Programmation Système

Tubes

Abdelkader Gouaïch

2022

Introduction aux IPC

Nous allons dans ce chapitre renforcer nos connaissances afin de maîtriser les techniques de coordination des activités entre des processus indépendants.

Cette coordination va nécessiter de la communication afin de synchroniser les actions ou partager des données.

L'ensemble de ces techniques est appelé *Inter Process Communication (IPC)* dans l'univers Unix.

Plus précisément, les mécanismes des IPC vont nous permettre:

— d'envoyer des signaux entre et en destination de processus
— de synchroniser les activités entre processus
— de transférer des données entre processus

IPC: Les Signaux

Les signaux, que nous avons étudiés dans le chapitre précédent, vont permettre à des processus de connaître un changement d'état de leur environnement et d'échanger des informations élémentaires codées avec juste des entiers.

IPC: La synchronisation

Dans cette catégorie nous allons trouver les mécanismes de coordination des activités. Cela comprend: les sémaphores entre processus, les mutex entre les threads, ainsi que les primitives d'attente comme `wait` et `join`.

IPC: La communication

Cette catégorie regroupe les mécanismes pour échanger des structures de données. Les modalités d'échange sont: (i) le transfert direct d'octets en utilisant soit un flux ou des messages; (ii) le partage d'un segment mémoire entre les mémoires virtuelles des processus.

Transfert de données par flux et messages

Dans le modèle de communication par flux, les processus envoient et reçoivent les octets sur un canal de communication. Ce canal est semblable à un fichier: nous pouvons écrire et lire des blocs d'octets avec un ordre de lecture qui respecte celui de l'écriture (First In First Out)

Plusieurs objets vont permettre l'échange par flux de données. Nous pouvons citer: les tubes, les tubes nommés ainsi que les sockets TCP.

Dans le modèle de communication par messages, nous définissons une unité d'échange, appelée *le message*. Le message sera transféré entre les différents processus en utilisant un protocole de communication.

Les messages sont échangés et rangés des boites à lettres à la réception. L'ordre d'arrivée des messages dans ce modèle n'est pas nécessairement celui des envois. Nous pouvons par exemple traiter des messages par priorité. Le réseau peut aussi inverser l'ordre de l'arrivée car les messages peuvent emprunter des chemins différents.

Les objets concrets pour permettre l'échange par des messages sont: les files de messages et les sockets UDP.

Communication par mémoire partagée

Une autre façon de communiquer consiste à partager tout simplement un segment mémoire. Ainsi, chaque écriture dans ce segment partagé sera visible par tous les processus qui le partagent.

Nous pouvons citer les objets suivants pour partager un segment: mémoire partagée (shared memory) et mapping de mémoire (memory mapping).

Introduction aux tubes unix

Nous avons déjà rencontré les tubes avec le shell. Considérons par exemple la commande suivante :

```
ls | grep '.txt'
```

Cette commande va créer un tube qui fera le lien entre le flux sortant du processus `ls` et le flux entrant du processus `grep`. C'est la coordination entre ces deux processus qui fera la tâche demandée: lister uniquement les fichiers avec le suffixe '.txt'

Dans ce chapitre nous allons étudier les tubes et voir comment les utiliser à partir de nos programmes C en utilisant l'API des IPCs.

Avant de commencer ce travail technique, essayons de lister les principales caractéristiques des tubes.

Les tubes sont orientés flux

Quand nous parlons d'un flux de données (ou d'octets pour être précis), nous spécifions que la communication se fera par des échanges d'octets. Le nombre d'octets échangés lors d'une session n'est pas limité et n'est pas forcément connu, à l'avance, par le destinataire. Par contre, le destinataire peut préciser le nombre d'octets qu'il souhaite retirer à chaque opération de lecture.

La notion de flux spécifie aussi que l'ordre de octets écrits est préservé lors de la lecture.

Nous voyons donc que le flux émule un fichier standard pour les opérations de lecture et d'écriture avec la différence que nous ne pouvons pas déplacer le curseur de lecture, qui n'existe pas, avec la fonction `lseek()`.

Lecture dans un tube et la fin de communication

Il est important de noter qu'une demande de lecture dans un tube vide va bloquer le lecteur. Si par l'extrémité d'écriture du tube est fermée alors le lecteur va lire la constante `EOF` (End Of File) pour signifier que toutes les données ont été lues et qu'aucun écrivain n'est présent: la communication est donc terminée.

