Syscall – File I/O

Digressione: man (2) ...

- Le funzioni della libreria standard UNIX associate alle system call di Linux sono documentate nella sezione 2 di man
  - e.g.: man 2 open
- Ogni manpage di system call riporta:
  - header: file .h che devono essere inclusi per avere accesso al prototipo delle funzioni
  - sinopsi: possibili invocazioni C delle funzioni
  - descrizione del funzionamento e dei parametri attuali
  - valore di ritorno
  - errori: valori impostati per la variabile errno
  - conformità

Digressione: strace

- il programma strace è molto utile nel debugging di programmi al livello di astrazione delle system call
- invocazione: *strace <programma> <arg> ...*
  - il programma specificato viene eseguito
  - su stderr vengono stampate tutte le invocazioni di system call effettuate dal processo corrispondente, comprensive di valori di ritorno
  - vengono inoltre stampate le ricezioni di segnali
- riferimenti:
  - man strace

File descriptor

- ad ogni processo è associato un insieme di file aperti (/proc/<pid>/fd/*)
  - il kernel mantiene la tabella dei file aperti associati ad ogni processo
  - i file aperti sono referenziati da numeri naturali detti *file descriptor*
- le shell *nix quando creano nuovi processi aprono di default 3 file, identificati da 3 costanti definite in <unistd.h>
  - standard input: STDIN_FILENO (solitamente 0)
  - standard output: STDOUT_FILENO (solitamente 1)
  - standard errro: STDERR_FILENO (solitamente 2)

open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags /*, mode_t mode */);
```

- utilizzando la syscall open si richiede al kernel di aprire un file assegnandogli un file descriptor
  - pathname: path del file da aprire
  - flags: OR di costanti definite in <fnctl.h>
  - mode: permessi da assegnare al file nel caso venga creato
  - ritorna il file descriptor assegnato dal kernel, o -1 in caso di errori

open

- alcuni flag da <fnctl.h>
  - necessari:
    - O_RDONLY/O_WRONLY/O_RDWR (mutualmente esclusivi), accesso in sola lettura / sola scrittura / entrambi
  - opzionali:
    - O_APPEND scritture alla fine del file
    - O_CREAT crea il file se non esiste
    - O_EXCL fallisce se O_CREAT è specificato ed il file esiste (*poor man's locking*)
    - O_TRUNC se il file esiste e viene aperto in scrittura azzera la sua lunghezza
    - O_SYNC attende il flush dei buffer dopo ogni scrittura

close

```
#include <unistd.h>

int close(int fd);
```

- chiude un file
  - fd: file descriptor
  - ritorna 0 in caso di successo, -1 altrimenti
- la chiusura di un file implica:
  - liberazione del file descriptor
  - rilascio di tutti i record lock acquisiti
  - flush dei buffer

lseek

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

off_t lseek(int fildes, off_t offset, int whence);
```

- ad ogni file aperto è associata la posizione corrente (*current file offset*): un numero naturale di byte dall'inizio del file
  - operazioni di lettura/scrittura sono effettuate a partire dalla posizione corrente e la incrementano del numero di byte letti/scritti
- lseek permette di cambiare la posizione corrente
  - fildes: file descriptor
  - ritorna la nuova posizione corrente, o -1 in caso di errori

lseek

- offset è un intero, la sua semantica dipende da whence
- whence è uno dei seguenti:
  - SEEK_SET la nuova posizione corrisponde ad offset
  - SEEK_CUR la nuova posizione corrisponde a posizione corrente + offset
  - SEEK_END la nuova posizione corrisponde a dimensione del file + offset

- osservazioni:
  - dimensione del file: fseek(fd, 0, SEEK_END)
  - file sparsi: fseek(fd, 1000000, SEEK_END)
  - seek capability: non tutti i file permetto seek (e.g. pipe, socket, fifo), possiamo verificarlo con fseek(fd, 0, SEEK_END)

read

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

- read permette di leggere dati da un file aperto, a partire dalla posizione corrente, muovendosi verso la fine del file
  - fd: file descriptor
  - buf: buffer di destinazione per i dati letti
    - per evitare segfault deve avere dimensione >= count
  - count: numero (massimo) di byte da leggere
  - ritorna il numero di byte letti, o -1 in caso di errori
    - read può ritornare un numero di byte inferiori a quelli richiesti
      - e.g.: fine del file, line-buffering dei terminali, buffering di rete, record-oriented device

