Pipe e file

Anno Accademico 2010-2011

Prof. Claudio Cilli

File

- In Linux, un file può essere un contenitore di informazione o uno strumento per la *comunicazione fra processi* o per interagire con dispositivi presenti sul calcolatore
  - Ecco perché spesso si dice che in Linux *tutto è un file*
  - Questa astrazione è elegante e al tempo stesso funzionale
  - Diverse tipologie di file sono distinte attraverso dei flag (si veda il manuale di `ls`, opzione `–l`)

    ```
    rw-r r 1 utente gruppo bytes Dec 4 16:30
    nomefile
    ```
  - Ogni file (aperto in lettura e/o scrittura)è identificato in maniera univoca sul sistema operativo da un *intero* non negativo (descrittore del file)

Definizione e caratteristiche di un 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: concetti generali

Una pipe è un file speciale gestito con politica First-In-First-Out (FIFO)

Specifiche: Tipicamente, può memorizzare 8 blocchi da 512KByte ciascuno

In generale, la costante `PIPE_SIZE` definisce la dimensione massima e `PIPE_BUF` definisce le dimensioni massime di ciascun blocco di dati (header `limits.h`)

Funzionamento: la pipe può essere vista come un nastro trasportatore

Un processo scrive ad una estremità della pipe (inserisce dei dati nel nastro

Un altro processo legge all'altra estremità della pipe (preleva dei dati dal nastro

Essenzialmente la pipe implementa una coda (algoritmo First-In-First-Out (FIFO))

Gestione della concorrenza

- Gestione dell'accesso: la pipe viene gestita in maniera *trasparente* dal sistema operativo
  - La scrittura viene negata se la pipe risulta piena
  - La lettura viene negata se la pipe risulta vuota
  - La lettura viene negata se la pipe non è stata aperta in scrittura(da un qualche processo
- Implementazione tra mite *buffer circolare*
  - Gestione della concorrenza basata sul modello **produttore/ consumatore**

Tipologie

- Unnamed pipe
  - Possono essere utilizzate solo da processi con qualche relazione di parentela (es.: figlio/figlio e padre/figlio)
  - La loro esistenza è strettamente legata ai processi che le usano
- Named pipe
  - Come file memorizzati all'interno di una directory (hanno un preciso nome
  - Possono essere utilizzate da processi senza alcuna relazione di parentela

Unnamed pipe

Syscall pipe

- La syscall **pipe** crea una unnamed pipe
  - Header richiesti `<unistd.h>`
  - Prototipo **`int pipe(int filedes[2])`**
  - Prende in ingresso un array di due interi nel quale la funzione memorizza i descrittori di file
    - `filedes[0]` conterrà il descrittore del file per la lettura
    - `filedes[1]` conterrà il descrittore del file per la scrittura
  - Output Restituisce 0 in caso di successo, altrimenti restituisce -1 e imposta la variabile globale `errno`
  - **NOTA**: Le prime versioni di UNIX e quelle attuali di Linux prevedono solo pipe unidirezionali

Named pipe

- Creazione: possono essere creati a livello di shell o all'interno di un programma:
  - La shell mette a disposizione il comando **`mknod`** per generare un file di tipo FIFO
  - All'interno di un programma è possibile generare delle named pipe richiamando la syscall **`mkfifo`** (quest'ultima a sua volta richiama la syscall mknod
- Caratteristiche: Qualunque processo sul calcolatore può accedere alla named pipe specificandone il *nome* (se detiene i permessi di accesso al file)

Named pipe: syscall `mkfifo`

- La syscall **`mkfifo`** crea una named pipe (file FIFO)
  - Header richiesti **`<sys/types.h>`**, **`<sys/stat.h>`**
  - Prototipo **`int mkfifo(const char *path, mode_t mode);`**
    - `path` è il nome del file che costituisce la FIFO
    - `mode` specifica i permessi sul file
  - Output: la chiamata a sistema restituisce 0 in caso di successo, -1 in caso di insuccesso. In quest'ultimo caso imposta la variabile `errno`

Named pipe: mode

- mode è un intero in formato ***ottale*** che specifica i permessi di accesso al file
- mode è nella forma ABC dove A, B, C indicano, rispettivamente, i permessi per proprietario, gruppo, altri
- I permessi per ogni categoria sono determinati da $R*2^2 + W*2^1 + X*2^0$: **R**ead, **W**rite, e**X**ecution (1 = permesso, 0 = non permesso). Ad esempio, se mode vale 231:
  - il proprietario può solo scrivere sul file (R=0, W=1, X=0: $R*2^2 + W*2^1 + X*2^0 = 2$)
  - il gruppo può scrivere e eseguire (R=0, W=1, X=1: $R*2^2 + W*2^1 + X*2^0 = 3$)
  - gli altri possono solo eseguirlo (R=0, W=0, X=1: $R*2^2 + W*2^1 + X*2^0 = 1$)