Le tube est unidirectionnel

Les tubes sont unidirectionnels, nous devons alors respecter un sens de transfert des données qui se fera toujours de l'extrémité d'écriture vers l'extrémité de lecture. Les processus se trouvant aux extrémités doivent alors prendre des rôles d'écrivain (émetteur)) et de lecteur (destinataire)

L'écriture concurrente sur un tube

Si plusieurs processus partagent le même tube alors il est possible d'écrire jusqu'à n octets atomiquement i.e sans problème de synchronisation. Pour bénéficier de

cela, le nombre d'octets n doit être inférieur à la valeur `PIPE_BUF`.

La taille de mémoire tampon d'un buffer

Un tube possède une mémoire tampon de taille limitée. Si la mémoire tampon est atteinte alors toutes les nouvelles tentatives d'écriture seront bloquées en attendant que les lecteurs consomment les octets déjà présents.

Les tubes anonymes

Création et utilisation d'un tube anonyme

```
#include <unistd.h>
int pipe(int filedes [2]);
```

La fonction `pipe()` prend en paramètre un tableau d'au moins deux entiers.

Ce tableau sera rempli par la fonction `pipe` comme suit:

- la case d'index 0 va contenir le descripteur de lecture. Nous appellerons ce descripteur *l'extrémité lecture* du tube.
- la case d'index 1 va contenir le descripteur d'écriture. Nous appellerons ce descripteur *l'extrémité d'écriture* du tube.

Une fois que les descripteurs sont donnés par `pipe()`, nous pouvons les utiliser par la suite avec les fonctions E/S bas niveau usuelles (read, write, close, dup2)

Modèle de flux de données

Avec ces deux extrémités, le fonctionnement d'un tube va suivre le schéma ci-dessus. Un processus va pouvoir écrire dans l'extrémité d'écriture; les octets seront lus par la suite dans l'extrémité de lecture.

Comment utiliser les tubes ?

Nous avons présenté un tube comme un objet composé de deux extrémités: une extrémité pour écrire les données et une autre pour les lire. Cet objet est configuré pour créer un flux directionnel d'octets entre ses deux extrémités.

Se pose alors la question de l'utilisation de cet objet ?

Si un seul processus a connaissance du tube alors le bénéfice est limité: le processus, va écrire et lire ses propres données. L'utilisation du tube sera dans ce cas similaire à l'utilisation d'une mémoire tampon annexe.

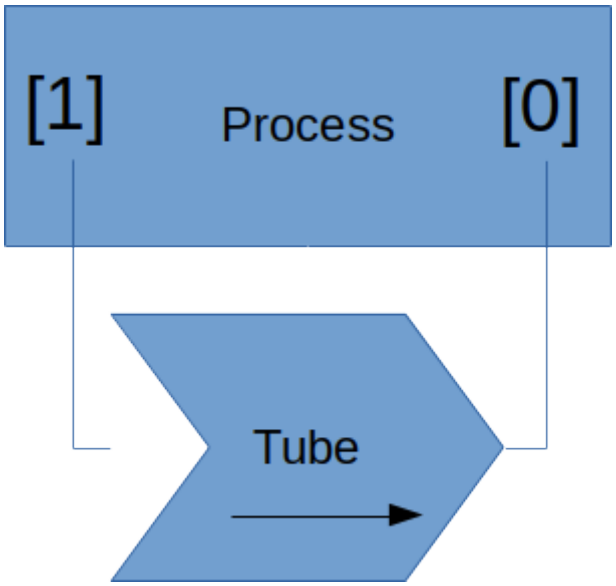

FIGURE 1 – Modèle de fonctionnement d'un tube

L'intérêt des tubes se révélera dans le cas où au moins deux processus partagent le *même* objet tube. Dans ce cas, un des processus va écrire dans l'extrémité réservée à cet usage et l'autre va lire depuis l'extrémité de lecture.

Ainsi, nous allons établir un canal directionnel de communication entre des processus différents pour échanger des octets.

Nous pouvons alors résumer la situation avec le schéma suivant:

Nous voyons dans le schéma ci-dessus que les extrémités sont traitées de façon similaire dans les deux processus.

Cependant, nous savons que le tube offre un flux directionnel. Cela veut dire qu'un processus va jouer le rôle de source des données et l'autre sera la destination des données et cela durant toute la vie du tube.