write

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

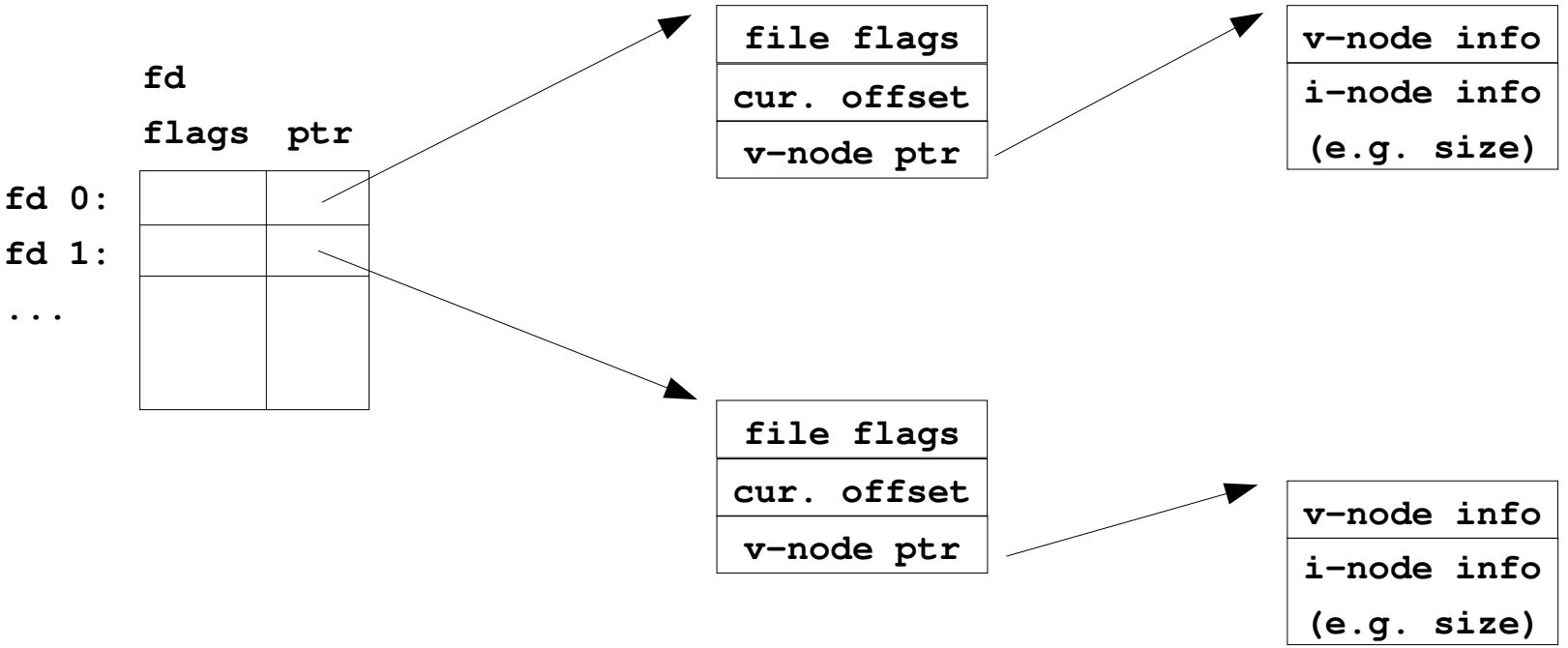
- duale a read: permette di scrivere dati su un file aperto, a partire dalla posizione corrente, verso la fine del file
  - fd: file descriptor
  - buf: buffer di origine per i dati da scrivere
  - count: numero (massimo) di byte da scrivere
  - ritorna il numero di byte scritti, o -1 in caso di errori

File sharing

- i file descriptor visti dal kernel
  - ogni processo ha una entry nella *process table*
  - uno dei campi della process table è il vettore dei file aperti
  - ad ogni file descriptor sono associati:
    - i flag del file descriptor
    - un puntatore ad una entry della *file table*
  - ogni entry nella file table contiene:
    - i flag associati al file (come da open)
    - il *current offset*
    - un puntatore ad una entry della *v-node table*
  - ogni v-node entry contiene:
    - le informazioni contenute nell'i-node del file
    - puntatori a funzioni associate al file

