Programmation Système

Les tubes

Emmanuel Bouthenot (emmanuel.bouthenot@u-bordeaux.fr)

Licence professionnelle ADSILLH - Université de Bordeaux

2020/2021

Les tubes

Définition

- Les tubes (**pipes**) sont un mécanisme de communication entre processus (*IPC* : InterProcess Communication)
- Sous UNIX, les **IPC** peuvent prendre de multiples formes. Ils facilitent le travail du programmeur et ils sont un élément clé de la réutilisation de composants.

Les tubes dans un shell

L'utilisation des tubes dans un shell est une implémentation concrète des tubes UNIX.

- `ps faux | less`
  - il n'est pas nécessaire d'implémenter la pagination dans chaque outil, il suffit d'un outil qui se charge de cette tâche, et uniquement cette tâche (KISS).
- `ps faux | cut -d ' ' -f1 | sort -u`
  - Enchainement de tâches complexes avec des outils simples

Caractéristiques des tubes

- Les tubes sont un mécanisme de transfert de données sous forme de flux entre processus. Le flux écrit d'un côté du tube peut être lu de l'autre.
- Les tubes sont créés directement par le noyau
- Une fois créés, les tubes sont manipulables par des descripteurs de fichiers
- Les tubes ne sont accessible que par les processus qui y sont associés
- Les tubes persistent le temps de la vie du processus, ils disparaissent à la terminaison du processus
- Les tubes sont portables et disponibles sur tous les UNIX connus.

pipe

La création de tube passe par un appel système au noyau :

```c
#include <unistd.h>

int pipe(int pipefd[2]);

// En cas de succès, renvoi 0. En cas d'échec, renvoi -1 et
// positionne errno en conséquence
```

- pipefd est un tableau de descripteurs de fichiers qui sera rempli par le noyau avant le retour de l'appel
- pipefd[0] est ouvert en lecture
  pipefd[1] est ouvert en écriture
  Rappel mnémotechnique : STDIN(0) / STDOUT(1)
- La sortie de pipefd[1] est la sortie de pipefd[1]
  Les tubes sont un canal de communication half-duplex (1 seul sens)

Pipes - Overview

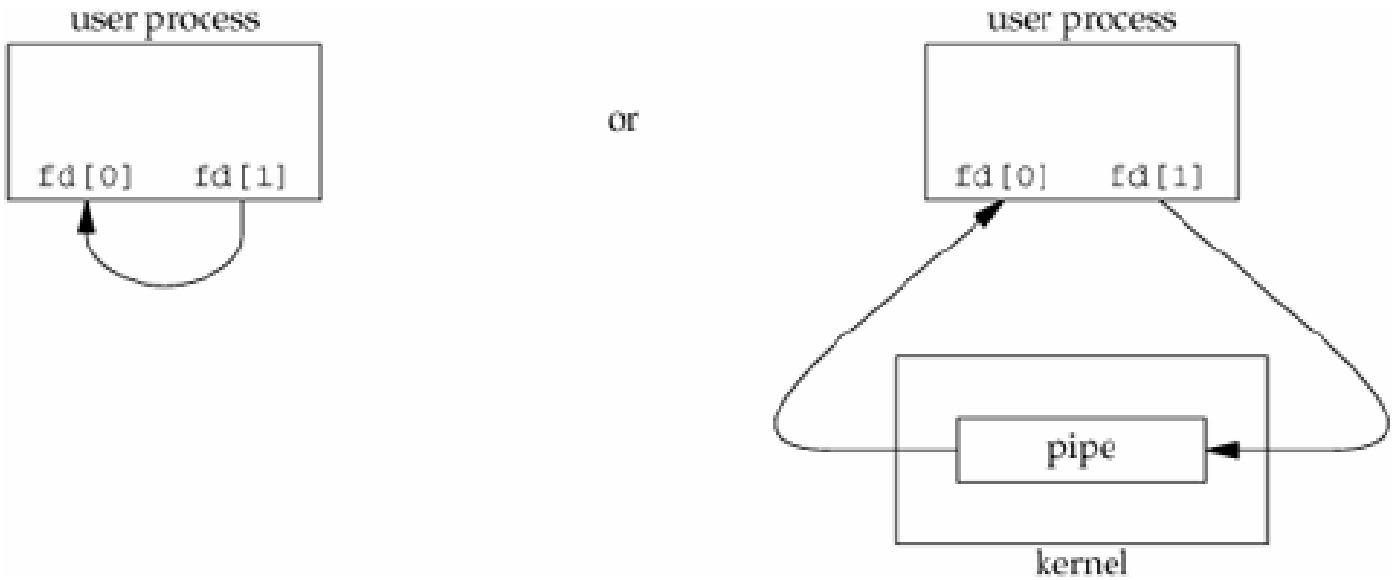


Figure: APUE 15.2

- Sur la gauche, le point de vue du programmeur
- Sur la droite, le point de vue de l'implémentation. Chaque lecture ou écriture dans le tube fait transiter les données de l'espace utilisateur vers l'espace noyau (et inversement)