Chaque processus va explicitement prendre son rôle en fermant une des extrémités qui ne sera pas utilisée. Le processus écrivain va fermer l'extrémité de lecture et le processus lecteur va fermer l'extrémité d'écriture comme indiqué sur le schéma ci-dessous.

Cette configuration offre aussi l'avantage de gérer correctement la fin de la communication entre les processus.

En effet, une fois tous les octets lus nous pouvons conclure à la fin de la communication uniquement si aucun processus ne garde encore de descripteur d'écriture ouvert. Dans ce cas, la fonction `read` indique la fin de la communication

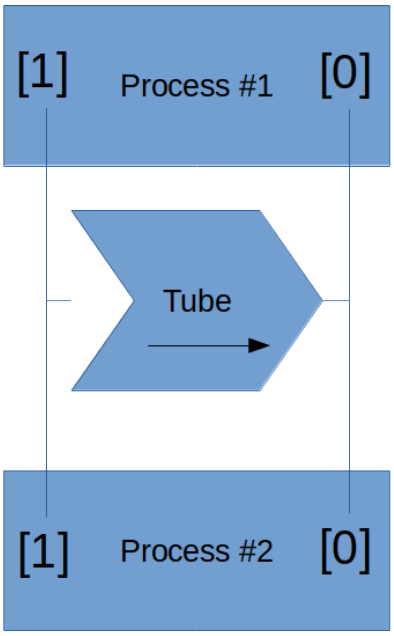

FIGURE 2 – Modèle de fonctionnement d'un tube avec deux processus

en renvoyant la valeur `EOF`.

Si par contre au moins un processus garde le descripteur d'écriture ouvert alors la fonction `read` va bloquer le lecteur en attendant une écriture éventuelle.

Nous pouvons résumer toutes ces étapes avec le code `C` suivant:

```c
int filedes[2];

/* Creation du tube */
if (pipe(filedes) == -1) errExit("pipe");

/* Creation processus fils */
switch (fork())
{
 case -1: errExit("fork");
 case 0: /* fils */
/* fermer l extremite ecriture donc je prends le role lecteur */
if (close(filedes[1]) == -1) errExit("close");
 /* le fils peut lire dans suite */
 ...
break;
```

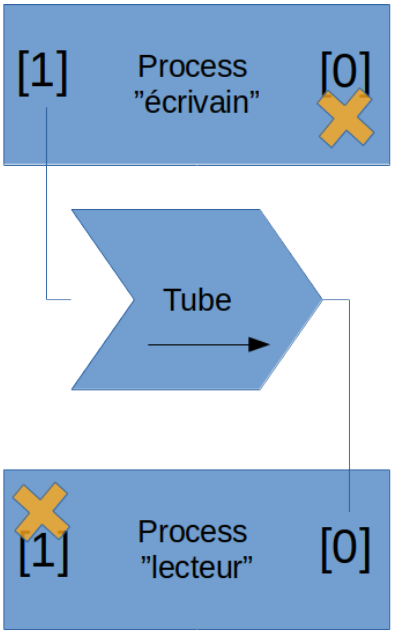

FIGURE 3 – Modèle de fonctionnement d'un tube avec deux processus qui prennent des rôles écrivain et lecteur

```
 default:
 /* Pere */
  /* fermer l extremite lecture donc je prends le role ecrivain */
 if (close(filedes[0]) == -1) errExit("close");

 /* le pere peut ecrire dans suite */
 ...
 break;
}
```

Dans ce code, après le `fork` le père et le fils se spécialisent en lecteur et écrivain sur le tube.

Pour préciser son rôle, un processus va fermer l'extrémité inutile et garder ouverte l'extrémité qui le concerne.

Avec cette configuration, la fin de la communication sera bien détectée par le lecteur une fois que toutes les données de l'écrivain reçues. En effet, puisque les processus auront fermé le descripteur d'écriture, aucune nouvelle écriture ne sera possible et `read` va retourner `EOF`.

Comment réaliser une communication bi-directionnelle ?

Nous avons bien indiqué qu'un tube est un canal de communication unidirectionnel. Si nous souhaitons établir une communication bi-directionnelle entre deux processus la solution est de créer deux tubes avec des sens opposés.

Un protocole applicatif devra ensuite spécifier comment l'échange des données se fera entre ces deux processus.

