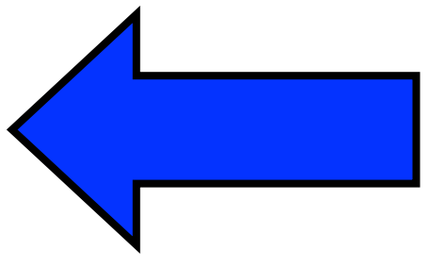
laboratorio di sistemi operativi

Pipes and FIFOs

Marco Botta
Materiale preparato da Daniele Radicioni

argomenti del laboratorio UNIX

1. introduzione a UNIX;
2. integrazione C: operatori bitwise, precedenze, preprocessore, pacchettizzazione del codice, compilazione condizionale e utility make;
3. controllo dei processi;
4. segnali;
5. pipe e fifo;
6. code di messaggi;
7. memoria condivisa;
8. semafori;
9. introduzione alla programmazione bash.

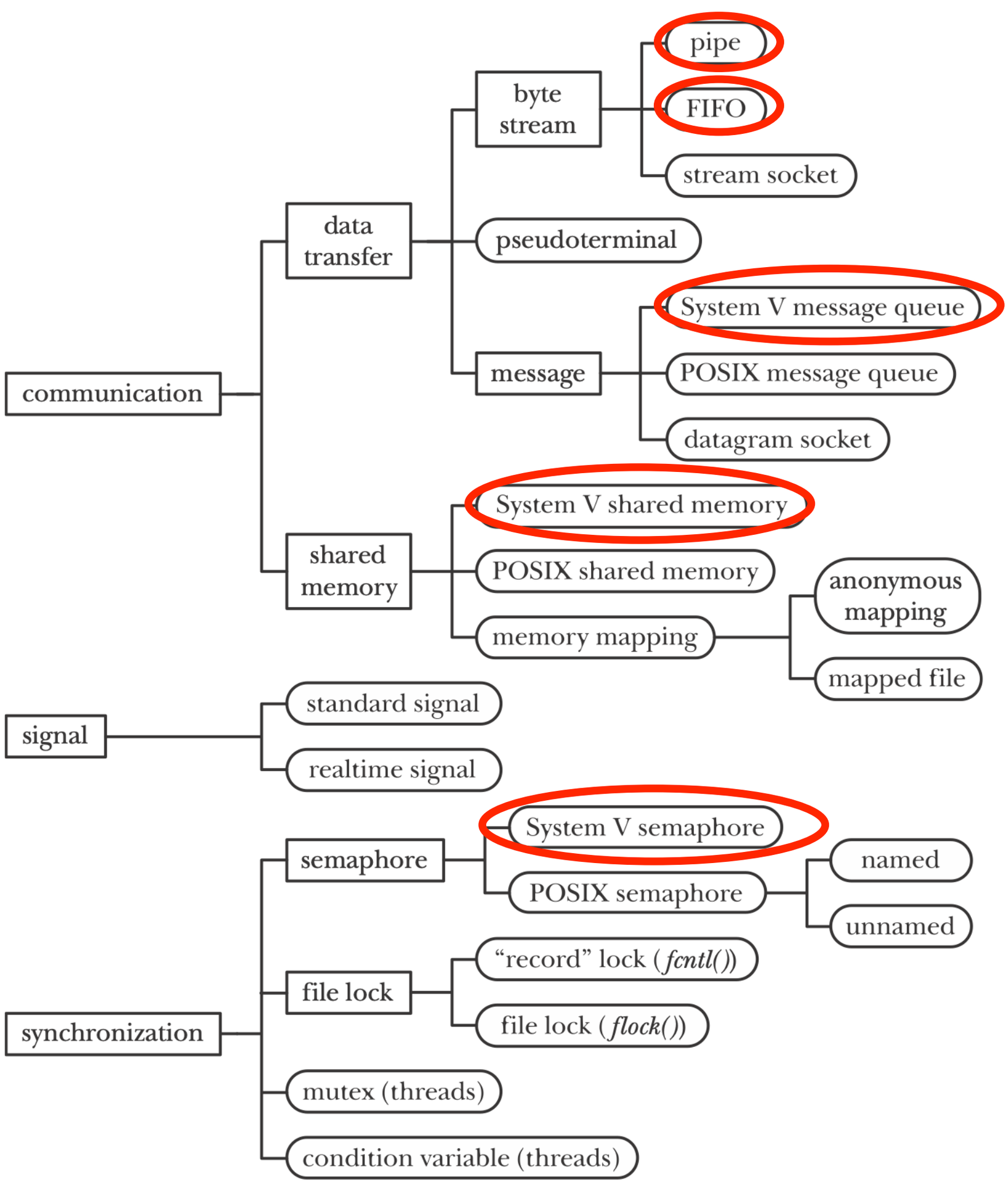


- il materiale di questa lezione è tratto prevalentemente dai testi:
  - Michael Kerrisk, ***The Linux Programming interface - a Linux and UNIX® System Programming Handbook***, No Starch Press, San Francisco, CA, 2010.
  - W. Richard Stevens (Author), Stephen A. Rago, ***Advanced Programming in the UNIX® Environment*** (2nd Edition), Addison-Wesley, 2005.

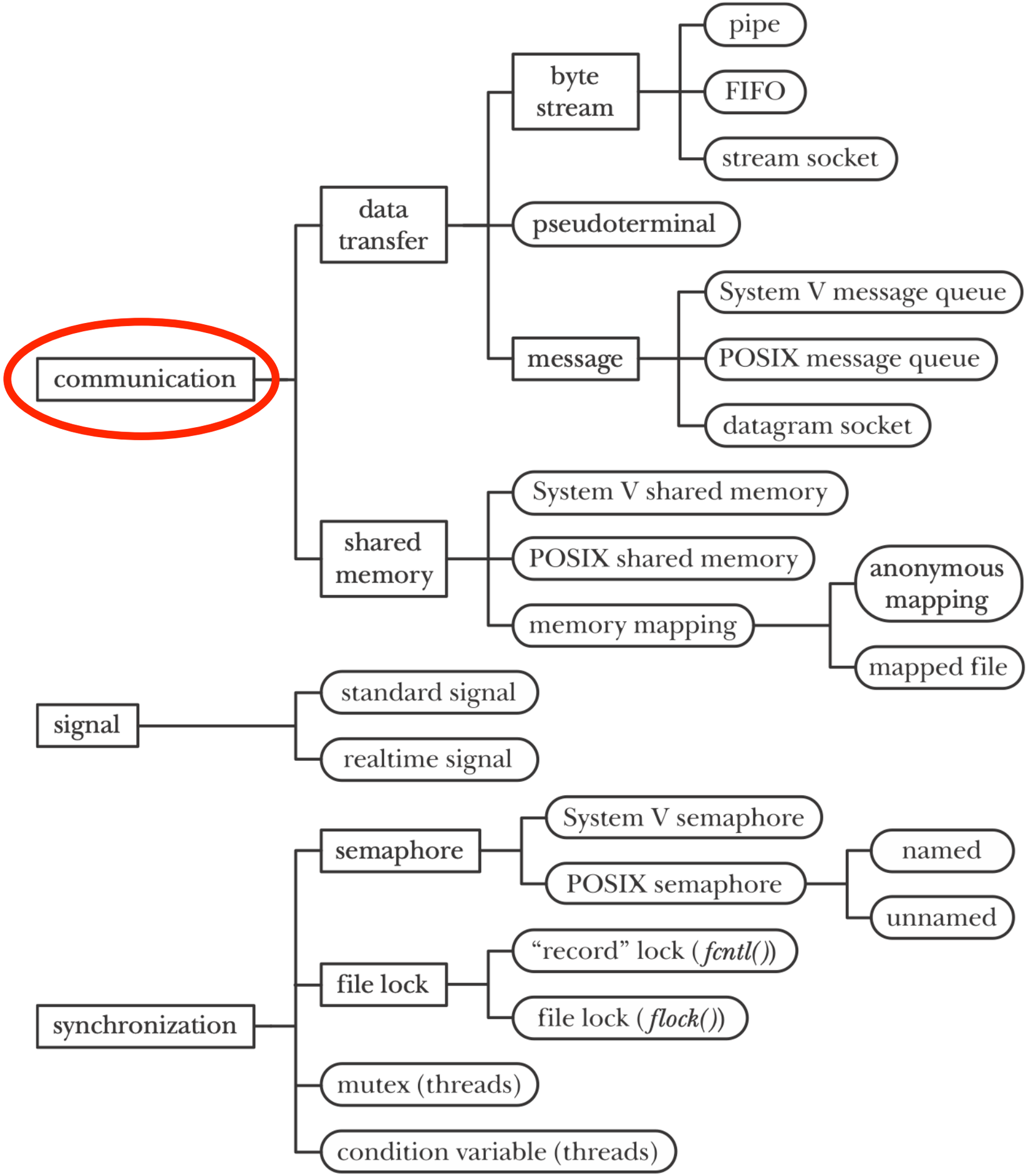
intro IPC

IPC facilities

- i vari strumenti che UNIX offre per la comunicazione e la sincronizzazione possono essere suddivisi in tre ampie categorie funzionali:
  - ***Comunicazione***: facilities utilizzate per lo scambio di dati fra processi.
  - ***Sincronizzazione:*** facilities utilizzate per sincronizzare le azioni dei processi.
  - ***Segnali***: sebbene i segnali siano nati prevalentemente con altri fini, in alcune circostanze possono essere utilizzati come strumenti di sincronizzazione.