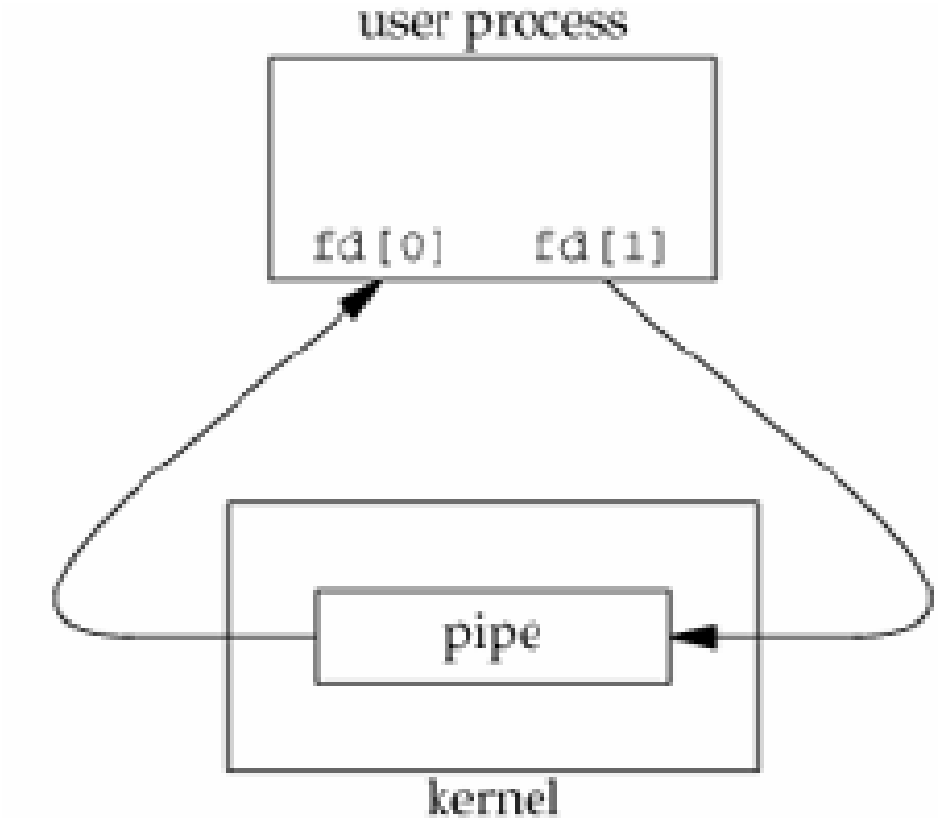
Mode half-duplex

Patron de conception (design pattern)

1. `pipe(fds)`
2. `fork()`
3. Processus père : `close(fds[0])`
4. Processus fils : `close(fds[1])`
5. Processus père qui écrit des données au fils : `write(fds[1], ...)`
6. Processus fils qui lit les données du père : `read(fds[0], ...)`

Half Duplex - étape 1

après le **pipe()**...



Half Duplex - étape 2

après le pipe() et le **fork()**...