Technique de duplication des descripteur

Nous allons présenter maintenant une technique qui permet de lier les descripteurs standards d'un processus (appelés aussi les descripteurs de filtre) : entrée standard 0, sortie standard 1 et sortie d'erreur 2 avec les extrémités d'un tube.

L'objectif est de simplifier l'utilisation des tubes. Par exemple, toute écriture sur la sortie standard avec une fonction d'affichage comme `printf` sera redirigée vers le tube.

De même, toute lecture de l'entrée standard comme `scanf()` se fera à partir du tube.

Cette technique utilise la fonction `dup2(des,l_des)` qui va prendre deux paramètres et réaliser les opérations suivantes:

— l'ancien descripteur `l_des` sera fermé

— la valeur de `l_des` sera récupérée et liée maintenant au fichier identifié par `des`

Avec `dup2` nous avons la situation suivante: `des` et `old_des` identifient le même fichier alors qu'ils peuvent avoir des valeurs différentes.

Ceci nous rappelle l'idée d'un lien mais avec des descripteurs cette fois-ci.

Il est recommandé de fermer l'ancien descripteur `des` car nous sommes intéressés par `l_des` comme valeur dans la suite du programme et garder `des` ouvert peut perturber la détection de la fin de la communication.

Dans le cas des tubes, `dup2` sera utilisée de la façon suivante:

```
dup2(tube[1], 1);
```

Ce code ferme le fichier de sortie standard (lié au terminal par défaut) et associe le descripteur de valeur 1 à l'extrémité d'écriture de notre tube.

Nous pouvons ensuite fermer le descripteur `tube[1]` pour éviter le problème d'attente sur des écrivains potentiels:

```
close(tube[1]);
```

Maintenant si une fonction va écrire sur la sortie standard du processus, dont la valeur est toujours égale à 1, elle va écrire en réalité dans le tube:

```
dup2(tube[1], 1);
close(tube[1]);
printf("Bonjour à vous !");
```

Ici la fonction `printf` va écrire le message dans le tube car nous avons lié la sortie standard avec l'extrémité d'écriture du tube.

Exemple

Dans cet exemple détaillé, nous allons simulé l'éxécution de la commande suivante par le shell:

```
$ ls | wc -l
```

Un processus père va créer deux processus fils pour exécuter respectivement `ls` et `wc`.

Le processus père va créer un tube anonyme pour lier la sortie standard du processus `ls` avec l'entrée standard du processus `wc`.

La technique de duplication des descripteur est utilisée pour permettre de lancer les programmes `ls` et `wc` avec `exec` de façon transparente.

Toute écriture ou lecture sur les descripteurs de filtres sera liée directement au tube anonyme.

```c
int main(int argc, char *argv[])
{
    int pfd[2];
    //creation d un tube
    if (pipe(pfd) == -1) errExit("pipe");
    switch (fork())
    {
        case -1: errExit("fork"); // erreur sur fork

        case 0: /* premier fils va faire un exec 'ls' et ecrire sur le tube*/
        /* Prendre le role ecrivain */
        if (close(pfd[0]) == -1) errExit("close 1");
        /* Duplication de stdout on write end of pipe; fermer le descripteur duplique */
       if (pfd[1] != STDOUT_FILENO) // verification avant
       {
            if (dup2(pfd[1], STDOUT_FILENO) == -1) errExit("dup2 1");
            if (close(pfd[1]) == -1) errExit("close 2"); //fermeture de l ancien descripte
       }
       /* lancement du programme ls */
       execlp("ls", "ls", (char *) NULL);
       errExit("execlp ls"); /* erreur */
       default: break;
    }



    /* Le pere va creer un deuxieme fils */


    switch (fork())
    {
        case -1: errExit("fork");
        case 0: /* fils n 2: va faire un exec 'wc' et lire du tube */
        /* prendre un role */
        if (close(pfd[1]) == -1)  errExit("close 3");
        /* Duplication de stdin on read end of pipe; fermer le descripteur inutile */
        if (pfd[0] != STDIN_FILENO)
        {
           if (dup2(pfd[0], STDIN_FILENO) == -1) errExit("dup2 2");
           if (close(pfd[0]) == -1) errExit("erreur close");
        }
        execlp("wc", "wc", "-l", (char *) NULL);
        errExit("execlp wc"); /* Reads from pipe */
        default:
        break;
   }
```

```
        /* Parent closes unused file descriptors for pipe, and waits for children */
        if (close(pfd[0]) == -1) errExit("close 5");
        if (close(pfd[1]) == -1) errExit("close 6");
        if (wait(NULL) == -1) errExit("wait 1");
        if (wait(NULL) == -1) errExit("wait 2");
        exit(EXIT_SUCCESS);
    }
```