Named pipe: aprire un file

- Syscall **`open`**
  - Un file può essere aperto/creato attraverso la syscall **`open`**
  - Header richiesti **`<sys/types.h>`**, **`<sys/stat.h>`**, **`<fcntl.h>`**
  - Prototipo **`int open (const char *nomefile, int flags);`**
    - `nomefile` è una stringa contenente il nome del file (path sul filesystem
    - `flags` è un intero che determina la modalità di apertura del file. Esistono delle macro che identificano varie modalità, fra cui le più comuni sono le seguenti (combinabili tramite OR logico, **|**): O_CREAT crea il file se non presente, O_RDONLY apre il file in lettura, O_WRONLY apre il file in scrittura, O_APPEND apre il file in modalità append
  - Output: restituisce il descrittore del file aperto in caso di successo, altrimenti restituisce -1 e imposta la variabile `errno`

Named pipe: chiudere un file

- Syscall **`close`**
  - Un file aperto può essere chiuso attraverso l'utilizzo della syscall **`close`**
    - Header richiesti **`<unistd.h>`**
    - Prototipo **`int close(intfd);`**
      - `fd` è il descrittore del file da chiudere
  - Output: restituisce il valore 0 in caso di successo, altrimenti restituisce -1 e imposta la variabile `errno`

Named pipe: rimuovere un file

- Syscall **`unlink`**
- Un file chiuso può essere rimosso dal filesystem attraverso l'utilizzo della syscall **`unlink`**
  - Header richiesti **`<unistd.h>`**
  - Prototipo **`int unlink(const char *filename);`**
    - `filename` è il nome del file da eliminare (può contenere il percorso completo sul filesystem)
  - Output: restituisce il valore 0 in caso di successo, altrimenti restituisce -1 e imposta la variabile `errno`

Named pipe: scrivere su un file

- Syscall **`write`**
- La syscall **`write`** scrive un numero predefinito di byte su file
  - Header richiesti **`<unistd.h>`**
  - Prototipo **`ssize_twrite (int filedes, const void *buf,size_t nbyte);`**
    - `filedes` è il descrittore del file su cui scrivere
    - `buf` è il buffer con i dati da scrivere sul file
    - `nbyte` è il numero di byte da leggere in `buf` e scrivere sul file
  - Output: restituisce il numero di byte scritti, altrimenti, in caso di errore restituisce -1 e imposta la variabile globale `errno`

Named pipe: proprietà della syscall `write`

- I byte vengono accodati alla fine del file
- Il sistema operativo garantisce la non interferenza tra più `write()` concorrenti su uno stesso file (se il numero di byte è minore di `PIPE_BUF`)
- Di default la `write()` è bloccante

Named pipe: leggere da file

- Syscall **`read`**
- Lasyscall legge un numero predefinito di byte da file
  - Header richiesti **`<sys/types.h>, <sys/uio.h>, <unistd.h>`**
  - Prototipo **`ssize_t read(int fildes, void *buf, size_t nbyte);`**
    - `filedes` è il descrittore del file da leggere
    - `buf` è il buffer nel quale inserire i byte letti dal file
    - `nbyte` è il numero di byte da leggere sul file e scrivere su `buf`
- Outpu: restituisce il numero di byte letti, altrimenti, in caso di errore restituisce -1 e imposta la variabile globale `errno`

Named pipe: proprietà della syscall `read`

- Le letture iniziano dalla posizione corrente
  - Non sono previste operazioni di slittamento del puntatore (`seek`)
- `read()` è bloccante di default
  - Se il file è vuoto, il processo chiamante viene bloccato finché non vengono scritti dati o il file viene chiuso