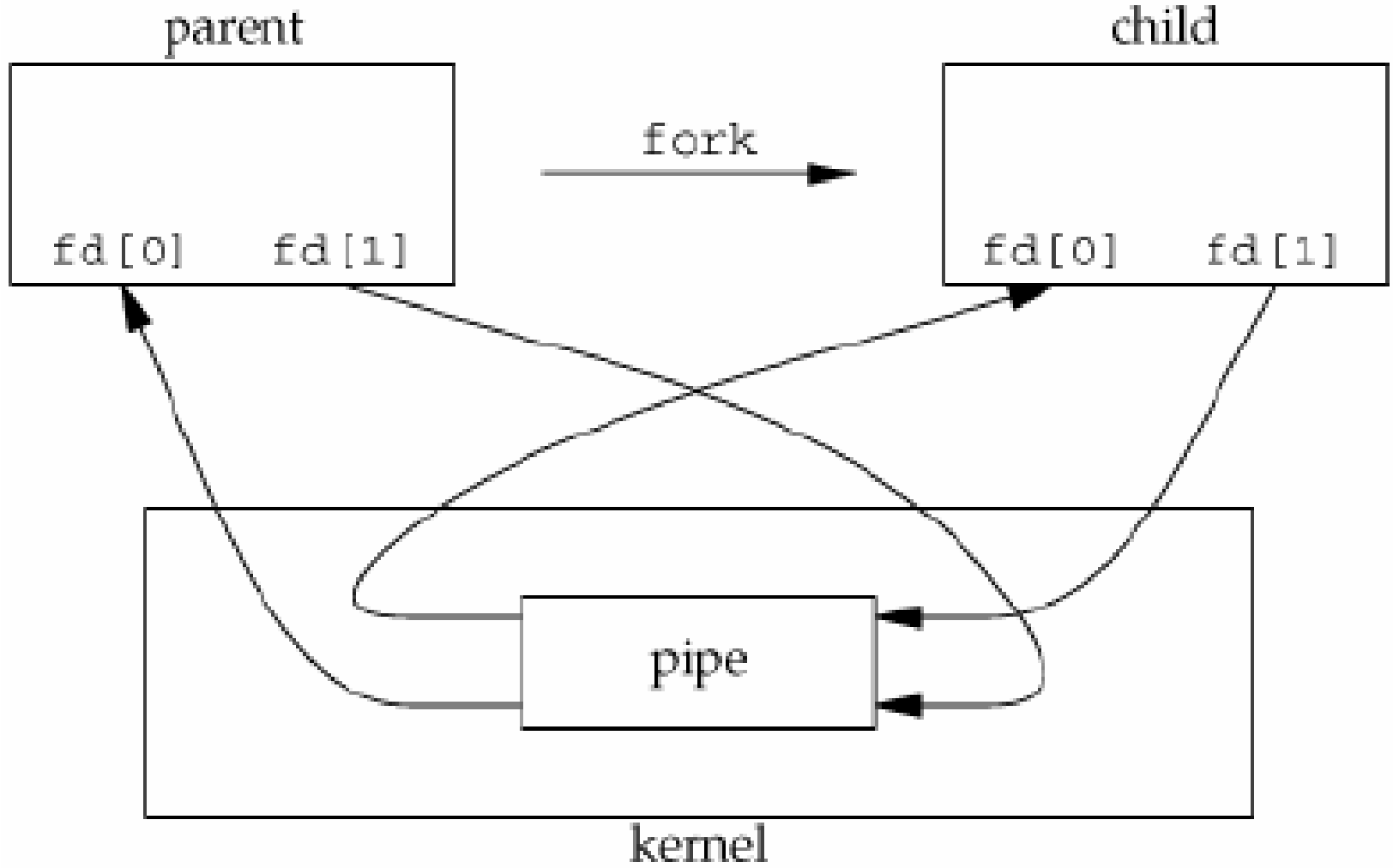


Figure: APUE 15.3

Half Duplex - étape 3

après le pipe(), le fork et le **close()** sur les extrémités du tube non utilisées ...