Les tubes processus

```
#include <stdio.h>
FILE *popen(const char * command , const char * mode );
int pclose(FILE * stream );
```

La fonction `popen` va nous permettre de réaliser la séquence d'actions suivantes:

- Le processus appelant, noté (A), va lancer un processus shell, noté (B), avec le mécanisme classique fork/exec
- Le processus shell (B) va à son tour lancer la commande `command` avec un mécanisme fork/exec dans un nouveau processus noté (C)

C'est ce dernier processus (C) qui va exécuter la commande passée en paramètre de popen.

Considérons maintenant comment les différents descripteurs standards de ces trois processus sont liés entre eux avec un tube.

Nous allons distinguer deux cas, qui sont exclusifs:

Mode R(ead)

Le premier cas est le mode de lecture avec le paramètre `mode` qui prendra la valeur `R`. Dans ce cas, un tube directionnel fera le lien entre le processus (C) et le processus (A).

La sortie standard du processus (C) sera liée à l'extrémité d'écriture; l'entrée standard du processus (A) sera elle liée à l'extrémité de lecture du tube.

Pour le processus (A) cela signifie qu'il sera capable de lire toutes les informations écrites dans la sortie standard du processus (C).

Mode W(rite)

Le deuxième cas est le mode lecture avec le paramètre `mode` qui prendra la valeur `W`. Un tube directionnel fera le lien entre le processus (A) et le processus (C). La sortie standard du processus (A) sera liée à l'extrémité d'écriture et l'entrée

standard du processus (C) sera liée à l'extrémité de lecture. Pour le processus (A) cela signifie qu'il sera capable de transmettre des informations au processus (C) qui seront lues sur son entrée standard.

Example 1

```c
#include <stdio.h>
#include <ctype.h>
#include <limits.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#define POPEN_TEMPLATE "/bin/ls -d %s 2> /dev/null"

#define MOTIF_TAILLE 50
#define PCMD_BUFFER_TAILLE (sizeof(POPEN_TEMPLATE) + MOTIF_TAILLE)

int main(int argc, char *argv[])
{
char pat[MOTIF_TAILLE];
char popenCmd[PCMD_BUFFER_TAILLE];
FILE *fp;
bool badPattern;
int len, status, fileCnt, j;
char pathname[PATH_MAX];
for (;;)
{
    // Section (1)---------//
    /* Lecture */
    printf("pattern: ");
    fflush(stdout);
    if (fgets(pat, MOTIF_TAILLE, stdin) == NULL) // EOF
        break;
    len = strlen(pat);
    if (len <= 1)
        continue; /* ligne vide */
    if (pat[len - 1] == '\n')
        pat[len - 1] = '\0'; // retirer retour chariot
    // petite verification que le motif contient que les bons caracteres
    for (j = 0, badPattern = false; j < len && !badPattern; j++)
    {
        badPattern = !isalnum((unsigned char)pat[j]) && strchr("_*?[^-].", pat[j]) == NULL;
    }
    if (badPattern)
```

```
    {
        printf("Bad pattern character: %c\n", pat[j - 1]);
        continue;
    } /* Construction de la commande en utilisant le motif 'pat' */
    snprintf(popenCmd, PCMD_BUFFER_TAILLE, POPEN_TEMPLATE, pat);
    popenCmd[PCMD_BUFFER_TAILLE - 1] = '\0'; // terminer le string en C
    //-------------------//
    // Section (2)---------//
    // ouverture du tube processus
    // deux process sont créé : shell et process de notre commande
    // nous voyons uniquement le flux issu du tube
    fp = popen(popenCmd, "r"); // mode lecture
    if (fp == NULL)
    {
        printf("popen() echec\n");
        continue;
    }
    //-------------------//
    // Section (3)---------//
    /* lecture du resultat*/
    fileCnt = 0;
    while (fgets(pathname, PATH_MAX, fp) != NULL)
    {
        printf("%s", pathname);
        fileCnt++;
    }
    /* fermer le pipe */
    status = pclose(fp);
    printf("%d ficher%s trouvé%s\n", fileCnt, (fileCnt != 1) ? "s" : "", (fileCnt != 1) ? "s
    printf("pclose() status == %#x\n", (unsigned int)status);
    //-------------------//
}
exit(0);
}
```

La section (1) de ce programme prépare la commande à lancer avec `popen`.