communication
data transfer
byte stream
pipe
FIFO
stream socket
pseudoterminal
message
System V message queue
POSIX message queue
datagram socket
shared memory
System V shared memory
POSIX shared memory
memory mapping
anonymous mapping
mapped file
signal
standard signal
realtime signal
synchronization
semaphore
System V semaphore
POSIX semaphore
named
unnamed
file lock
"record" lock (fcntl())
file lock (flock())
mutex (threads)
condition variable (threads)

Communication Facilities


communication
data transfer
byte stream
pipe
FIFO
stream socket
pseudoterminal
message
System V message queue
POSIX message queue
datagram socket
shared memory
System V shared memory
POSIX shared memory
memory mapping
anonymous mapping
mapped file
signal
standard signal
realtime signal
synchronization
semaphore
System V semaphore
POSIX semaphore
named
unnamed
file lock
"record" lock (fcntl())
file lock (flock())
mutex (threads)
condition variable (threads)

Communication Facilities

- *Data-transfer facilities*: l'elemento fondamentale che distingue questi strumenti è la nozione di scrittura e lettura.
  - per comunicare, un processo scrive i dati alla facility per l'IPC e un altro processo legge questi dati.
  - questi strumenti richiedono due trasferimenti dati fra la memoria utente e quella del kernel: un trasferimento durante la scrittura e un trasferimento durante la lettura.

Communication Facilities

- *Memoria condivisa*: la memoria condivisa permette ai processi di scambiarsi le informazioni mettendole in una regione della memoria condivisa fra i processi.
  - un processo può rendere i dati disponibili per gli altri processi collocandoli in una regione di memoria condivisa.
  - poiché la comunicazione non richiede system call o trasferimento di dati fra la memoria utente e quella del kernel, la memoria condivisa è uno strumento di comunicazione molto veloce.

Data-transfer facilities: byte streams

- Le data-transfer facilities possono essere ulteriormente suddivise nelle seguenti sottocategorie:
- Byte stream: i dati scambiati per mezzo di pipe, FIFOs, e datagram sockets sono uno stream di byte.
  - ogni operazione di lettura può leggere un numero arbitrario di byte, senza considerare la dimensione dei blocchi scritti dallo scrivente.
  - questo modello riflette il tradizionale modello di UNIX in cui il file è visto come una sequenza di byte.

Data-transfer facilities: messaggi

- Le data-transfer facilities possono essere ulteriormente suddivise nelle seguenti sottocategorie:
- Messaggio: i dati scambiati con le code di messaggi, e i socket hanno la forma di messaggi delimitati.
  - ogni operazione di lettura legge un intero messaggio, così come scritto dal processo scrivente.
  - non è possibile leggere parzialmente un messaggio, lasciando il resto sulla IPC facility, e non è possibile leggere molteplici messaggi con una singola operazione di lettura.

Data-transfer facilities

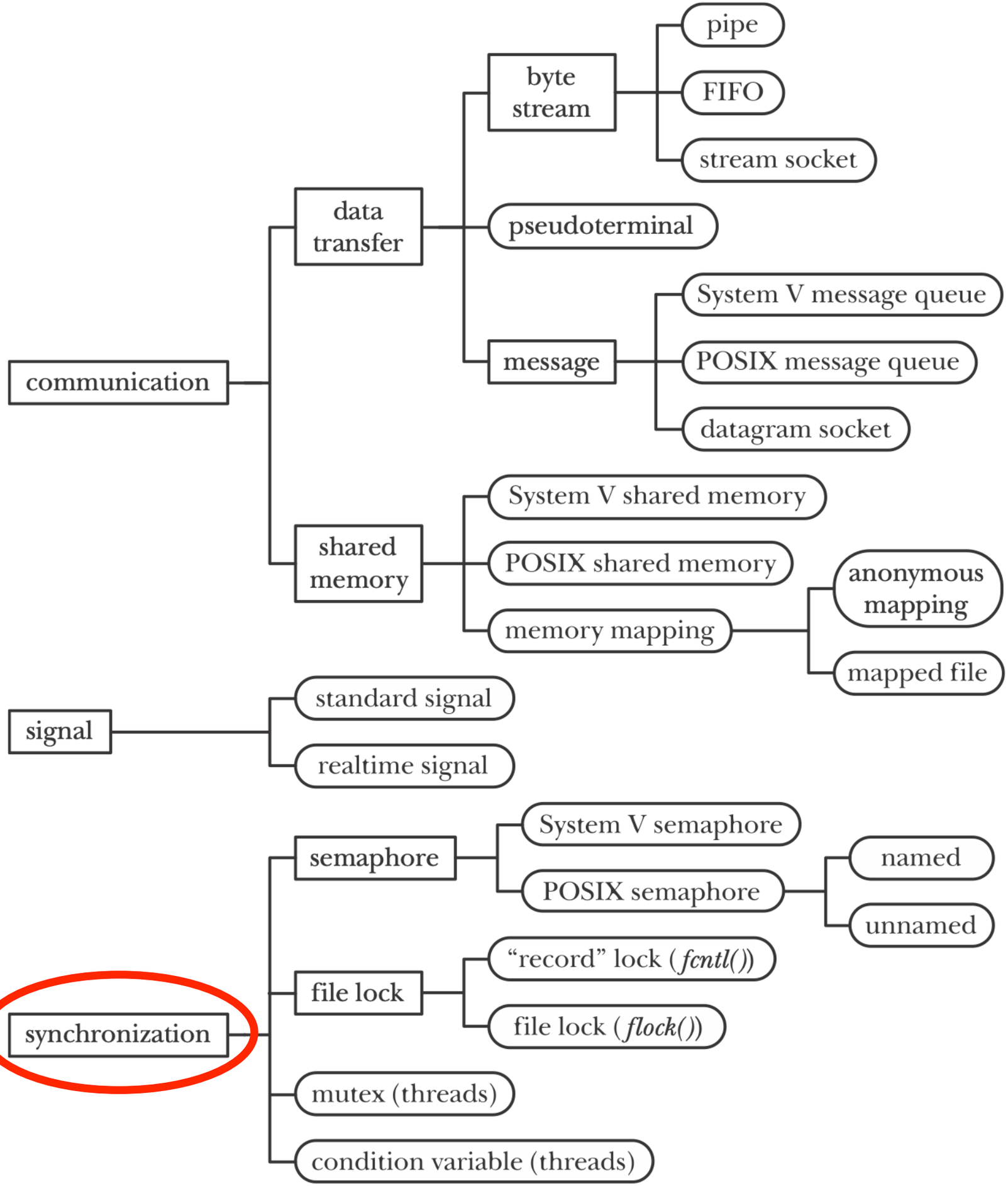
- Le data-transfer facilities sono distinte dalla memoria condivisa per alcune caratteristiche generali:
  - Sebbene le data-transfer facilities possano avere molteplici lettori, le operazioni di lettura sono distruttive. Una operazione ***read*** consuma i dati, e i dati non sono più disponibili per altri processi.
  - La sincronizzazione fra processo lettore e scrittore è automatica. Se un lettore tenta di consumare dati da una facility che attualmente non ne contiene, (di default) l'operazione di lettura si bloccherà finché un processo non avrà scritto dei dati su quella facility.

Shared memory

- Sebbene la memoria condivisa fornisca una comunicazione veloce, questo vantaggio è bilanciato dalla necessità di sincronizzare le operazioni sulla memoria condivisa.
  - per esempio, un processo non dovrebbe cercare di accedere a una struttura dati presente nella memoria condivisa mentre un altro processo la sta modificando.
- il ***semaforo*** è lo strumento di sincronizzazione abitualmente utilizzato con la memoria condivisa.
  - i dati presenti nella memoria condivisa sono visibili a tutti i processi che condividono quel segmento di memoria, diversamente dalla semantica distruttiva delle operazioni di lettura messe a disposizione dalle data-transfer facilities.

Synchronization Facilities

Michael Kerrisk, *The Linux Programming interface - a Linux and UNIX® System Programming Handbook*, No Starch Press, San Francisco, CA, 2010.



Semafori

- Semafori: un semaforo è un intero mantenuto dal kernel, il cui valore non può divenire minore di ***0***.
  - Un processo può decrementare o incrementare il valore di un semaforo. Se viene fatto un tentativo di decrementare il valore di un semaforo sotto lo ***0***, il kernel blocca l'operazione finché il valore del semaforo aumenta a un livello che permette di eseguire l'operazione.
  - In alternativa, il processo può richiedere una ***nonblocking operation***; in questo caso, invece di bloccarlo, il kernel provoca una restituzione immediata con un errore che indica che l'operazione non ha potuto essere eseguita immediatamente.

pipes

pipe

- I pipe forniscono una soluzione a un problema frequente: avendo creato due processi per eseguire programmi diversi (***comandi***), come può la shell fare in modo che l'output prodotto da un processo sia utilizzato come input per l'altro processo?