Named pipe: redirezione input/ output

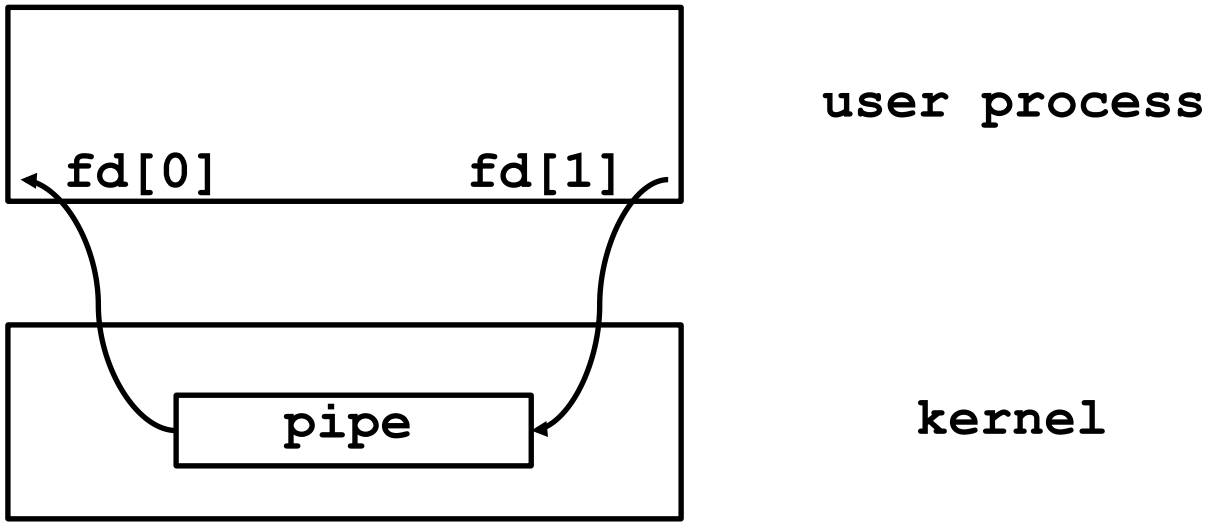
- La redirezione di input e/o ouput di un processo è necessaria se:
  - piuttosto che leggere da tastiera (standard input) abbiamo bisogno di leggere dati provenienti da un file (es.: una pipe)
  - piuttosto che scrivere su video (standard output) abbiamo bisogno di scrivere dati su un file (es.: una pipe)
- Con la syscall **`dup2`** possiamo fare in modo di associare il file a uno dei descrittori seguenti:
  - 0 standard input
  - 1 standard output
  - 2 standard error

Named pipe: redirezione input/ output

- **Syscall `dup2`**
- La syscall **`dup2`** permette di associare allo stesso file un nuovo descrittore (crea cioè un descrittore di file duplicato)
  - Header richiesti **`<unistd.h>`**
  - Prototipo **`int dup2(int fd, int dupfd);`**
    - `fd` è il descrittore del file
    - `dupfd` è il nuovo descrittore del file (duplicato
- Output: restituisce il nuovo descrittore di file, altrimenti, in caso di errore restituisce -1 e imposta la variabile globale `errno`

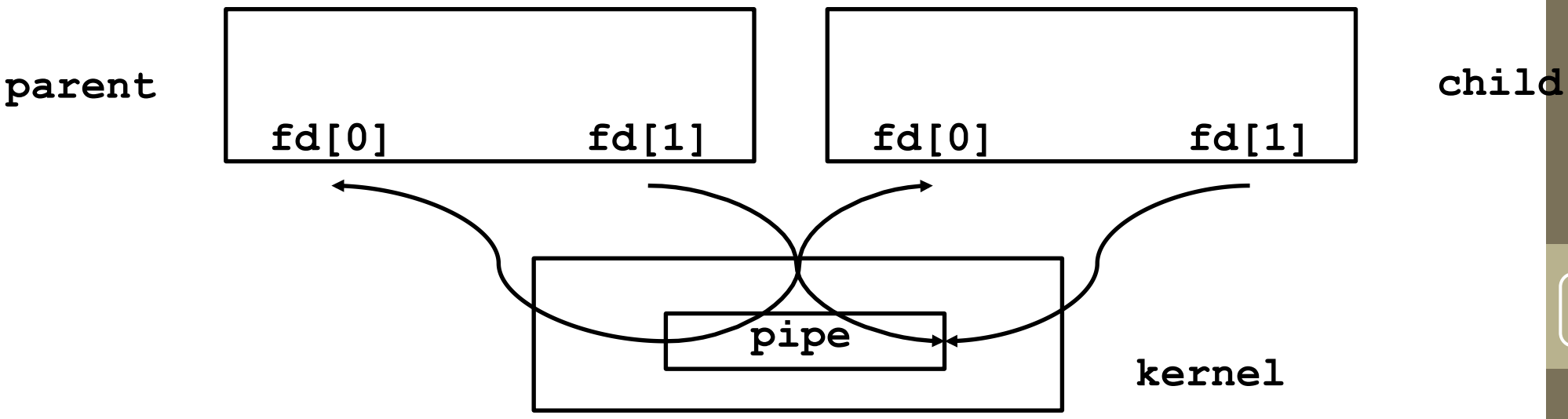
System call pipe e file descriptor

- 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]`** (estremo di write del pipe) è l'input di **`filedes[0]`** (estremo di read del pipe)