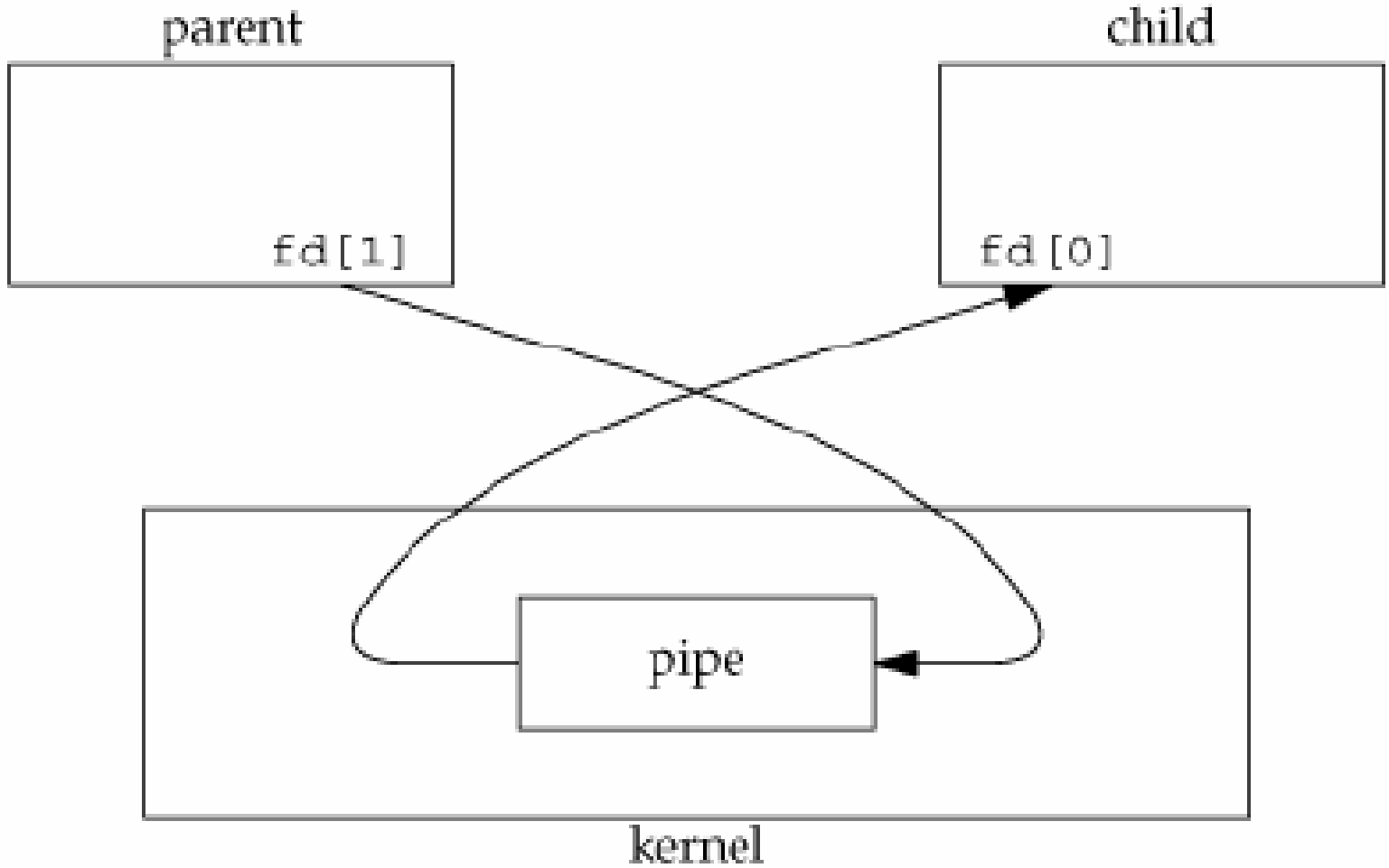


Figure: APUE 15.4

Half Duplex - Exemple

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

#define BUFMAX 256

int main () {
    char *buffer[BUFMAX];
    pid_t pid;
    int n, fds[2];

    if (pipe(fds) == -1) {
        perror("Unable to create pipe");
    }
    pid = fork();
    if (pid == -1) {
        perror("Unable to fork");
    }
    else if (pid > 0) { /* parent */
        if (close(fds[0]) == -1) {
            perror("Unable to close pipe from parent");
        }
        write(fds[1], "I am your father\n", 17);
    }
    else { /* child */
        if (close(fds[1]) == -1) {
            perror("Unable to close pipe from child");
        }
        n = read(fds[0], buffer, BUFMAX);
        write(STDOUT_FILENO, buffer, n);
    }
    exit(EXIT_SUCCESS);
}
```

Half Duplex - Demo

```
> gcc -Wall pipe_half-duplex.c -o pipe_half-duplex
> ./pipe_half-duplex
I am your father
```

Half Duplex - Python - Exemple