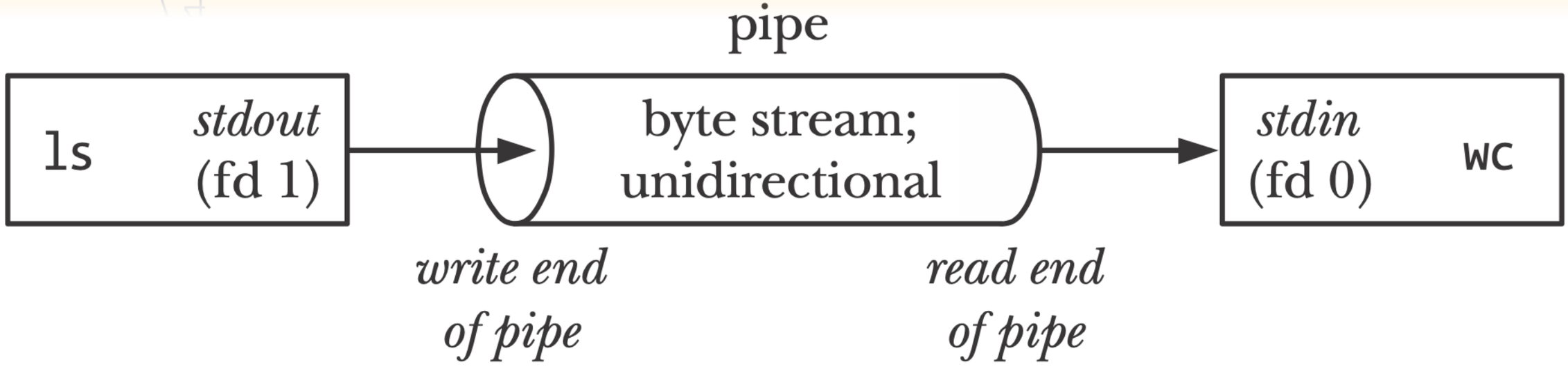
- I FIFO sono una variazione del concetto di pipe. La differenza sostanziale è che i FIFO possono essere utilizzati per la comunicazione fra processi qualsiasi.

Pipes

```
$ ls -al | wc -l
      74
```

- per eseguire questi i comandi, la shell crea due processi, che eseguono ***ls*** e ***wc***, rispettivamente.
  - questo è fatto utilizzando la ***fork()*** e la ***exec()***.

```
$ ls -al | wc -l
      74
```



- i due processi cono collegati al pipe: il processo scrittore (***ls***) ha il proprio standard out (***file descriptor 1***) collegato con il write end del pipe, mentre il processo lettore (***wc***) ha il proprio standard input (***file descriptor 0***) collegato al read end del pipe.
  - NB: i due processi sono completamente ignari dell'esistenza del pipe: semplicemente leggono e scrivono da/su descrittori di file standard.

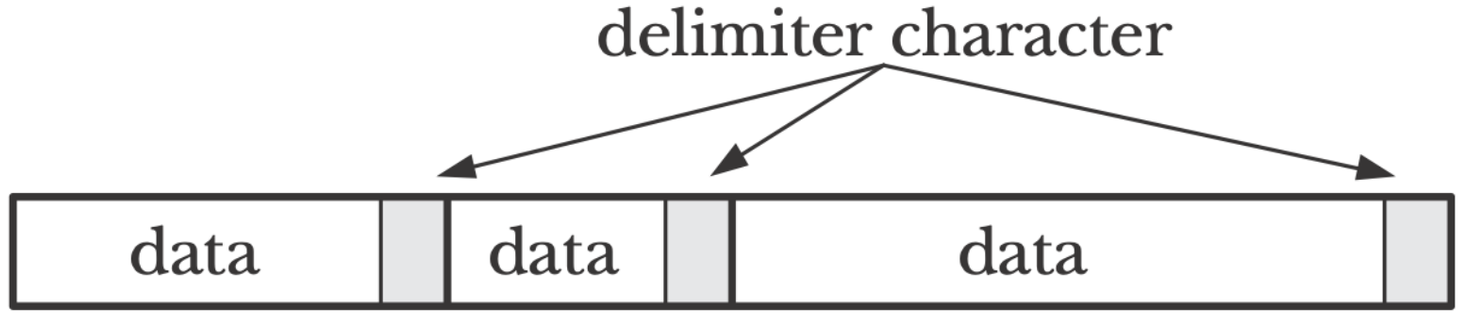
i pipe sono stream di byte

- un pipe è uno stream di byte: usando un pipe, non facciamo riferimento ad alcun concetto di messaggio o di delimitazione di messaggio.
  - il processo che legge da un pipe può leggere blocchi di qualsiasi dimensione, indipendentemente dalla dimensione dei blocchi scritti dal processo che scrive.

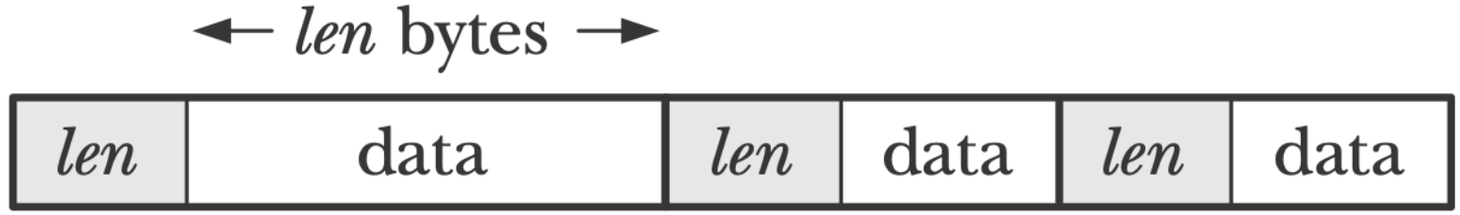
- i dati passano attraverso il pipe in sequenza: i byte sono letti nello stesso ordine in cui sono stati scritti. non è possibile accedere ai dati in maniera casuale utilizzando ***lseek()***.

protocol

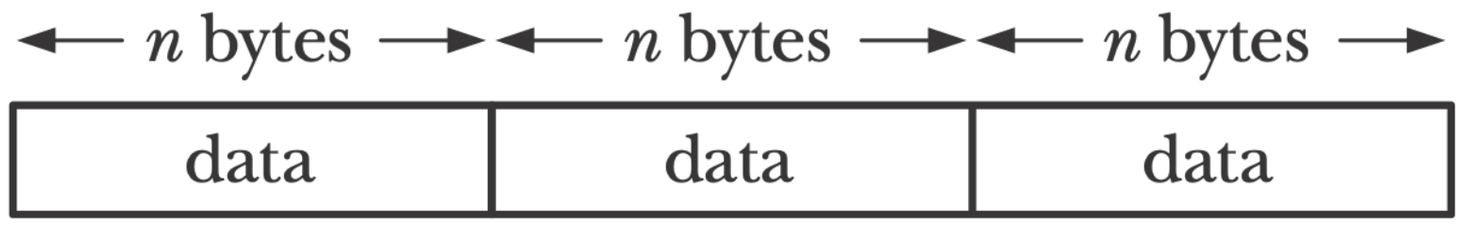
delimiter character



header with length field



fixed-length messages



Michael Kerrisk, *The Linux Programming interface - a Linux and UNIX® System Programming Handbook*, No Starch Press, San Francisco, CA, 2010.

lettura da pipe

- I tentativi di leggere da un pipe vuoto restano bloccati finché almeno un byte non è stato scritto sul pipe.
- Se il write end di un pipe viene chiuso, un processo che legge dal pipe riceverà il codice ***end-of-file*** (i.e., ***read()*** restituirà ***0***) una volta che avrà letto tutti i dati presenti nel pipe.
- I pipe sono unidirezionali. I dati possono viaggiare solo in una direzione.
  - Un'estremità (***end***) del pipe è utilizzato in scrittura, e l'altro in lettura.

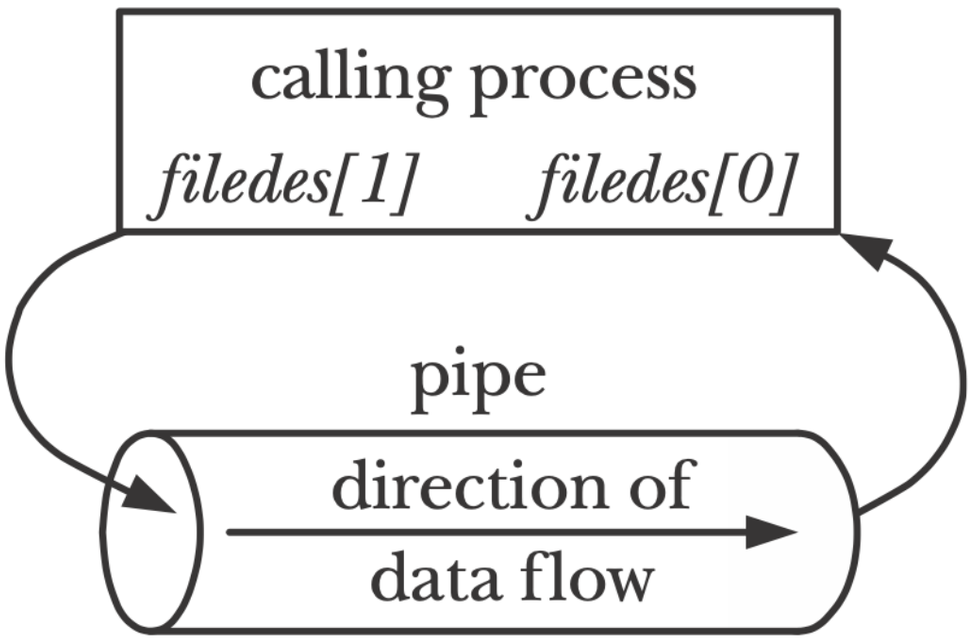
Capacità limitata

- Un pipe è semplicemente un buffer mantenuto in memoria.
- Questo buffer ha una ***capacità massima***. Una volta che un pipe è pieno, ulteriori tentativi di scrittura si bloccano finché il lettore rimuove alcuni dati dal pipe.
  - In generale, un'applicazione non ha bisogno di conoscere la capacità del pipe.
  - Se vogliamo evitare ai processi scrittori di restare bloccati, è necessario che i processi che leggono dal pipe siano progettati in modo da leggere i dati appena questi sono disponibili.