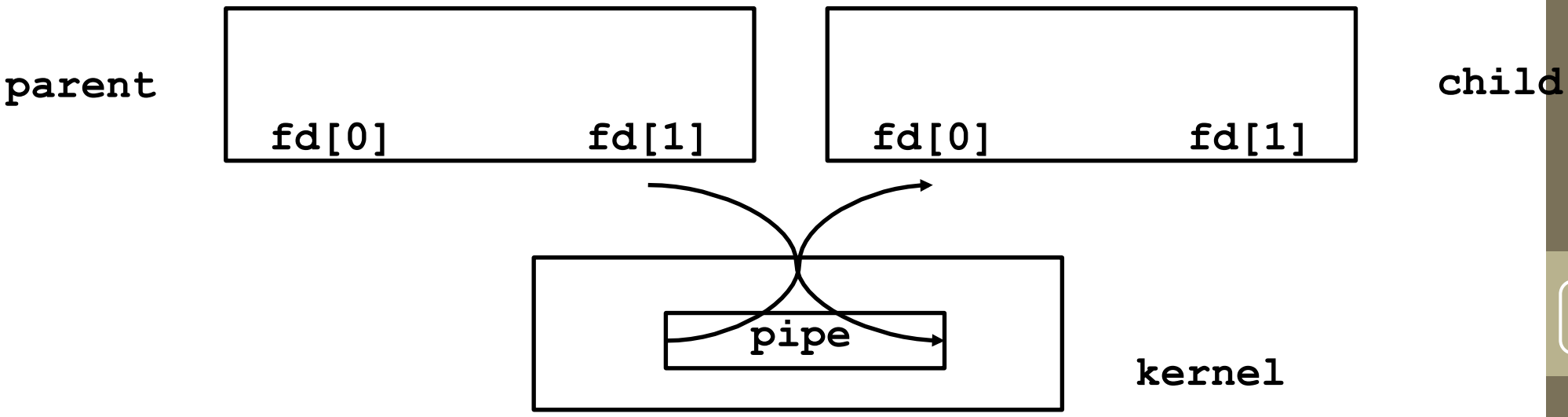
Utilizzo di pipe - I

- 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

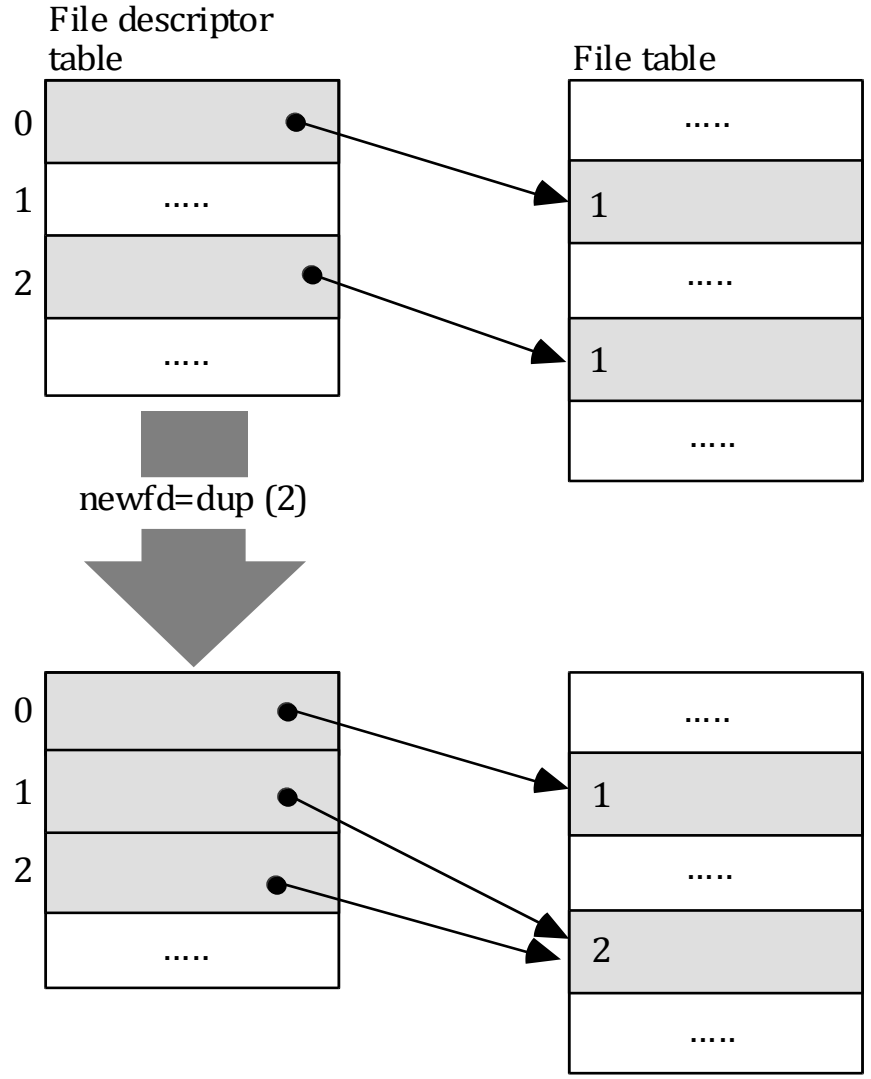
- La chiamata **`write`**
  - se l'estremo di read è aperto
    - i dati in scrittura vengono bufferizzati fino a quando non saranno letti dall'altro processo
  - se l'estremo di read è stato chiuso
    - viene generato un segnale `SIGPIPE`
      - ignorato/catturato: write ritorna **`–1`** e **`errno=EPIPE`**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