```python
#!/usr/bin/python3
import os, sys

BUFMAX = 256

def main():
    try:
        read_fd, write_fd = os.pipe()
    except OSError as e:
        sys.stderr.write("Unable to create a pipe: {}".format(e))
        sys.exit(os.EX_OSERR)
    try:
        pid = os.fork()
    except OSError as e:
        sys.stderr.write("Unable to fork: {}".format(e))
        sys.exit(os.EX_OSERR)
    if pid > 0:
        try:
            os.close(read_fd)
        except IOError as e:
            sys.stderr.write("Unable to close pipe from parent: {}".format(e))
        else:
            os.write(write_fd, b"I am your father")
    else:
        try:
            os.close(write_fd)
        except IOError as e:
            sys.stderr.write("Unable to close pipe from child: {}".format(e))
        else:
            data = os.read(read_fd, 64)
            sys.stdout.write(data.decode('utf-8'))
    sys.exit(os.EX_OK)
main()
```

Mode full-duplex

- Les tubes sont half-duplex
- Certains UNIX proposent des tubes full-duplex mais ils sont moins portables et rarement utilisés
- Pour faire des tubes full-duplex, il suffit de 2 tubes utilisés dans des directions opposées

Patron de conception (design pattern)

1. `pipe(p2c); pipe(c2p)`
2. `fork()`
3. Processus père : `close(p2c[0]; close(c2p[1]))`
4. Processus fils : `close(p2c[1]); close(c2p[0])`
5. Processus père qui écrit des données au fils : `write(p2c[1], ...)`
6. Processus fils qui écrit des données au père : `read(c2p[1], ...)`

pipe - Cas particuliers

- En général, on ferme les extrémités du tube avant de l'utiliser.
- Une lecture (`read()`) sur une extrémité du tube déjà fermé retourne 0.
- Une écriture (`write()`) sur une extrémité du tube déjà fermé retourne -1 avec errno positionné à `EPIPE`.
  De plus, le signal `SIGPIPE` est envoyé au processus qui essaye d'écrire.

Les filtres UNIX

Définition

Les filtres UNIX, sont des programmes qui lisent leur données en entrée depuis l'entrée standard (**stdin**) et écrivent leurs données en sorties sur la sortie standard (**stdout**).

Parmi les filtres les plus utilisés ont trouve :
`cat`, `cut`, `grep`, `sed`, `sort`, `uniq`, `head`, `tail`, `wc`, ...

Les filtres et les tubes

Imaginons un programme pour lequel on souhaite avoir une pagination. Dans l'idéal on utilisera le `$PAGER` du système (e.g less ou more) au lieu d'en réécrire un.

Patron de conception (design pattern)

1. `pipe()`
2. `fork()`
   Le processus père produit les données qui seront paginées
   Le processus fils exécutera le programme de pagination
3. Le processus fils duplique la sortie en lecture du tube sur l'entrée standard **stdin**. *(En lisant depuis l'entrée standard, le fils lira en fait depuis le tube.)*
4. Le processus fils exécutera le programme de pagination qui lira les données depuis son entrée standard
5. Le processus père écrira ses données dans le pipe qui seront lues par le fils à l'autre extrémité.

Duplication de descripteur de fichiers

```c
#include <unistd.h>

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

// En cas de succès, renvoi 0. En cas d'échec, renvoi -1 et
// positionne errno en conséquence
```

dup2() transforme `newfd` en une copie de `oldfd` (`newfd` est fermé si besoin).

- Si `oldfd` n'est pas un descripteur de fichier valable, alors l'appel échoue et `newfd` n'est pas fermé.
- Si `oldfd` est un descripteur de fichier valable et newfd a la même valeur que `oldfd`, alors `dup2()` ne fait rien et renvoie `newfd`.

Les filtres et les tubes - Exemple

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

#define PAGER "less"

int main () {
    pid_t pid;
    int status, fds[2];
    FILE *fdout;

    if (pipe(fds) == -1) {
        perror("Unable to create pipe");
    }
    pid = fork();
    if (pid == -1) {
        perror("Unable to fork");
    }
    else if (pid > 0) { /* parent */
        if (close(fds[0]) == -1) {
            perror("Unable to close pipe from parent");
        }
        fdout = fdopen(fds[1], "w");
```

Les filtres et les tubes - Exemple (suite)

```
        if (fdout == NULL) {
            perror("Unable to open pipe as a stream for writing");
        }
        for(int i=1; i<=1000; i++) {
            fprintf(fdout, "%d\n", i);
        }
        fclose(fdout);
        wait(&status);
    }
    else { /* child */
        if (close(fds[1]) == -1) {
            perror("Unable to close pipe from child");
        }
        if (dup2(fds[0], STDIN_FILENO) != STDIN_FILENO) {
            perror("Unable to duplicate stdin file descriptor");
        }
        close(fds[0]);
        execlp(PAGER, PAGER, NULL);
    }
    exit(EXIT_SUCCESS);
}
```

Bibliographie

[**APUE**] Advanced Programming in the UNIX Environment.
W. Richard Stevens and Stephen A. Rago.
Addison-Wesley Professional, 2005.

[**TLPI**] The Linux Programming Interface.
Michael Kerisk.
No Starch Press, 2010.