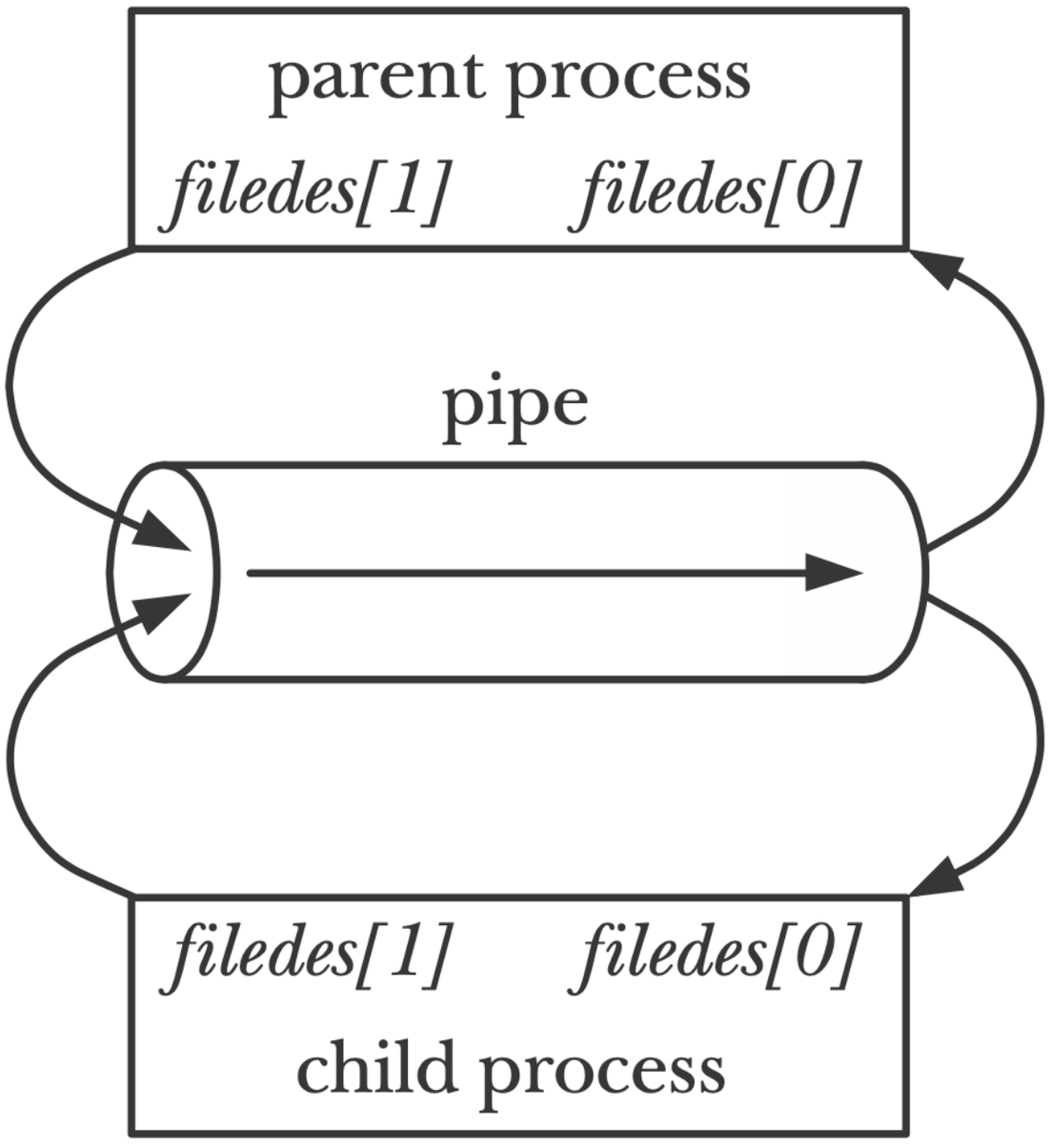
Capacità limitata

- In teoria, non ci sono motivi per cui un pipe non debba utilizzare capacità minime, fino al buffer costituito da un solo byte.
- La ragione per utilizzare buffer di dimensioni maggiori è l'efficienza: ogni volta che uno scrittore riempie il pipe, il kernel deve eseguire un ***context switch*** per consentire al lettore di essere 'scheduled' per prelevare qualche dato dal pipe.
  - L'utilizzo di un buffer di dimensione maggiore comporta la riduzione del numero di ***context switch***.

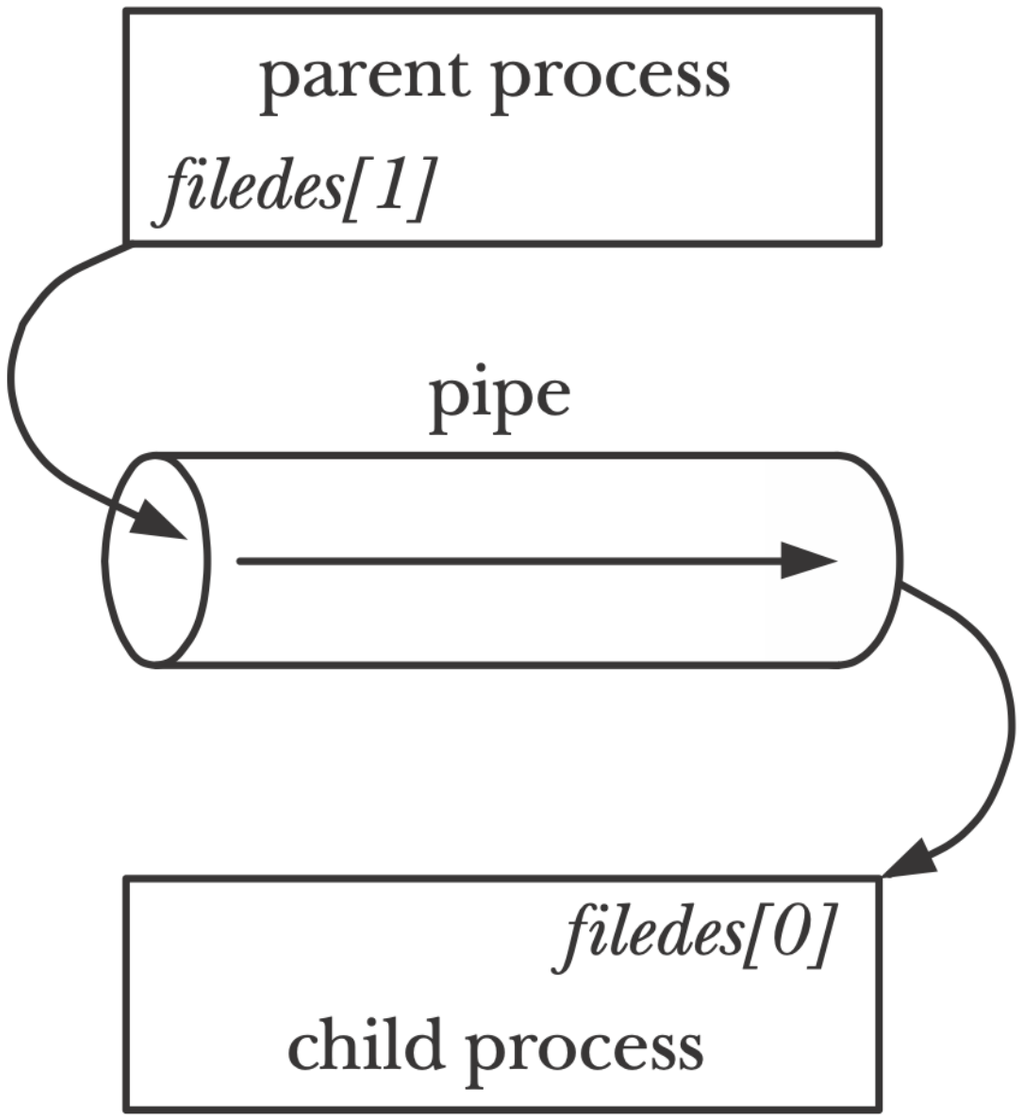
```
#include <unistd.h>

int pipe(int filedes[2]);

                Returns 0 on success, or –1 on error
```



- la system call ***pipe()*** crea un nuovo pipe; se va a buon fine, la chiamata alloca un array (***filedes***) contenente due descrittori di file aperti.
  - Un estremo è aperto in lettura (***filedes[0]***) e uno in scrittura (***filedes[1]***).
  - Come con qualsiasi descrittore di file, possiamo utilizzare le system call ***read()*** e ***write()*** per eseguire le operazioni di I/O sul pipe.