Copia del file descriptor - I

- Un file descriptor esistente viene duplicato da una delle seguenti funzioni:
  - **`int dup(int filedes);`**
  - **`int dup2(int filedes, int filedes2);`**
- Entrambe le funzioni “duplicano” un file descriptor, ovvero creano un nuovo file descriptor che punta alla stessa file table entry del file descriptor originario
- Nella file table entry c’e’ un campo che registra il numero di file descriptor che la “puntano”



Copia del file descriptor - II

- Funzione **`dup`**
  - Seleziona il più basso file descriptor libero della tabella dei file descriptor
  - Assegna la nuova file descriptor entry al file descriptor selezionato
  - Ritorna il file descriptor selezionato
- Funzione **`dup2`**
  - Con dup2, specifichiamo il valore del nuovo descrittore come argomento **`filedes2`**
  - Se **`filedes2`** è già aperto, viene chiuso e sostituito con il descrittore duplicato
  - Ritorna il file descriptor selezionato

Utilizzo congiunto di `pipe` e `dup`

- **Problema**: Consideriamo un programma **prog1** che scrive su standard output. Come si può fare in modo che l'output venga visualizzato una pagina alla volta, senza però modificare il programma stesso?
- **Soluzione**: si scrive un altro programma che:
  - crea un pipe e poi genera un processo child mediante **`fork`**
  - nel codice del parent chiude l'estremo di read del pipe e lo stdout, e riassegna mediante **`dup2`** il fd dello **`stdout(1)`** sull'estremo di write del pipe
  - nel codice del child chiude l'estremo di write del pipe e lo stdin, e riassegna mediante **`dup2`** il fd dello **`stdin(0)`** sull'estremo di read del pipe
  - il parent mediante **`exec`** lancia il programma `prog1`
  - il child mediante **`exec`** lancia un programma tipo di paginazione dell'output tipo **`more`** o **`less`**

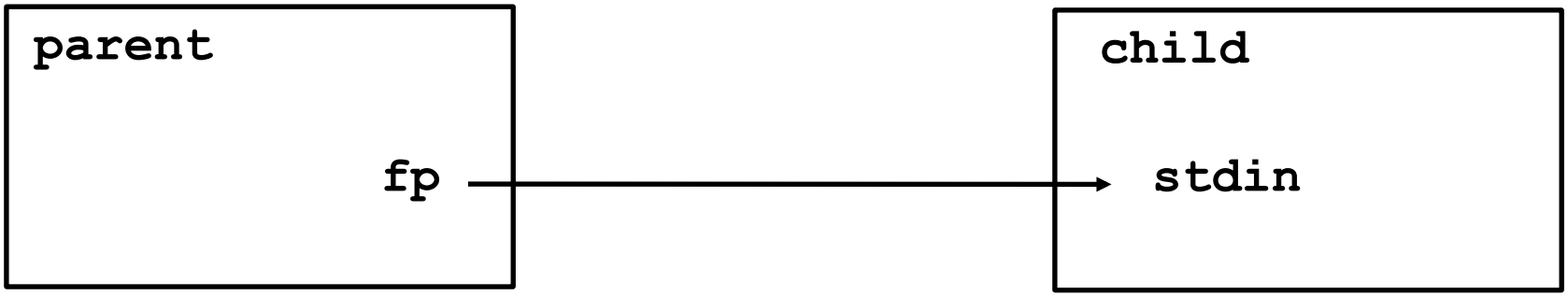
popen - I

`FILE *popen(char *cmdstring, char *type);`

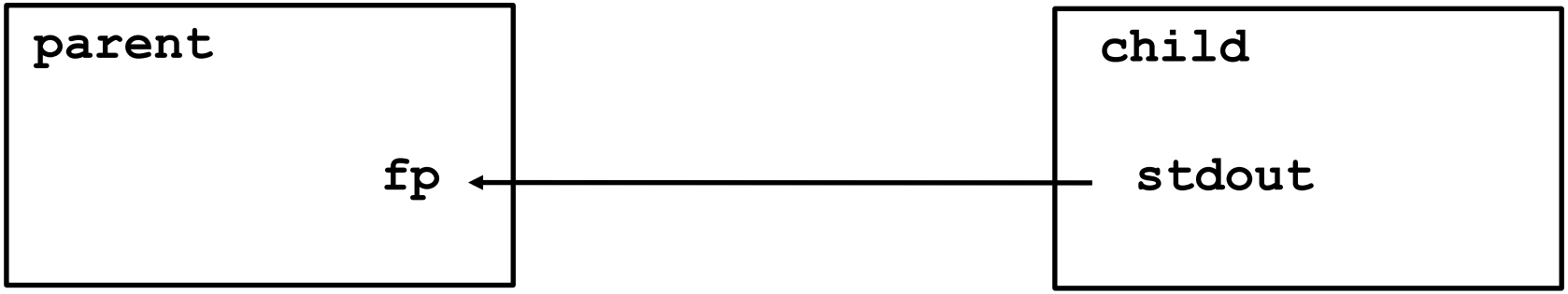
- Descrizione di **`popen`**:
  - crea un pipe
  - crea mediante **`fork`** un processo child
  - chiude gli estremi non utilizzati del pipe
  - esegue mediante `exec` una shell (**`sh -c cmdstring`**) per eseguire il comando **`cmdstring`**
  - ritorna uno standard I/O file pointer:
    - se si specifica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