Nous récupérons un motif saisi par l'utilisateur et nous construisons ensuite la commande complète.

La commande sera ensuite lancée par `popen` qui sera en mode `R`. Le résultat de cette commande est ensuite récupéré par lecture sur le flux `fp`.

Les tupes nommés

Introduction

Un tube nommé, ou FIFO, est un tube que nous pouvons identifier. Cela veut dire que deux processus qui partagent le nom de la FIFO et qui n'ont aucun lien de parenté, peuvent l'ouvrir et communiquer.

Une fois que le tube nommé a été ouvert, nous pouvons l'utiliser comme un simple fichier avec les opérations de lecture, écriture et fermeture.

Les tubes nommés sont des FIFO pour mettre en exergue leur mode de transmission des octets en flux qui respecte l'ordre d'écriture. First In First Out nous indique donc que le premier octet entré dans le tube sera le premier à en sortir.

Création de tubes nommés

Méthode par le shell

La première méthode pour créer un tube nommé est d'utiliser la commande shell `mkfifo`

```
mkfifo /tmp/montube
```

Cette commande va créer le tube `montube` dans le répertoire `/tmp`

Vous remarquerez que le chemin du système de fichier sera un moyen très pratique pour organiser vos tubes.

De plus, la commande `ls` va permettre de lister tous les tubes se trouvant sur un répertoire.

Pour montrer qu'il s'agit d'un tube la commande `ls` va mettre un `p` (pour 'pipe') avant de lister les droits associés:

```
prw-rw-r-- 1 gouaich gouaich /tmp/montube
```

Méthode par API C

```
#include <sys/stat.h>
int mkfifo(const char * pathname , mode_t mode );
```

La fonction `mkfifo()` permet de créer une fifo à partir du code C. Le mode va spécifier les droits associés à la fifo comme pour un fichier standard.

Utilisation d'un tube nommé

Ouverture bloquante

Nous devons spécifier notre rôle par rapport à la FIFO à l'ouverture. Cela est possible en spécifiant le mode d'ouverture soit en lecture ou écriture avec les flags `O_RDONLY` et `O_WRONLY`.

Contrairement à un fichier standard où l'ouverture n'est pas bloquante; l'ouverture d'une FIFO permet de synchroniser le lecteur et l'écrivain.

L'ouverture d'une FIFO en mode lecture avec `O_RDONLY` va donc bloquer le processus qui en fait la demande jusqu'à la présentation d'un processus qui va ouvrir la FIFO en mode écriture avec le flag `O_WRONLY`.

De façon similaire, l'ouverture d'une FIFO en mode écriture avec le flag `O_WRONLY` sera bloquante jusqu'à la présentation d'un processus qui ouvrir la FIFO en lecture avec le flag `O_RDONLY`.

Ouverture non bloquante

Si nous ne souhaitons pas être bloqué par le mécanisme de synchronisation de la FIFO, nous pouvons utiliser le flag `O_NONBLOCK`. Dans ce cas la fonction `open` retourne directement le descripteur de la FIFO.

Exemple d'utilisation de fifo

La commande `wc` avec l'argument `-l` permet de comptabiliser le nombre de lignes dans un flux texte.

```
$wc -l
>premiere
>deuxieme
>troisieme
^D
$3
```

Nous allons créer une FIFO nommée `testwc` sous le répertoire /tmp:

```
$mkfifo /tmp/testwc`
```

Nous lançons ensuite le processus `wc` en background avec une redirection de son entrée standard vers la FIFO:

```
$wc -l < /tmp/testwc &
```

Nous notons que ce processus est en cours d'exécution dans le groupe background. Seulement, il est bloqué en attente d'un processus qui va ouvrir la FIFO en mode écriture et transmettre des données.

```
$ls -al > /tmp/testwc
```

Le processus `ls` va écrire son résultat sur la FIFO. Le texte sera récupéré par le processus `wc` qui attend en background et qui affichera le nombre de lignes.