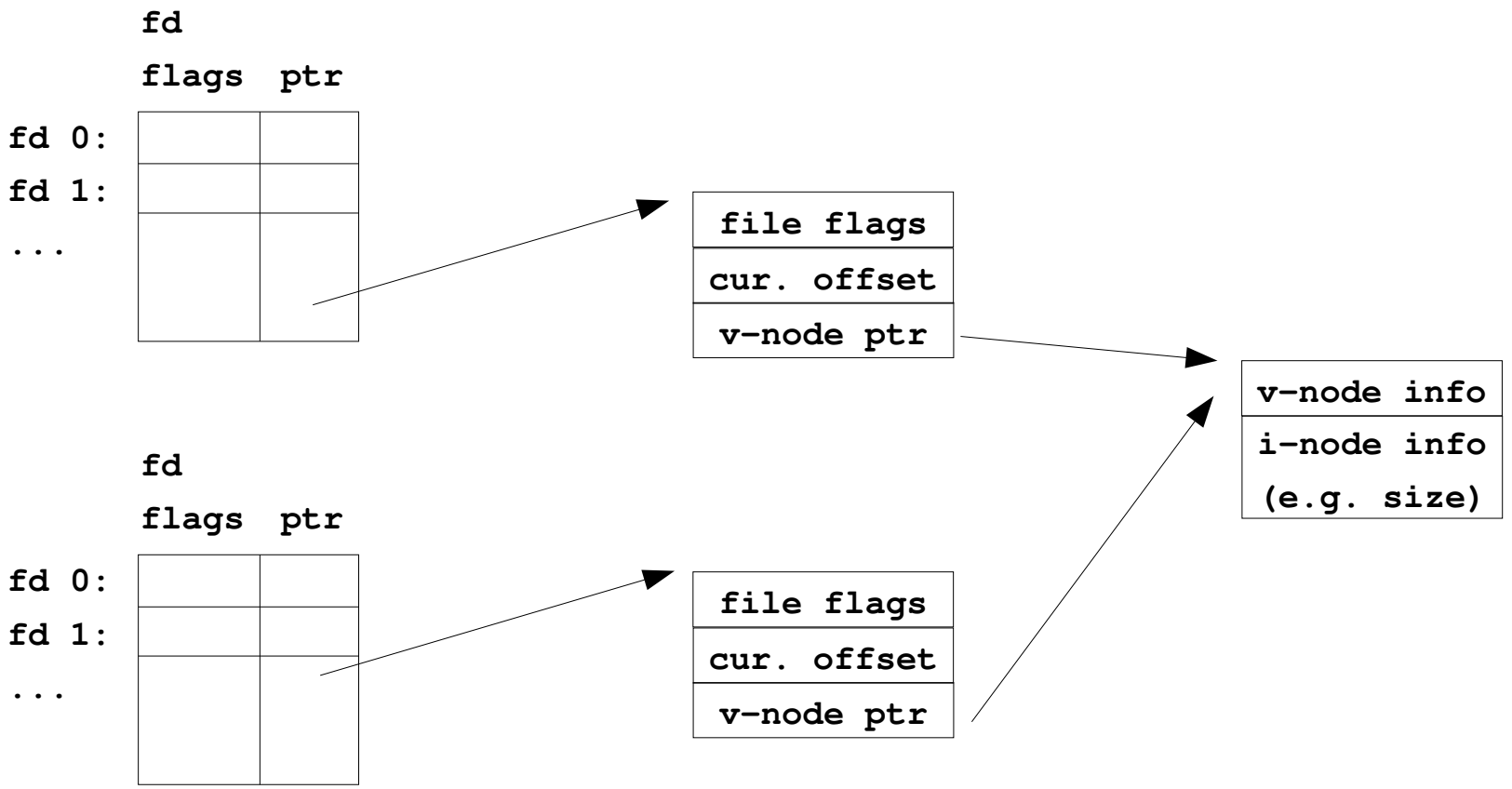
File sharing

process table entry | file table entry | v-note table entry



File sharing

- sharing di un file da parte di due processi
  - ogni processo ha il suo offset
  - la dimensione del file è condivisa