- Di norma si utilizzano i pipe per permettere la comunicazione fra due processi.
- Per collegare due processi con un pipe, eseguiamo prima una system call ***pipe()*** e poi una ***fork()***.
  - Con la ***fork()***, il processo figlio eredita copia dei descrittori di file dei genitori.



- tecnicamente sarebbe possibile leggere e scrivere sul pipe per il genitore e il figlio, ma non è il modo standard di utilizzare i pipe.
- subito dopo la ***fork()***, un processo chiude il proprio descrittore per l'estremità in scrittura, e l'altro chiude il proprio descrittore per l'estremità in lettura.
  - Per esempio, se il genitore deve inviare dei dati al figlio, deve chiudere l'estremità aperta in lettura, ***filedes[0]***, mentre il figlio deve chiudere ***filedes[1]***.

```c
int filedes[2];

if (pipe(filedes) == -1) // creazione pipe
  ddErrExit("pipe");

switch (fork()) {
  case -1:
  ddErrExit(“fork"); // da implementare
  case 0: // ----- Child
  if (close(filedes[1]) == -1)// --- chiude il write end
  ddErrExit("close");
  // --- il figlio compie qualche operazione ---
  break;
  default: // padre
  if (close(filedes[0]) == -1) // --- chiude il read end
    ddErrExit("close");

  // --- il padre compie qualche operazione ---

  break;
}
```

Comunicazione fra processi ***non*** imparentati

- I pipe possono essere usati per la comunicazione fra (due o più) processi ***parenti***, supponendo che il pipe sia creato da un ***antenato*** comune e ***prima*** della serie di ***fork()*** con cui sono stati creati i vari figli.
  - Per esempio, un pipe potrebbe essere usato per mettere in comunicazione un processo e un processo ***nipote*** (***granchild***) del primo. Il primo processo crea il pipe, e quindi effettua una ***fork()***, e il figlio effettua una ulteriore ***fork()*** creando così il nipote.
  - Un altro scenario possibile è l'utilizzo di un pipe per la comunicazione fra due processi ***fratelli*** (***siblings***): il loro genitore crea il pipe, e quindi crea i due figli.
- così facendo, la shell crea una ***pipeline***.

Chiusura dei descrittori inutilizzati (lettore)

- I file descriptors inutilizzati in lettura e in scrittura devono essere chiusi.
- Il processo che legge dal pipe chiude il proprio ***write*** descriptor, così che quando l'altro processo completa il proprio output e chiude il proprio descrittore ***write***, il lettore riceve un carattere terminatore, ***end-of-file***.
  - Se invece il processo lettore non chiude la propria estremità aperta in scrittura, dopo che l'altro processo avrà chiuso il proprio descrittore ***write***, il lettore ***non*** riceverà l'***end-of-file*** neppure dopo avere letto tutti i dati dal pipe.
  - In questo caso, una ***read()*** si bloccherebbe in attesa di dati (che sappiamo non arriveranno!) perché il kernel sa che c'è ancora almeno un descrittore di file aperto per il pipe.

Chiusura dei descrittori inutilizzati (scrittore)

- Il processo scrittore chiude l'estremità aperta in lettura del pipe per una ragione diversa. Quando un processo tenta di scrivere su un pipe per il quale nessun processo ha un descrittore aperto in lettura, il kernel invia il segnale ***SIGPIPE*** al processo scrittore.
  - Per default, tale segnale uccide il processo.
- Un processo può organizzarsi per intercettare o ignorare tale segnale; in questo caso la ***write()*** sul pipe fallisce con un errore ***EPIPE*** (broken pipe).
  - Ricevere il segnale ***SIGPIPE*** o ricevere l'errore ***EPIPE*** è un'indicazione utile in merito allo ***status*** del pipe, ed è la ragione per cui i descrittori aperti in lettura dovrebbero essere chiusi.

```c
int main(int argc, char** argv) {
  int     n;
  int     fd[2];
  pid_t   pid;
  char    line[MAXLINE];

  if (pipe(fd) < 0)
      ddErrExit("pipe error”);

  if ((pid = fork()) < 0) {
      ddErrExit("fork error");
  } else if (pid > 0) { // --------- padre ---
      close(fd[0]);
      write(fd[1], "hello world\n", 12);
  } else { // ---------------------- figlio ---
      close(fd[1]);
      n = read(fd[0], line, MAXLINE);
      write(STDOUT_FILENO, line, n);
  }
  exit(EXIT_SUCCESS);
}
```

```
...

int main(int argc, char *argv[]) {

    int pfd[2]; /* pipe file descriptors */
    char buf[BUF_SIZE];
    ssize_t numRead;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        ; /* stampa formato invocazione */

    if (pipe(pfd) == -1) /* creo il pipe */
        ; /* gestione errore */

    switch (fork()) {
    case -1:
        ; /* gestione errore */
    ...
```

pipe1.c

```
case 0: /* ------ figlio - legge dal pipe -------- */
  if (close(pfd[1]) == -1) /* chiusura write end */
      ; /* gestione errore */

  for (;;) { /* legge dal pipe, e scrive su stdout */
    numRead = read(pfd[0], buf, BUF_SIZE);
    if (numRead == -1)
       ; /* gestione errore */
    if (numRead == 0)
       break;  /* End-of-file */
    if (write(STDOUT_FILENO, buf, numRead) != numRead)
       ; /* gestione errore */
  }

write(STDOUT_FILENO, "\n", 1);
if (close(pfd[0]) == -1)
    ; /* gestione errore */
exit(EXIT_SUCCESS);
```

pipe1.c

```
...

default: /* ------ padre - scrive sul pipe -------- */
  if (close(pfd[0]) == -1) /* chiusura del read end */
    ; /* gestione errore */

  if(write(pfd[1],argv[1],strlen(argv[1]))!=strlen(argv[1]))
    ; /* gestione errore */

  if (close(pfd[1]) == -1) /* il figlio riceverà EOF */
    ; /* gestione errore */

    wait(NULL); /* attesa della terminazione del figlio */

    exit(EXIT_SUCCESS);

  } // end switch
}
```

pipe1.c

```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
                    Returns file stream, or NULL on error
```

- La ***popen()*** crea un pipe, quindi esegue una ***fork()***; il figlio generato esegue (***execs***) una shell, che a sua volta crea un processo figlio che esegue l'istruzione presente in ***command***.

- L'argomento ***mode*** è una stringa che determina se il processo chiamante leggerà dall'estremità ***read*** del pipe (mode settato a ***r***) o vi scriverà (mode a ***w***).

```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);

                        Returns file stream, or NULL on error
```

- Il valore di ***mode*** determina se lo standard output del comando eseguito è connesso all'estremità del pipe aperta in scrittura, o se il suo standard input è connesso all'estremità del pipe aperta in lettura.
  - In caso di successo, ***popen()*** restituisce un file stream pointer che può essere gestito con le funzioni della libreria ***stdio***.
  - In caso di fallimento (e.g., il mode non è ***r*** o ***w***, la creazione del pipe fallisce, o fallisce la ***fork()*** per creare il figlio), allora la ***popen()*** restituisce ***NULL*** e assegna ***errno*** indicando la causa dell'errore.

```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);

                    Returns file stream, or NULL on error
```


```
fp = popen(cmdstring, "r")
```

```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);

                    Returns file stream, or NULL on error
```


```
fp = popen(cmdstring, "r")
```

```
fp = popen(cmdstring, "w")
```

```
#include <stdio.h>

int pclose(FILE *stream);

        Returns termination status of child process,
                                        or -1 on error
```

- Completate le operazioni di *I/O*, si utilizza la funzione ***pclose()*** per chiudere il pipe ed attendere che la shell figlia termini.
  - In caso di successo, la ***pclose()*** ottiene lo status di terminazione della shell figlia (cioè, lo status di terminazione dell'ultimo comando eseguito dalla shell, a meno che la shell sia uccisa da un segnale).

convenience ***vs***. efficiency

- L'utilizzo della ***popen()*** garantisce *semplicità d'uso*.
  - La ***popen()*** crea il pipe, esegue la duplicazione dei descrittori, chiude i descrittori inutilizzati, e gestisce tutti i dettagli della ***fork()*** e della ***exec()*** per nostro conto.
- Tale semplicità d'uso ha un costo in termini di efficienza. Vengono creati almeno due processi in più: uno per la shell e uno o più per i comandi eseguiti dalla shell.
  - Come per ***system()***, ***popen()*** non dovrebbe essere utilizzata da programmi 'privileged'.

```c
#include <stdio.h>
#define PATH_MAX 1024

int main(int argc, char** argv) {
   FILE *fp;
   int status;
   char path[PATH_MAX];

   fp = popen("ls", "r");
   if (fp == NULL)
       ; // gestione errore

   while (fgets(path, PATH_MAX, fp) != NULL)
       printf("%s", path);

   status = pclose(fp);
   if (status == -1) {
       ; // gestione errore
   } else {
       ; // analisi dell'exit status
   }
   return(0);
}
```

FIFOs

FIFO e pipe

- Un ***FIFO*** è simile a un ***pipe***. La principale differenza è che un FIFO ha un nome all'interno del file system ed è aperto nello stesso modo di un file.
- Questo permette a un FIFO di essere utilizzato per comunicazioni fra processi non imparentati (e.g., un client e un server).