popen - II

- `type = "w"`



- `type = "r"`



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

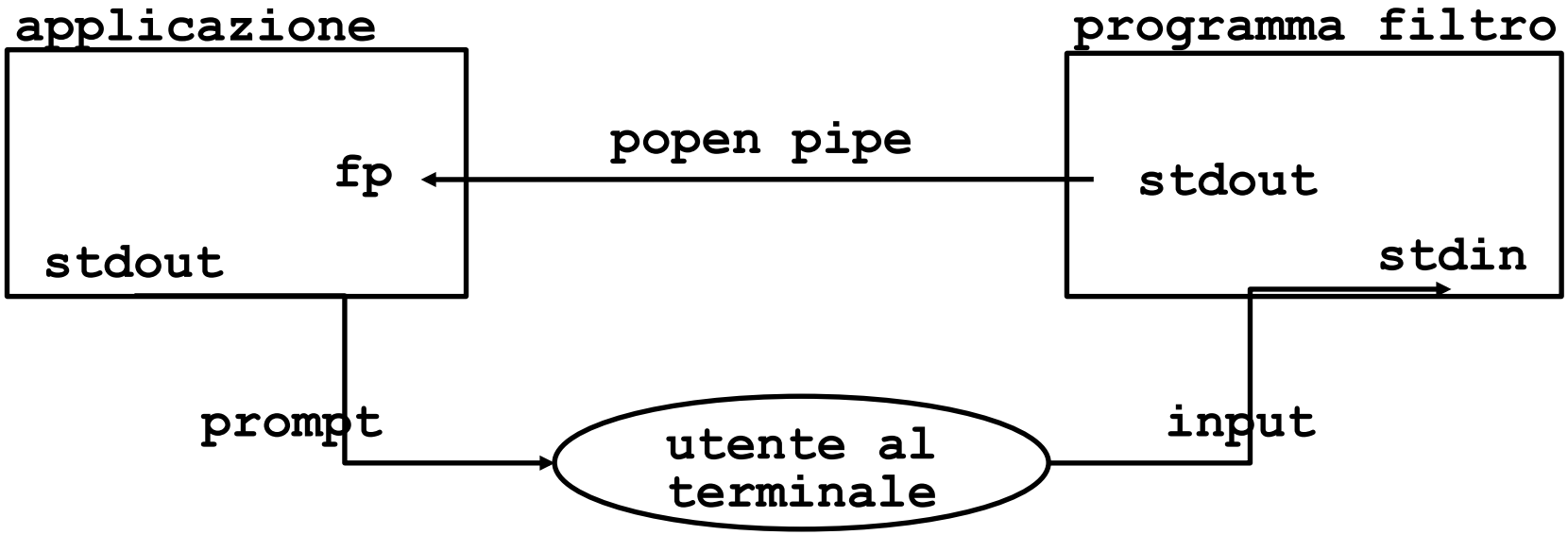
`pclose` - esempio

```
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

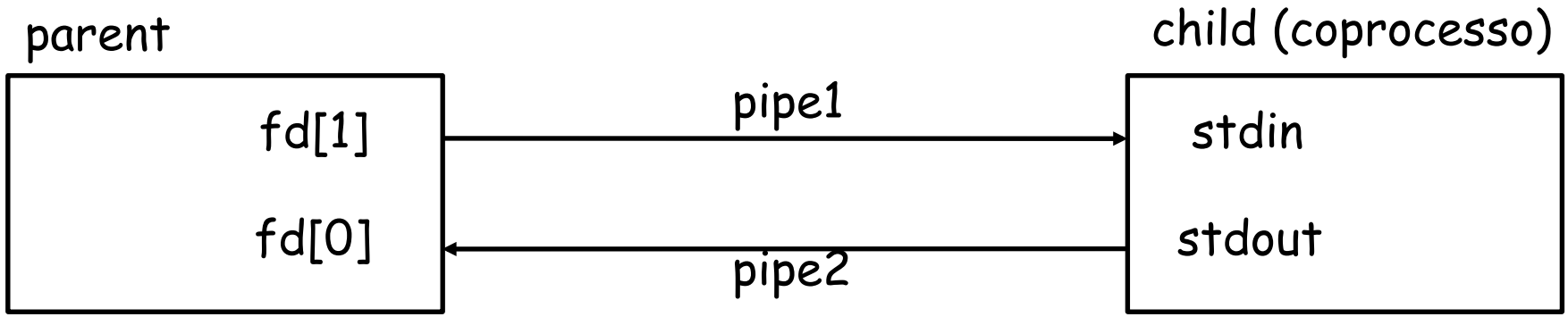
popen - esempio di utilizzo

- 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

Nota: **`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

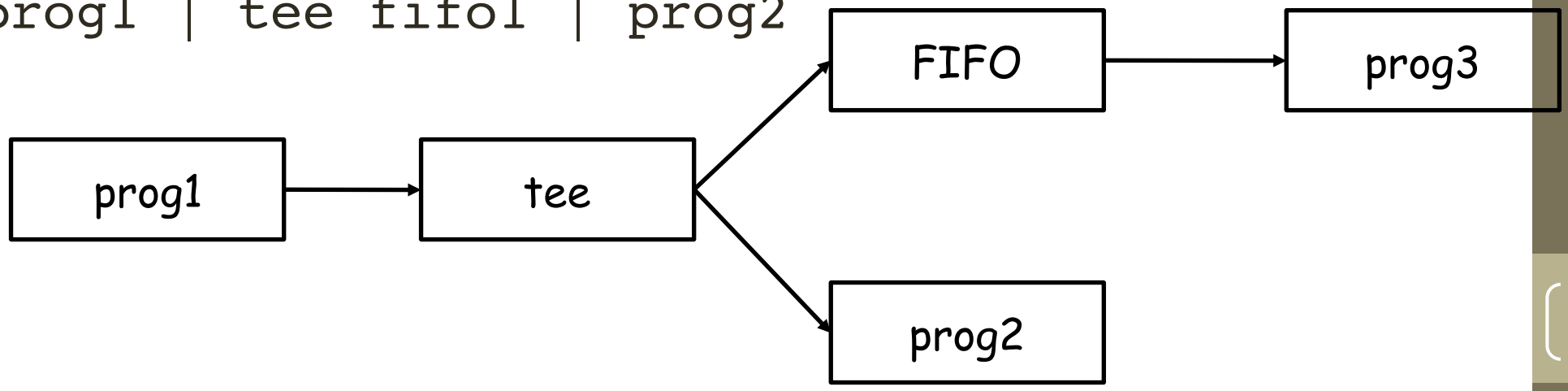
Tabella riassuntiva sull'effetto del flag **`O_NONBLOCK`** su pipe e FIFO

CONDIZIONE	COMPORTAMENTO DI DEFAULT	COMPORTAMENTO CON O_NONBLOCK
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, errno pari a ENXIO
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, dopodiche' scrittura dei dati	ritorno immediato, valore di ritorno pari a 0

FIFO - V