File sharing

- sharing di un file da parte di due processi
  - semantica *nix:
    - dopo ogni write il processo che l'ha effetuata incremente il proprio offset; se il file è aumentato di dimensione, il campo size dell'i-node viene incrementato
    - se O_APPEND è stato specificato, prima di ogni write l'offset viene impostato alla attuale dimensione del file
    - lseek modifica solo l'offset del processo che la invoca
    - lseek SEEK_END posizione il file in base alla dimensione del file nel momento in cui viene invocata
  - esistono due modi per condividere una entry della file table tra processi diversi:
    - O_APPEND (come sopra)
    - fork + le system call dup/dup2

dup

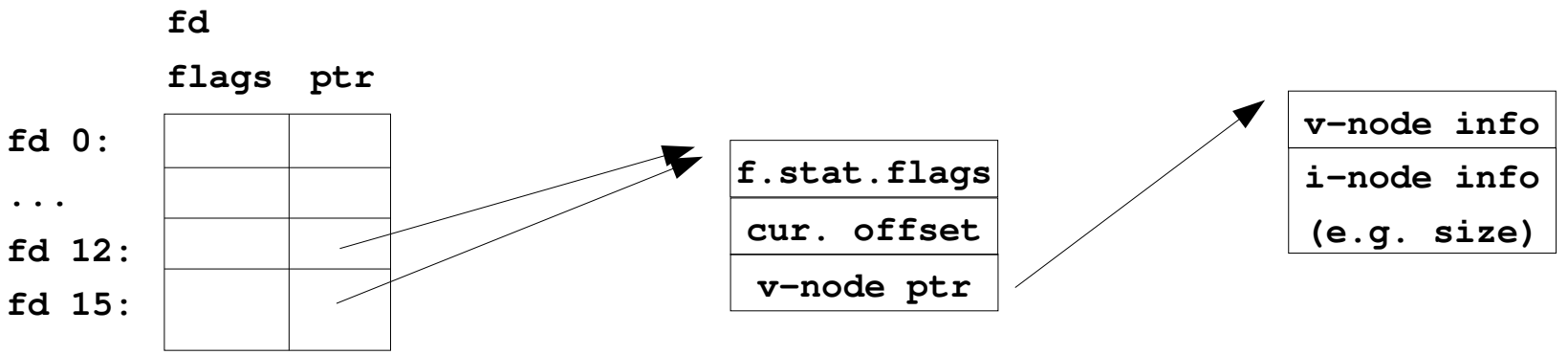
```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

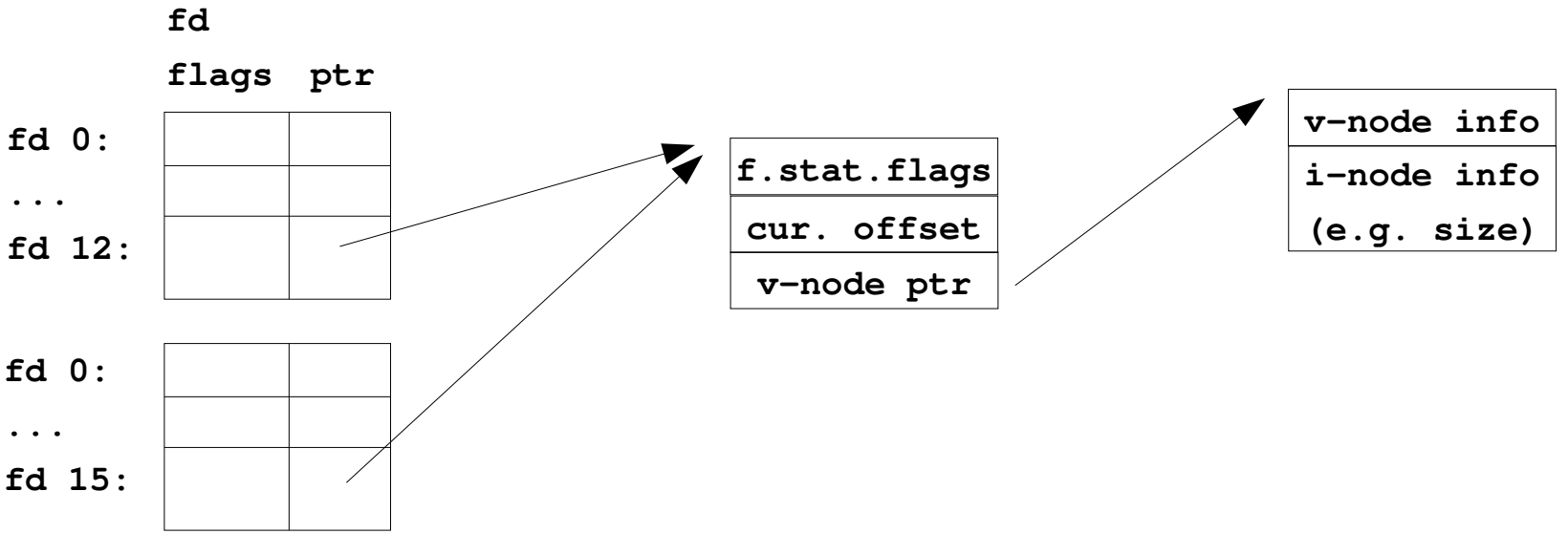
- dup e dup2 duplicano un file descriptor
  - dup sceglie il più piccolo file descriptor disponibile, dup2 permette di specificarlo
  - oldfd: file descriptor che si vuole duplicare
  - ritorna il nuovo file descriptor, o -1 in caso di errori
- invocando fork dopo la duplicazione di un file descriptor è possibile condividere entry della file table tra processi

dup

- dopo l'invocazione di dup



- dopo l'invocazione di dup + fork (+ close appropriate)



fcntl

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd /* , long arg */);
```