FIFO e pipe

- Una volta che un FIFO è stato aperto possiamo utilizzare le stesse system call dell'***I/O*** utilizzate con i pipe e gli altri file (cioè, ***read()***, ***write()***, e ***close()***).

- Come i pipe, anche i FIFO hanno un'estremità aperta in scrittura e una aperta in lettura, e come per i pipe, i dati sono letti nello stesso ordine in cui sono stati scritti.
  - Questa caratteristica conferisce ai FIFO il loro nome: ***first in, first out***. I FIFO sono anche noti come ***named pipes***.

- Possiamo creare un FIFO dalla shell con il comando ***mkfifo***:

```
$ mkfifo [ -m mode ] pathname
```

- Il ***pathname*** è il nome del FIFO che si intende creare, e l'opzione ***-m*** specifica i permessi analogamente al comando ***chmod***.

- Listato con il comando ***ls -l***, un FIFO è mostrato con il carattere ***p*** nella prima colonna.

```
$ mkfifo -m 644 prima_fifo
$ ls -l prima_fifo
prw-r--r--  1 radicion  staff  0 Nov 09 03:55 prima_fifo
```

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

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

                    Returns 0 on success, or -1 on error
```

- La funzione *mkfifo()* function crea un nuovo FIFO con il ***pathname*** dato.
- L'argomento ***mode*** specifica i permessi per il nuovo FIFO. Tali permessi sono specificati mettendo in OR (ORing) varie possibili costanti.

Constant	Octal value	Permission bit
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	User-read
S_IWUSR	0200	User-write
S_IXUSR	0100	User-execute
S_IRGRP	040	Group-read
S_IWGRP	020	Group-write
S_IXGRP	010	Group-execute
S_IROTH	04	Other-read
S_IWOTH	02	Other-write
S_IXOTH	01	Other-execute

sincronizzazione con FIFO

- L'utilizzo di un FIFO serve ad avere un processo lettore e uno scrittore alle due estremità del FIFO.
  - di default, l'apertura di un FIFO in lettura (flag ***O_RDONLY*** della ***open()***) si blocca finché un altro processo apre il FIFO in scrittura (flag ***O_WRONLY*** della ***open()***).
  - per contro, l'apertura del FIFO in scrittura si blocca finché un altro processo non apre la FIFO in lettura.
  - In altri termini, ***l'apertura di un FIFO sincronizza i processi lettori e scrittori***.
  - Se l'altra estremità del FIFO è già aperta (per esempio nel caso in cui una coppia di processi hanno già aperto ciascuna estremità del FIFO), la ***open()*** va immediatamente a buon fine.

user1.c

```c
#define FIFOSIZE 128


int main(){
  int fd;
  char * myfifo = "/tmp/myfifo";
  mkfifo(myfifo, 0666);
  char arr1[FIFOSIZE], arr2[FIFOSIZE];

  while (1)  {
    fd = open(myfifo, O_WRONLY); // apertura FIFO in scrittura
    fgets(arr2, FIFOSIZE, stdin); // lettura da stdin su arr2
    write(fd, arr2, strlen(arr2)+1);// scrittura e chiusura FIFO
    close(fd);

    fd = open(myfifo, O_RDONLY); // apertura FIFO in lettura
    read(fd, arr1, sizeof(arr1)); // lettura da FIFO su arr1
    printf("User2: %s\n", arr1);
    close(fd);                    // chiusura FIFO
  }
  return 0;
}
```

user2.c

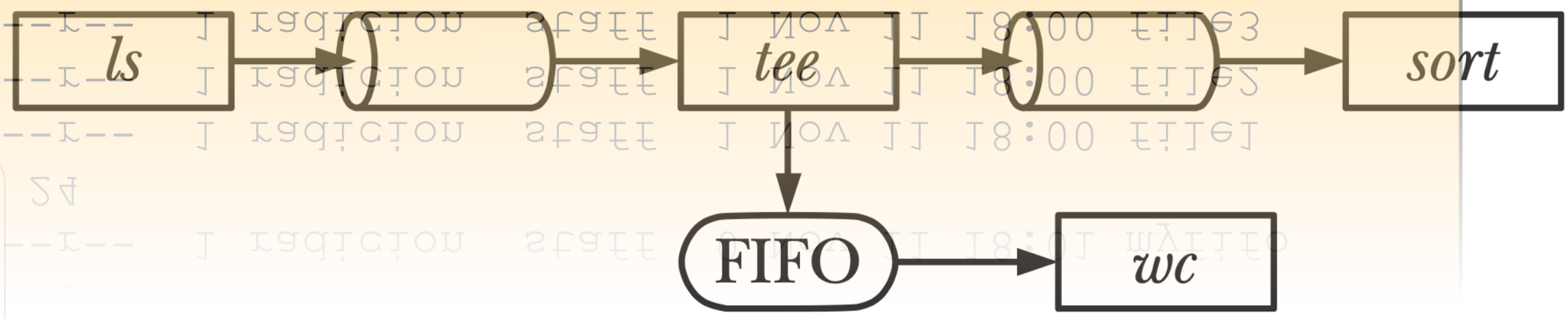
```
#define FIFOSIZE 128

int main(){
  int fd1;

  char * myfifo = "/tmp/myfifo";
  mkfifo(myfifo, 0666);
  char arr1[FIFOSIZE], arr2[FIFOSIZE];

  while (1){
    fd1 = open(myfifo,O_RDONLY); // apertura FIFO
    read(fd1, arr1, FIFOSIZE); // lettura da FIFO su arr1 e
chiusura
    close(fd1);
    printf("User1: %s\n", arr1);

    fd1 = open(myfifo,O_WRONLY); // apertura FIFO in scrittura
    fgets(arr2, FIFOSIZE, stdin); // lettura da stdin su arr2
    write(fd1, arr2, strlen(arr2)+1); // scrittura su FIFO
    close(fd1);
  }
  return 0;
}
```

FIFOs and ***tee()***

- Una delle caratteristiche delle pipeline di shell è che esse sono sequenziali; ogni processo nella pipeline legge dati prodotti dal proprio predecessore, e invia i dati al proprio successore.
- Utilizzando i FIFO, è possibile creare una fork nella pipeline, così che un duplicato dell'output di un processo viene inviato a un altro processo oltre che al proprio successore nella pipeline.
  - Per fare questo si utilizza il comando ***tee***, che scrive due copie di ciò che legge dal proprio standard input: uno su standard output e l'altro sul file specificato dal suo argomento della linea di comando.

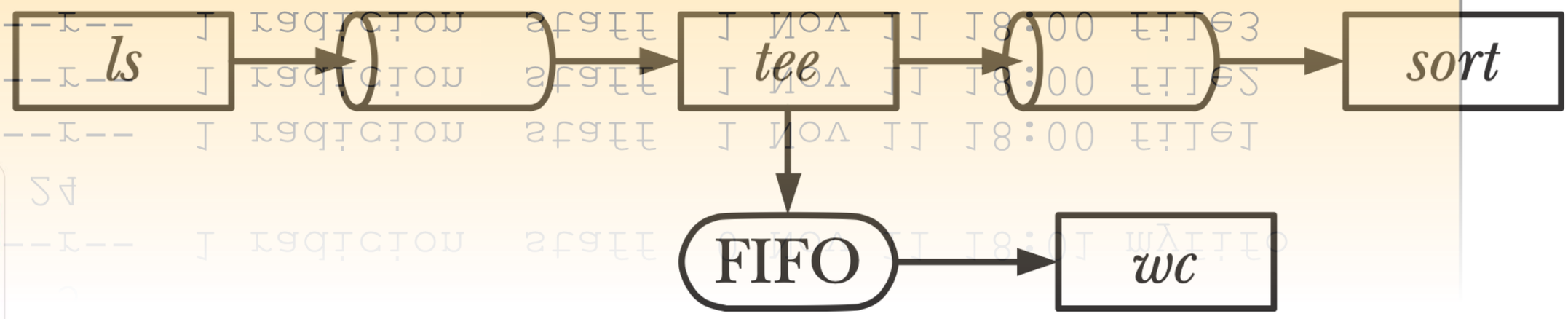
```
$ mkfifo myfifo
$ ls -l
total 24
-rw-r--r--  1 radicion  staff    1 Nov 11 18:00 file1
-rw-r--r--  1 radicion  staff    1 Nov 11 18:00 file2
-rw-r--r--  1 radicion  staff    1 Nov 11 18:00 file3
prw-r--r--  1 radicion  staff    0 Nov 11 18:00 myfifo
$ wc -l < myfifo &
[1] 24651
$ ls -l | tee myfifo | sort -k5n
       5
prw-r--r--  1 radicion  staff  0 Nov 11 18:01 myfifo
total 24
-rw-r--r--  1 radicion  staff  1 Nov 11 18:00 file1
-rw-r--r--  1 radicion  staff  1 Nov 11 18:00 file2
-rw-r--r--  1 radicion  staff  1 Nov 11 18:00 file3
[1]+  Done                    wc -l < myfifo
```