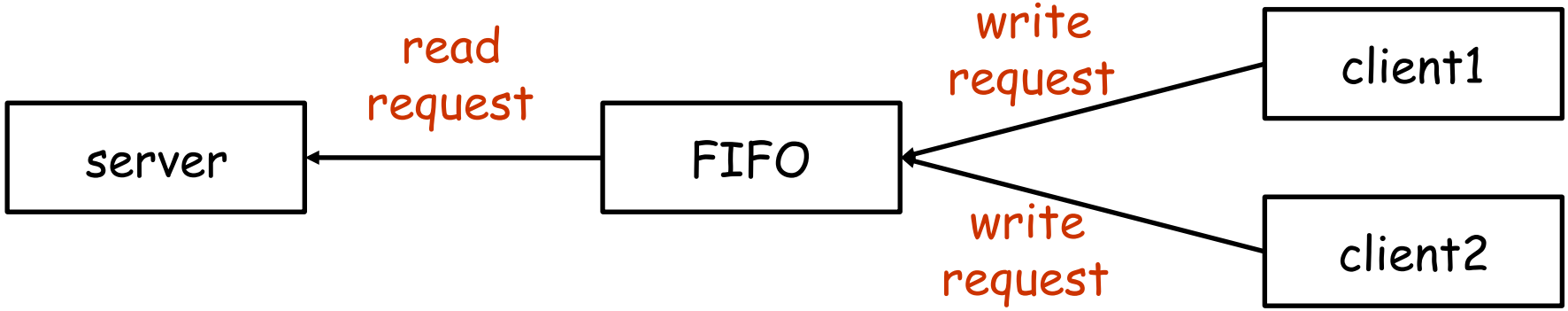
- Utilizzo dei FIFO
  - Utilizzati dai comandi shell per passare dati da una shell pipeline ad un'altra, senza passare creare file intermedi
- Esempio:

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



FIFO - VI

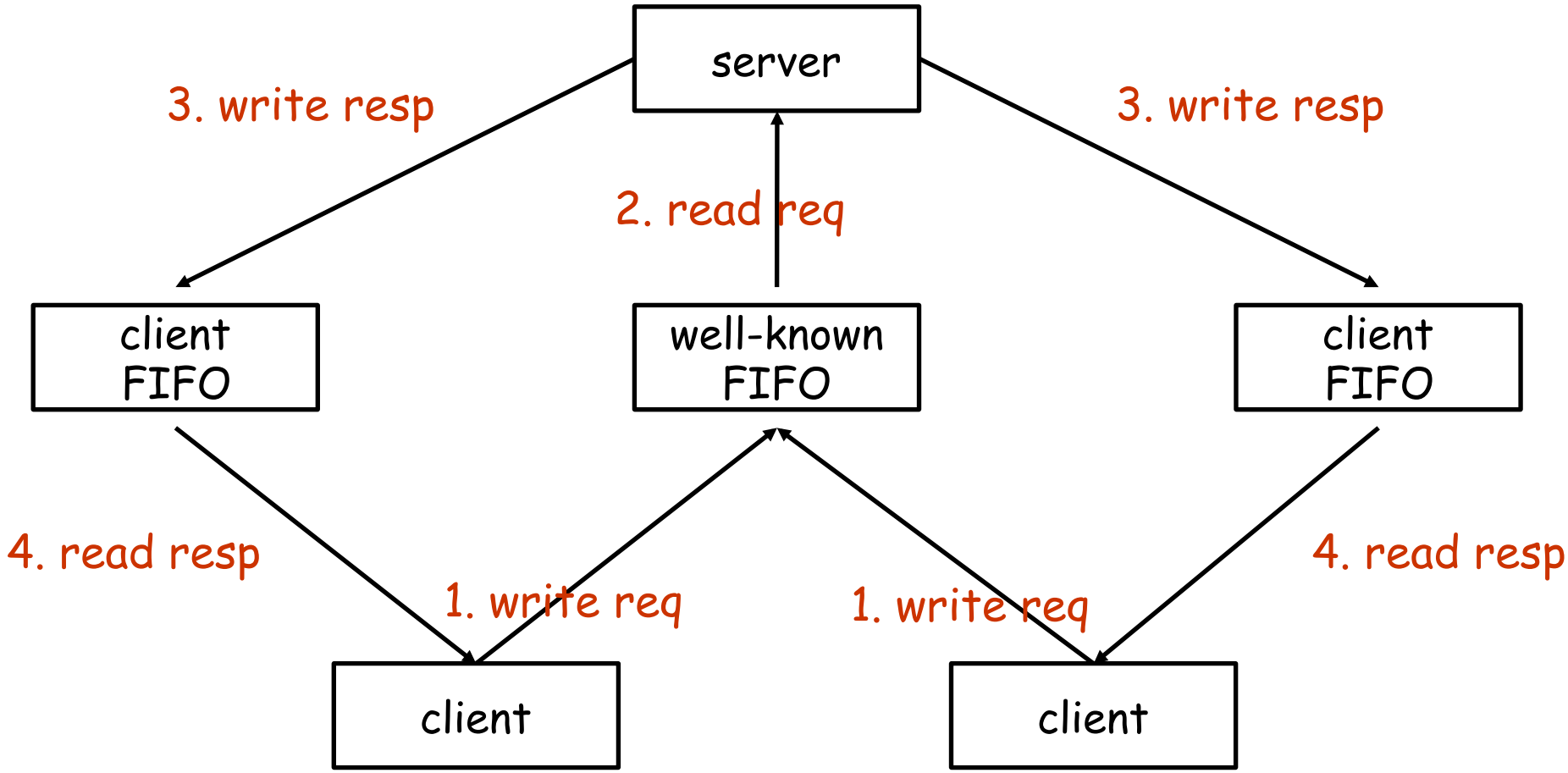
- Utilizzo dei FIFO
  - Utilizzati nelle applicazioni client-server per comunicare
- Esempio:
  - Comunicazioni client → server
    - il server crea un FIFO
    - il pathname di questo FIFO deve essere "well-known" (ovvero, noto a tutti i client)
    - i client scrivono le proprie richieste sul FIFO
    - il server leggere 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 può 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
  - Suggerimenti:
    - 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


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

Esercitazioni

- **Esercizio 1:** scrivere un programma test_fifo.c per verificare i quattro casi possibili di configurazione di una FIFO: read oppure write, con o senza il flag O_NONBLOCK
- **Esercizio 2:** scrivere un programma che esemplifica l'interazione <Produttore-Consumatore>:
  - Utilizzare la named pipe (FIFO) come buffer
  - Il produttore scrive interi sulla pipe e il consumatore li stampa
  - Utilizzare anche più produttori
- **Esercizio 3:** scrivere un programma che estende lo schema di comunicazione della pagina precedente, creando un server dedicato (mediante fork) per ogni client e una FIFO tra client (in write) e server dedicato (in read). Il client scrive sulla FIFO la linea inserita da stdin e il server dedicato la legge da FIFO e la stampa su stdout