- fcntl permette di modificare le proprietà di un file aperto
  - proprietà:
    1. flag associati al file descriptor
    2. flag associati al file status
    3. processo gestore dell'I/O asincrono
    4. record locking
    5. (duplicazione di file descriptor)
  - fd: file descriptor
  - cmd: comando da eseguire
  - il valore di ritorno dipende da cmd, -1 in caso di errori

Miscellanea

- altre system call correlate alla gestione dei file
  - stat ritorna informazioni relative ad un file
  - umask modifica la maschera di bit per la creazione di file
  - chmod modifica i permessi associati ad un file
  - chown modifica l'owner di un file
  - link crea link (fisici)
  - unlink rimuove un file
  - symlink crea link simbolici
  - readlink legge la destinazione di un link simbolico
  - utime modifica i tempi di accesso e modifica di un file
  - mkdir/rmdir/opendir/readdir/rewinddir/closedir/chdir
    gestione delle directory
  - sync flush dei buffer del kernel

Libreria I/O standard

- ANSI C definisce una libreria standard di I/O, i cui prototipi sono descritti in <stdio.h>
  - la libreria standard:
    - offre un layer d'astrazione sul sistema operativo, può essere implementata anche in sistemi operativi non *nix
    - offre inoltre primitive di più alto livello, e.g.:
      - lettura/scrittura linea per linea
      - gestione automaticamente del buffering
      - formattazione di input/output
  - le syscall di I/O
    - sono implementata solo su sitemi *nix
    - sono spesso più efficienti della libreria standard che deve effettuare copie tra i suoi buffer e quelli del kernel

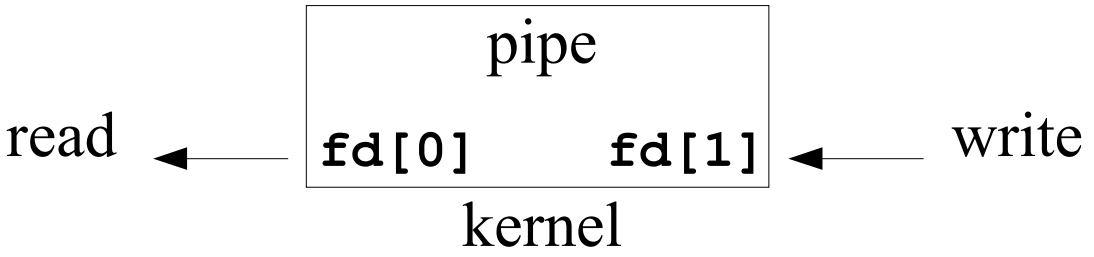
Riferimenti

- APUE
  - file I/O: cap. 3
  - file e directory: cap. 4

Syscall – Basic IPC

pipe

- sono le più supportate strutture di InterProcess Communication (IPC) su sistemi *nix
- caratteristiche:
  - half-duplex
  - possono essere utilizzate solo tra processi che hanno un antenato comune nella gerarchia dei processi
- una pipe è formata da una coppia di file descriptor, il primo aperto in sola lettura, il secondo in sola scrittura; tutto ciò che viene scritto su quest'ultimo può essere letto dal primo