```
$ mkfifo myfifo
$ ls -l
total 24
-rw-r--r--  1 radicion  staff    1 Nov 11 18:00 file1
-rw-r--r--  1 radicion  staff    1 Nov 11 18:00 file2
-rw-r--r--  1 radicion  staff    1 Nov 11 18:00 file3
prw-r--r--  1 radicion  staff    0 Nov 11 18:00 myfifo
$ wc -l < myfifo &
[1] 24651
$ ls -l | tee myfifo | sort -k5n
       5
prw-r--r--  1 radicion  staff  0 Nov 11 18:01 myfifo
total 24
-rw-r--r--  1 radicion  staff  1 Nov 11 18:00 file1
-rw-r--r--  1 radicion  staff  1 Nov 11 18:00 file2
-rw-r--r--  1 radicion  staff  1 Nov 11 18:00 file3
[1]+  Done                    wc -l < myfifo
```



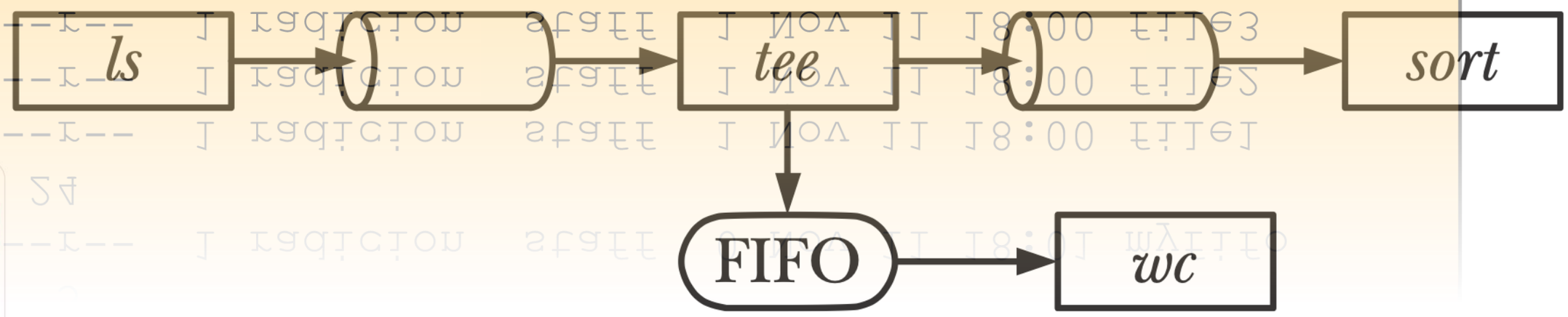
```
$ mkfifo myfifo
$ ls -l
total 24
-rw-r--r--  1 radicion  staff    1 Nov 11 18:00 file1
-rw-r--r--  1 radicion  staff    1 Nov 11 18:00 file2
-rw-r--r--  1 radicion  staff    1 Nov 11 18:00 file3
prw-r--r--  1 radicion  staff    0 Nov 11 18:00 myfifo
$ wc -l < myfifo &
[1] 24651
$ ls -l | tee myfifo | sort -k5n
       5
prw-r--r--  1 radicion  staff  0 Nov 11 18:01 myfifo
total 24
-rw-r--r--  1 radicion  staff  1 Nov 11 18:00 file1
-rw-r--r--  1 radicion  staff  1 Nov 11 18:00 file2
-rw-r--r--  1 radicion  staff  1 Nov 11 18:00 file3
[1]+  Done                    wc -l < myfifo
```

creo una FIFO di nome *myfifo*



```
$ mkfifo myfifo
$ ls -l
total 24
-rw-r--r--  1 radicion
-rw-r--r--  1 radicion
-rw-r--r--  1 radicion
prw-r--r--  1 radicion  staff    0 Nov 11 18:00 myfifo
$ wc -l < myfifo &
[1] 24651
$ ls -l | tee myfifo | sort -k5n
       5
prw-r--r--  1 radicion  staff  0 Nov 11 18:01 myfifo
total 24
-rw-r--r--  1 radicion  staff  1 Nov 11 18:00 file1
-rw-r--r--  1 radicion  staff  1 Nov 11 18:00 file2
-rw-r--r--  1 radicion  staff  1 Nov 11 18:00 file3
[1]+  Done                    wc -l < myfifo
```

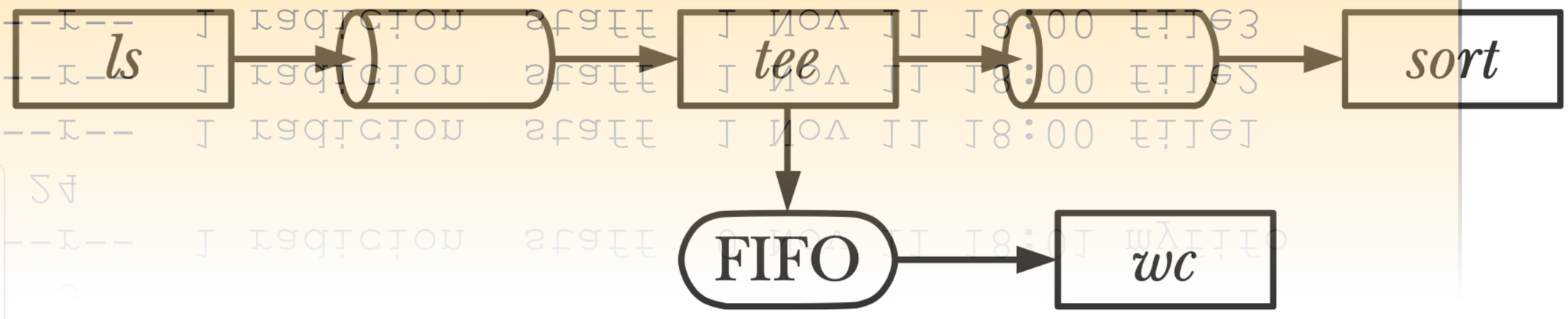
comando *wc* in background, che apre la FIFO per leggerne l'output (NB: *wc* resta bloccato finché la FIFO non viene aperta in scrittura...)



