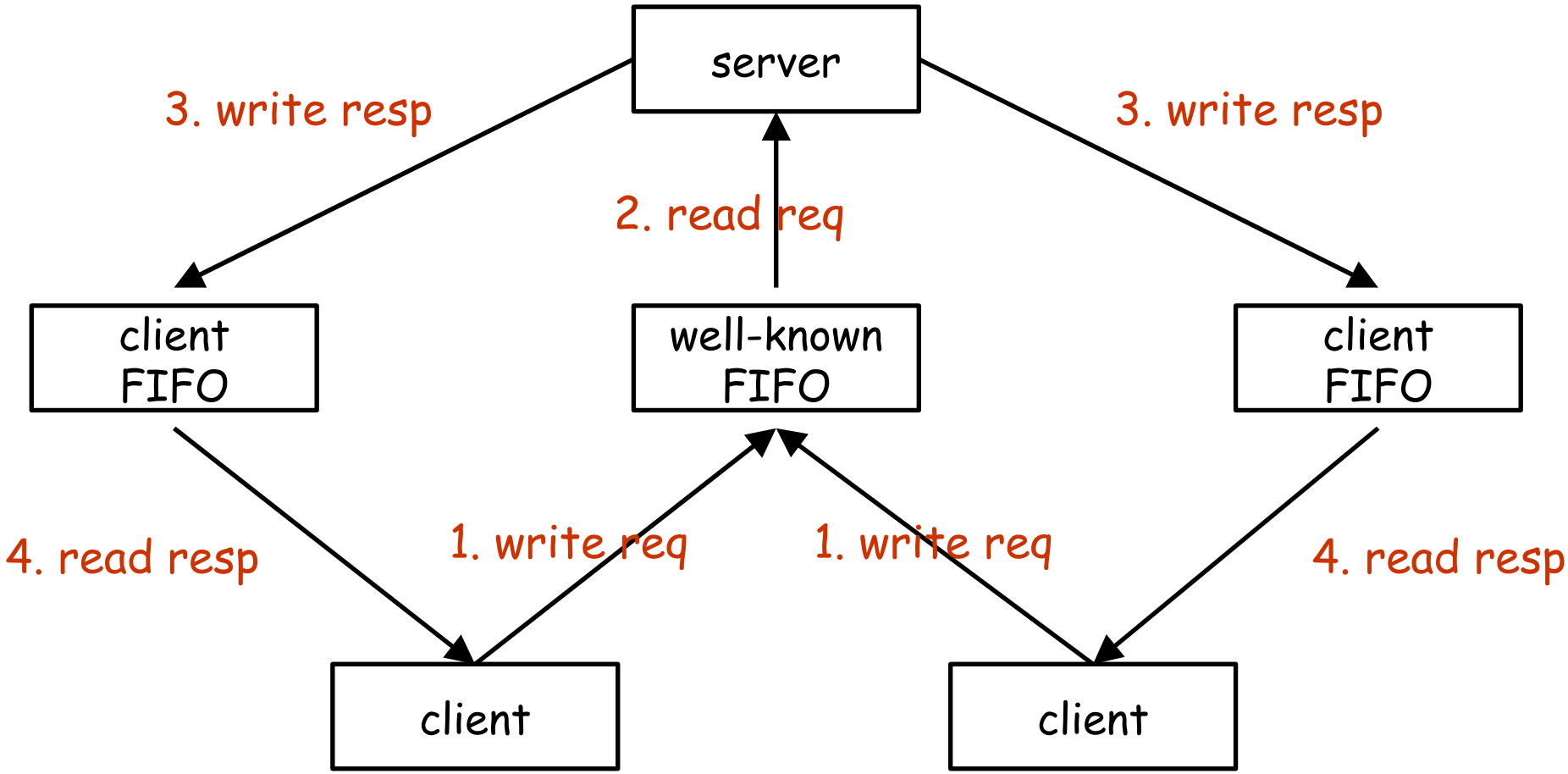
- **Utilizzazioni dei FIFO**
  - Utilizzati nelle applicazioni client-server per comunicare
- **Esempio:**
  - Comunicazioni **`client`** → **`server`**
    - il server crea un FIFO puntato da **`pathname`**
    - **`pathname`** deve essere conosciuto (well-known) da tutti i client
    - i client scrivono le proprie richieste sul FIFO
    - il server leggere le richieste dal FIFO



Named pipe

- **Problema: come rispondere ai client?**
  - Non è possibile utilizzare il "well-known" FIFO in quanto i client non saprebbero quando leggere le proprie risposte
  - Soluzione sui client:
    - spediscono il proprio **`PID`** al server
    - creano un FIFO in lettura per la risposta (il nome è il **`PID`** inviato)
  - Soluzione sul server:
    - aprono in scrittura la FIFO del client
    - scrive sul canale FIFO
  - Problemi:
    - Il server deve catturare **`SIGPIPE`**
      - il client può andarsene prima di leggere la risposta
    - Il server deve aprire in lettura il proprio FIFO
      - Altrimenti esso termina quando l'ultimo client termina (leggendo **EOF**)

Named pipe


server
3. write resp
3. write resp
2. read req
client FIFO
well-known FIFO
client FIFO
4. read resp
1. write req
1. write req
4. read resp
client
client