pipe

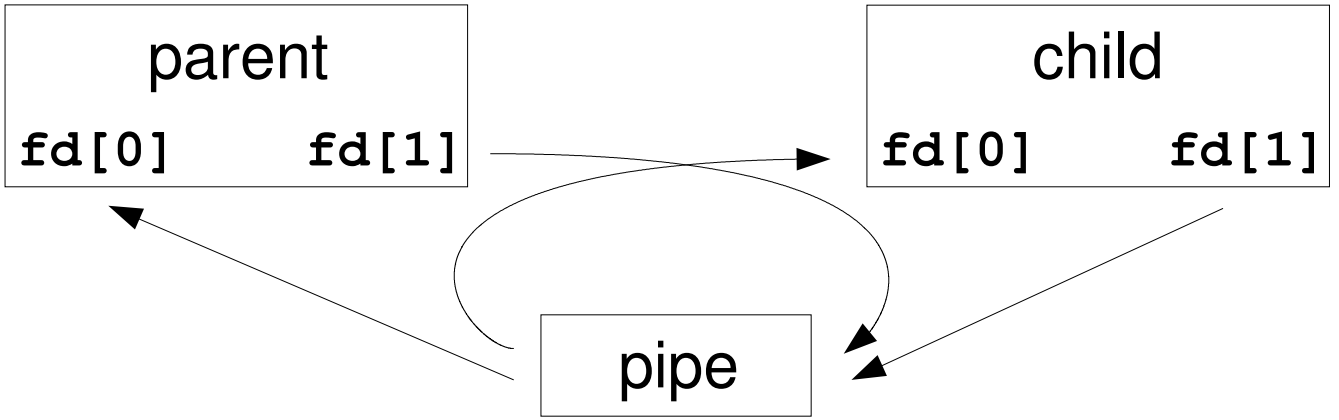
```
#include <unistd.h>

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

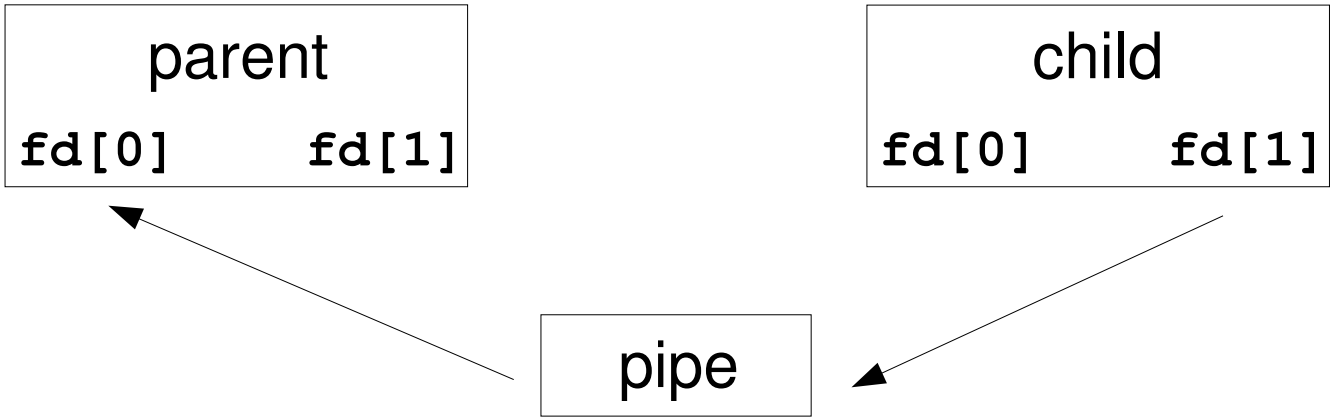
- la system call pipe permette di richiedere al kernel la creazione di una pipe
  - filedes: array di due file descriptor, viene riempito dalla syscall
    - fd[0] è aperto in lettura, fd[1] in scrittura
  - ritorna 0 in caso di successo, -1 altrimenti
- utilizzo tipico di pipe:
  1. creazione di una pipe
  2. fork
  3. uno dei due processi chiude fd[0], l'altro fd[1]

pipe

- scenario dopo l'invocazione di pipe + fork



- scenario dopo l'invocazione di pipe + fork + close
  (assumiamo sia il figlio a voler scrivere al padre)



pipe – esempio

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

int main(void)
{
       int    n, fd[2];
       pid_t  pid;
       char   line[1024];
       if (pipe(fd) < 0) {
           fprintf(stderr, "pipe error\n");
           exit(1);
   }
   if ( (pid = fork()) < 0) {
           fprintf(stderr, "fork error\n");
           exit(1);
   } 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, 1024);
           write(STDOUT_FILENO, line, n);
       }
       exit(0);
}
```