```
$ mkfifo myfifo
$ ls -l
total 24
-rw-r--r--  1 radicion  staff    1 Nov 11 18:00 file1
-rw-r--r--  1 radicion  staff    1 Nov 11 18:00 file2
-rw-r--r--  1 radicion  staff    1 Nov 11 18:00 file3
prw-r--r--  1 radicion  staff    0 Nov 11 18:00 myfifo
$ wc -l < myfifo &
[1] 24651
$ ls -l | tee myfifo | sort -k5n
       5
prw-r--r--  1 radicion  staff  0 No
total 24
-rw-r--r--  1 radicion  staff  1 No
-rw-r--r--  1 radicion  staff  1 Nov 11 18:00 file2
-rw-r--r--  1 radicion  staff  1 Nov 11 18:00 file3
[1]+  Done                    wc -l < myfifo
```

esecuzione della pipeline che invia l'output di *ls* a *tee*, che lo duplica passandolo sia a *sort*, sia a *myfifo* (che lo passa a *wc*)

opzione *–k5n*: *sort* utilizza l'output di *ls* per un ordinamento numerico (*n*) condotto sul quinto campo (key, *k*).



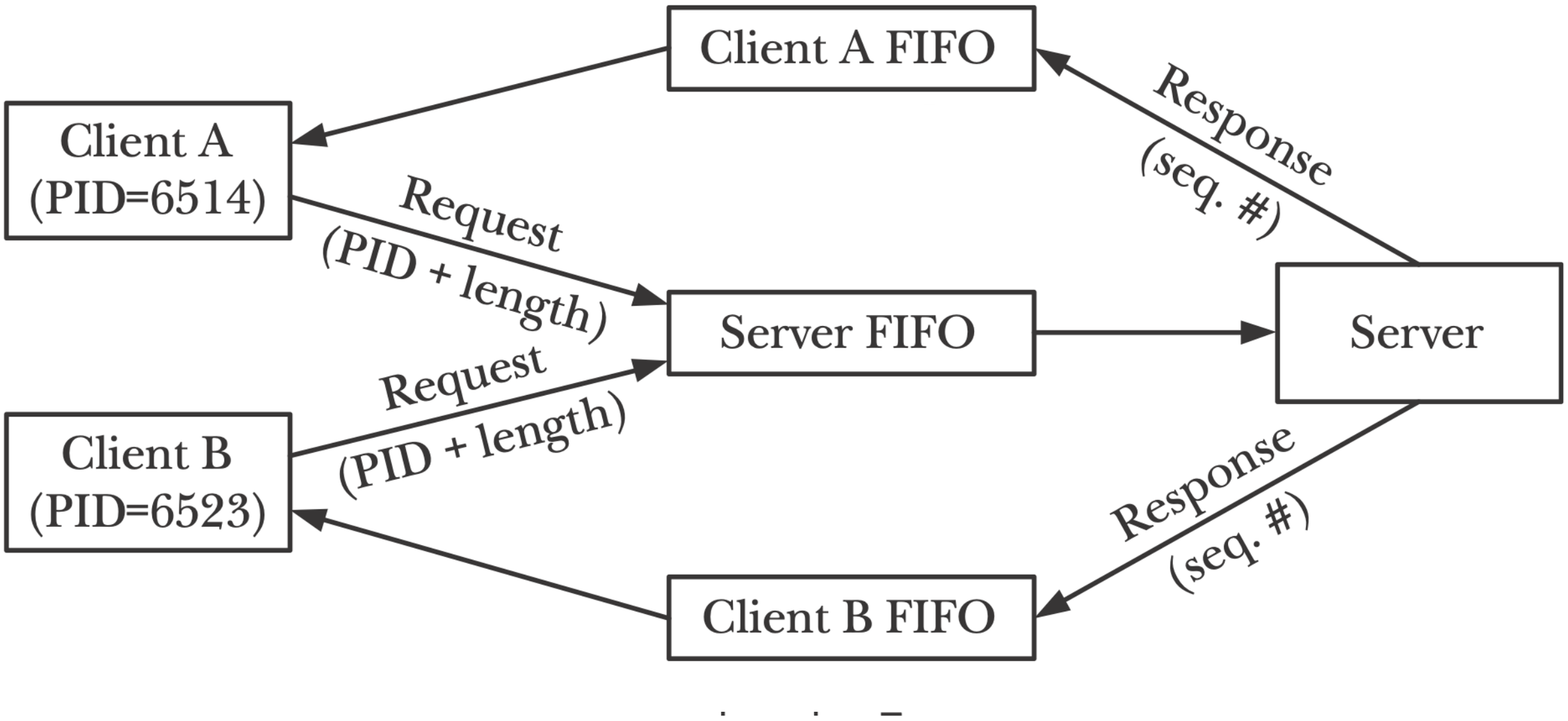
un'applicazione client-server di esempio

Client-Server Application with FIFOs

- The server provides the service of assigning unique sequential numbers to each client that requests them.
- all clients send their requests to the server using a single server FIFO. The header file defines the well-known name that the server uses for its FIFO.
  - This name is fixed, so that all clients know how to contact the server.
- however, it is not possible to use a single FIFO to send responses to all clients, since multiple clients would race to read from the FIFO, and possibly read each other's response messages rather than their own.

Client-Server Application with FIFOs

- Therefore, each client creates a unique FIFO that the server uses for delivering the response for that client, and the server needs to know how to find each client's FIFO.
  - Client and server can agree on a convention for constructing a client FIFO pathname, and, as part of its request, the client can pass the server the information required to construct the pathname specific to this client.
    - Each client's FIFO name is built from a template consisting of a pathname containing the client's process ID.

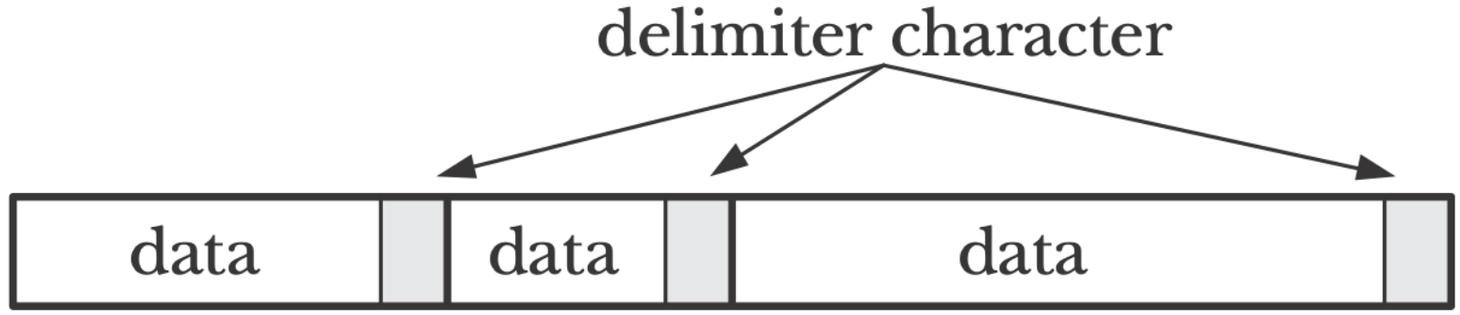

Client A FIFO
Client A
(PID=6514)
Request
(PID + length)
Response
(seq. #)
Server FIFO
Server
Request
(PID + length)
Client B
(PID=6523)
Response
(seq. #)
Client B FIFO

protocol

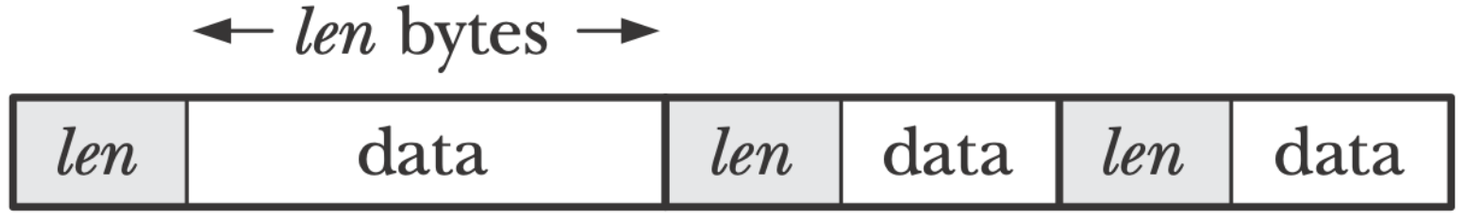
- Recall that the data in pipes and FIFOs is a byte stream; boundaries between multiple messages are not preserved.
  - When multiple messages are being delivered to a single process, the sender and receiver must agree on some convention for separating the messages.
    - Terminate each message with a delimiter character, such as a newline character.
    - Include a fixed-size header with a length field in each message specifying the number of bytes in the remaining variable-length component of the message.
    - Use fixed-length messages, and have the server always read messages of this fixed size.

protocol

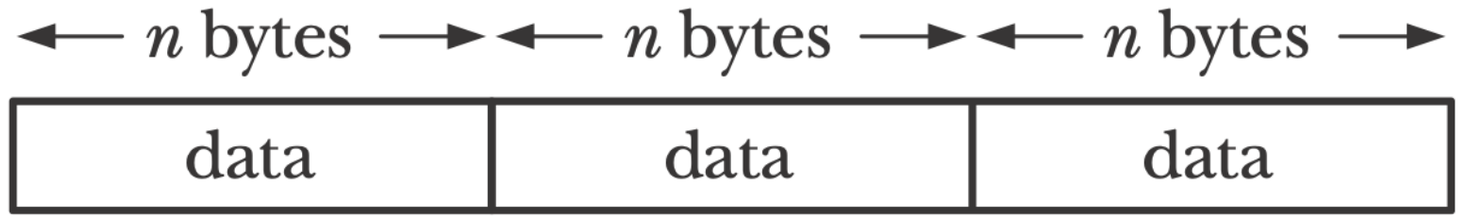
delimiter character

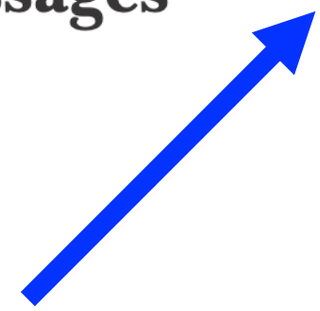


header with length field



fixed-length messages





Michael Kerrisk, *The Linux Programming interface - a Linux and UNIX® System Programming Handbook*, No Starch Press, San Francisco, CA, 2010.

protocol

```
#define SERVER_FIFO "seqnum_server"
                /* Well-known name for server's FIFO */
#define CLIENT_FIFO_TEMPLATE "seqnum_client.%ld"
                /* Template for building client FIFO name */
#define CLIENT_FIFO_NAME_LEN (sizeof(CLIENT_FIFO_TEMPLATE) +
20)
                /* Space required for client FIFO pathname
                (+20 as a generous allowance for the PID) */

struct request { /* Request (client --> server) */
    pid_t pid;   /* PID of client */
    int seqLen;  /* Length of desired sequence */
};

struct response { /* Response (server --> client) */
    int seqNum;   /* Start of sequence */
};
```

server

- The server performs the following steps:
  - Create the server's well-known FIFO and open the FIFO for reading. The server's ***open()*** blocks until the first client opens the other end of the server FIFO for writing.
  - Open the server's FIFO once more, this time for writing.
    - This will never block, since the FIFO has already been opened for reading. This second open is a convenience to ensure that the server doesn't see end-of-file if all clients close the write end of the FIFO.

server

- The server performs the following steps:
  - Enter a loop that reads and responds to each incoming client request. To send the response, the server constructs the name of the client FIFO and then opens that FIFO.
  - If the server encounters an error in opening the client FIFO, it abandons that client's request.

client

- The client performs the following steps:
  - Create a FIFO to be used for receiving a response from the server. This is done before sending the request, in order to ensure that the FIFO exists by the time the server attempts to open it and send a response message.
  - Construct a message for the server containing the client's process ID and a number (taken from an optional command-line argument) specifying the length of the sequence that the client wishes the server to assign to it.
    - If no command-line argument is supplied, the default sequence length is 1.
  - Open the server FIFO and send the message to the server.
  - Open the client FIFO, and read and print the server's response.

```
$ ./seq_server &
[1] 29685
$ ./seq_client 3
0
$ ./seq_client 2
3
$ ./seq_client
5
$
```