Programmation système : redirections, tubes

Juliusz Chroboczek

16 Décembre 2009

1 Descripteurs standard

Si tous les descripteurs de fichiers sont identiques du point de vue du noyau, les descripteurs de fichier 0, 1 et 2 ont un rôle conventionnel :

- le descripteur de fichier 0 s'appelle l'*entrée standard*, et il sert à fournir l'entrée à un processus qui ne lit qu'à un seul endroit ;
- le descripteur de fichier 1 s'appelle la *sortie standard*, et il sert de destination pour la sortie d'un processus qui n'écrit qu'à un seul endroit ;
- le descripteur de fichier 2 s'appelle la *sortie d'erreur standard*, et il sert de destination pour les messages d'erreur.

Comme tous les descripteurs de fichiers, les descripteurs dits *standard* sont hérités du père. Normalement, ils ont été associés au terminal par un parent, et permettent donc de lire et d'écrire sur le terminal.

2 Redirections

Plutôt que de forcer tous les programmes à implémenter la sortie sur le terminal, vers un fichier, vers l'imprimante etc., Unix permet de *rediriger* les descripteurs standard d'un processus vers un descripteur arbitraire, par exemple un fichier ouvert auparavant.

Une redirection se fait à l'aide de l'appel `dup2`, qui copie un descripteur de fichier :

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

Un appel à `dup2` copie le descripteur `oldfd` en `newfd`, en fermant celui-ci auparavant s'il était ouvert ; en cas de succès, il retourne `newfd`. Après un appel à `dup2`, les entrées `oldfd` et `newfd` de la table de descripteurs de fichiers pointent sur la même entrée de la table de fichiers ouverts — le pointeur de position courante est donc partagé.

L'appel système `dup` est une version obsolète de `dup2` :

```
int dup(int oldfd);
```

Un appel à `dup` copie le descripteur de fichier `oldfd` vers la première entrée libre de la table de descripteurs de fichier, et retourne le numéro de cette dernière.

3 Tubes

Comme nous l'avons vu en cours, la communication entre deux processus peut se faire à travers un fichier. Une telle approche demande un mécanisme supplémentaire de synchronisation (par exemple `wait`), et risque de laisser des fichiers temporaires sur le disque.

Un *tube* consiste de deux descripteurs de fichier, appelés le *bout écriture* et le *bout lecture* du tube, connectés à travers un tampon. Une donnée écrite sur le bout écriture est stockée dans le tampon, et une lecture retourne les données contenues dans le tampon.

Un tube est créé à l'aide de l'appel système `pipe` :

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

L'appel système `pipe` crée un nouveau tube. Son bout écriture est stocké en `fd[1]`, et son bout lecture est stocké en `fd[0]`. La taille du tampon interne du tube n'est pas spécifiée, mais POSIX garantit qu'elle fait au moins 8192 octets.

Sémantique des entrées-sorties sur les tubes Les lectures sur les tubes ont la sémantique suivante :

- un appel système `read` peut retourner un nombre d'octets strictement positif mais inférieur à celui demandé (on parle alors de *lecture partielle*) ;
- un appel système `read` retourne 0 (indication de fin de fichier) si l'écrivain a fermé le tube et celui-ci est vide ;
- un appel système `read` effectué sur un tube vide bloque.

Dualement, les écritures ont la sémantique suivante :

- un appel système `write` peut écrire un nombre d'octets strictement positif mais inférieur à celui demandé (on parle alors d'*écriture partielle*) ;
- un appel système `write` effectué sur un tube que le lecteur a fermé tue le processus avec un signal `SIGPIPE`[1] ;
- un appel système `write` effectué sur un tube plein bloque.

Cette sémantique est plus relâchée que celle des entrées-sorties sur disque, où les entrées/sorties ne bloquent jamais, où une lecture partielle n'est possible qu'à la fin du fichier, et où une écriture partielle est impossible. En pratique, cela signifie qu'on ne pourra pas utiliser un tampon simple avec les tubes, il faudra utiliser un tampon avec pointeur de début ou un tampon circulaire.

4 Exemple

Le fragment de code suivant combine tous les appels système vus au cours des deux derniers cours.

```
int fd[2], rc;
pid_t pid;

rc = pipe(fd);
```

1 Ce qu'on peut éviter en ignorant `SIGPIPE` ; dans ce cas, le `write` retourne -1 avec `errno` valant `EPIPE`.

```
if(rc < 0) ...

pid = fork();
if(pid < 0) ...

if(pid == 0) {
    close(fd[0]);
    rc = dup2(fd[1], 1);
    if(rc < 0) ...
    execlp("whoami", "whoami", NULL);
    ...
} else {
    char buf[101];          /* Pourquoi 101 ? */
    int buf_end = 0;
    close(fd[1]);
    while(buf_end < 100) {
        rc = read(fd[0], buf + buf_end, 100 - buf_end);
        if(rc < 0) ...
        if(rc == 0)
            break;
        buf_end += rc;
    }

    if(buf_end > 0 && buf[buf_end - 1] == '\n')
        buf_end--;
    buf[buf_end] = '\0';

    printf("Je suis %s\n", buf);

    wait(NULL);             /* Pourquoi faire ? */
}
```