popen / pclose

```
#include <stdio.h>

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

- la libreria standard offre le funzioni popen/pclose per semplificare l'uso di pipe
  - popen = pipe + fork + exec + close
    - command: patch del comando da eseguire
    - type: “r” per apire una pipe in lettura, “w” in scrittura
    - ritorna FILE pointer in caso di successo, NULL altrimenti
  - pclose chiude lo stream ed attende la terminazione
    - stream: stream da chiudere
    - ritorna stato di terminazione del proesso, -1 in caso di errori

mkfifo

```
#include <sys/types.h>
#include <sys/stat.h>

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

- le FIFO (o *named pipe*) sono concettualmente simili alle pipe, ma:
  - esistono sul filesystem
  - permettono comunicazione tra processi arbitrari
- la syscall mkfifo permette di creare FIFO sul filesystem
  - pathname: percorso della FIFO sul filesystem
  - mode: flag come per open
  - ritorna 0 in caso di successo, -1 altrimenti

mkfifo

- dopo la creazione le FIFO vengono utilizzate come file ordinari (open, write, close, ...)
- la semantica dell'utilizzo delle FIFO dipende dal flag O_NONBLOCK
  - nel caso in cui O_NONBLOCK non sia specificato
    - invocazioni di open in lettura restano bloccate fino a quando un altro processo non apre la FIFO in scrittura, e viceversa
  - nel caso in cui sia specificato
    - invocazioni di open in lettura ritornano immediatamente
    - invocazioni di open in scrittura falliscono con errno ENXIO se nessun processo ha ancora aperto la FIFO in lettura

mkfifo

- rilevanza delle FIFO
  - shell scripting: comunicazione tra programmi che non compongono la stessa pipeline, senza ricorrere a file temporanei
    - il comando mkfifo (man **1** mkfifo) permette di creare FIFO da linea di comando
  - programmi client server eseguiti su uno stesso host

- riferimenti
  - APUE, cap. 14, Interprocess Communication

Syscall – Daemon programming

Relazioni tra processi

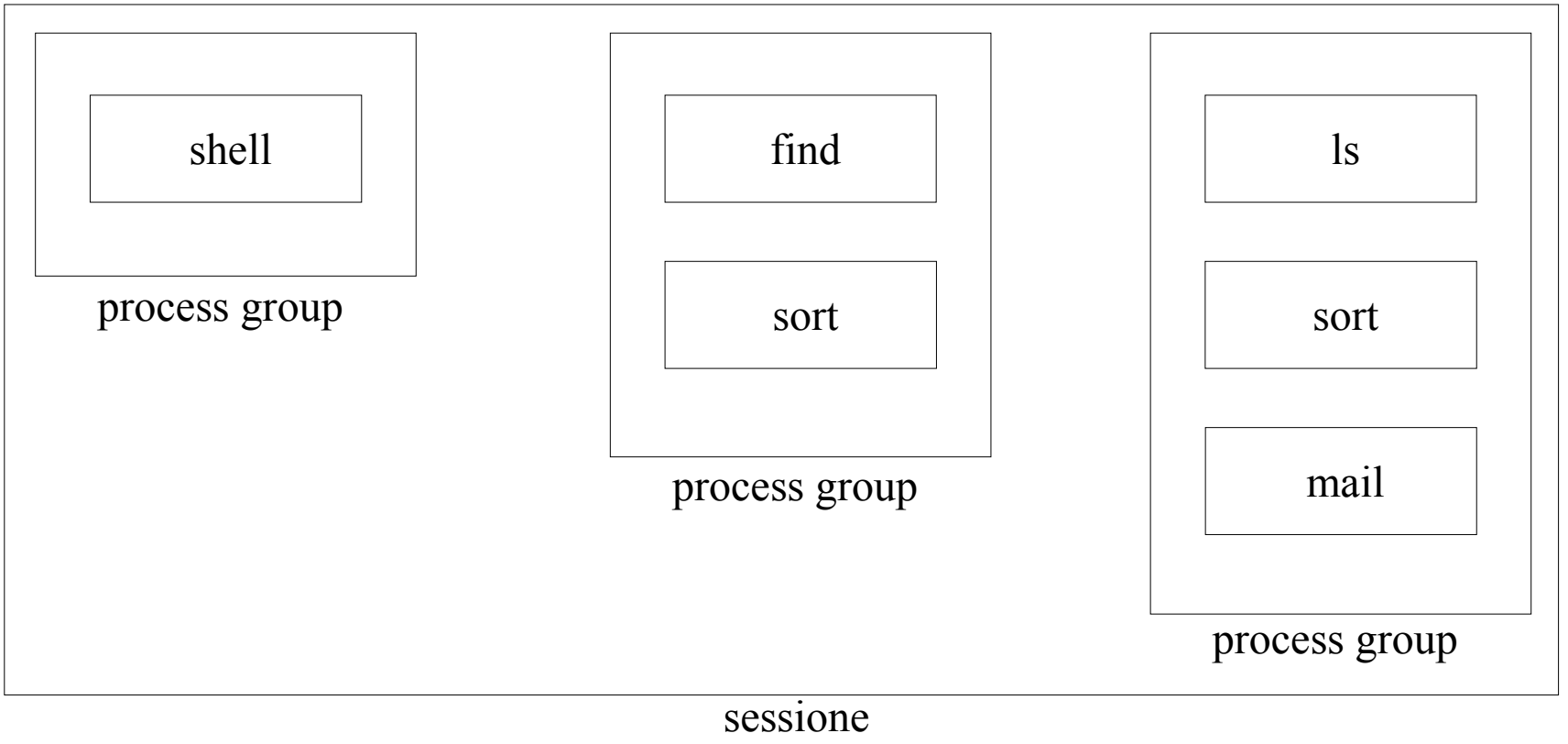
- abbiamo visto che i processi sono collegati tra loro da relazioni di discendenza padre-figlio, esistono altri tipi di relazioni tra processi
- ogni processo appartiente ad un *process group*
  - un process group è un insieme di processi
  - ad ogni process group è assegnato un PGID numerico
  - ogni process group può avere un *group leader*
  - esempi di utilità di process group sono la consegna dei segnali (già vista)
- ogni process group appartiene ad una *sessione*
  - ogni sessione può essere associate ad un *terminale*
  - il processo che crea una sessione è detto *session leader*

Relazioni tra processi – esempio

- eseguiamo con una shell

```
find /etc -type f  |  sort >  ~/conffiles.txt  &
ls ~/  |  sort  |  mail foo@test.com
```

- la sessione risultante sarà la seguente:



Relazioni tra processi – syscall

- system call relative alle relazioni tra processi
  - getpgrp ritorna il process group di appartenenza del processo corrente
  - setpgid entra a far parte di un gruppo di processi o ne crea uno nuovo
  - setsid crea una nuova sessione (senza terminale)

Creazione di un “demone”

- abbiamo visto esempi di processi “demoni” (e.g. cron)
- per programmarli occorre seguire una sequenza di passi
  1. fork (il padre termina, il figlio continua)
     - la shell attende solo il padre
     - assicura che il figlio non sia process group leader
  2. creare una nuova sessione
     - il processo diventa session e process group leader, nessun terminale associato
  3. impostare la directory corrente a “/”
  4. impostare umask a 0
  5. chiudere i file descriptor non necessari

Creazione di un “demone”

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

int main(void)
{
     pid_t pid;
     if ( (pid = fork()) < 0)
        exit(-1);
     else if (pid != 0)
        exit(0);   /* parent exits */
     /* child */
     setsid();     /* new session */
     chdir("/");   /* go to / */
     umask(0);     /* reset umask */
     /* real code here */
     exit(0);
}
```

Riferimenti

- relazioni tra processi
  - APUE, cap. 9
- programmazione di demoni
  - APUE, cap. 13