CHAPITRE 6

POSIX.1 : Redirection des flux d'entrée-sortie et communication entre processus par tubes

6.1 Objectif

Ces exercices mettent en œuvre les mécanismes de redirection de Posix (primitives `fcntl()`, `dup()`, `dup2()`), ainsi que la communication de données par tubes (primitive `pipe()`). La première partie 6.2 est constituée de deux exercices simples et courts. La seconde 6.3 est plus ambitieuse (mais aussi plus intéressante ?).

6.2 Pour se mettre en jambes

6.2.1 Redirections du shell

Créez un répertoire, et allez-y. Créez-y trois fichiers `A`, `B` et `out` (peu importe leur contenu). Si vous utilisez **zsh** (**bash** doit donner des résultats à peu près identiques), l'exécution de **ls** doit à peu près vous dire :

```
kheops% ls
A   B   out
kheops%
```

(**kheops%** est mon propre prompt de zsh.) Maintenant, exécutez ls avec une liste de fichiers dont certains existent dans le répertoire et d'autres pas. Par exemple

```
kheops% ls * foo bar
A   B   out
ls: foo: No such file or directory
ls: bar: No such file or directory
kheops%
```

Les messages d'erreur (les deux dernières lignes) sont affichés sur le flux d'erreur standard (`stderr`, descripteur de fichier 2) alors que la liste des fichiers va sur la sortie standard

(`stdout`, descripteur de fichier 1). Essayons de rediriger ces deux flux dans un même fichier, `out` [1] :

```
ls * foo bar >! out 2>! out
```

Visualisez le contenu de out. Que constatez-vous ? Quelle explication pourriez-vous donner ? En fait, la bonne commande si l'on veut avoir les deux flux redirigés sans conflit sur le même fichier, est :

```
ls * foo bar >! out 2>&1
```

La syntaxe spéciale `2>&1` fait du descripteur de fichier 2 un synonyme du descripteur 1 (écrire sur 1 ou 2, c'est pareil). Comme 1 a lui-même été redirigé sur `out` juste avant, 2 désigne aussi `out`, avec partage du pointeur d'entrée-sortie.

Écrivez un programme, disons `tst_redirect`, qui simule le comportement du **shell** lorsqu'il exécute la commande précédente. Ne trichez pas ! Vous devez utiliser ici `fork()` et `execXX()`, pas la fonction `system()` (ni celle de base, ni la votre développée en 5.5.2).

6.2.2 Tubes du shell

Écrire un programme, `tst_pipes`, qui simule la commande au **shell** suivante :

```
ps | grep emacs | wc -l
```

En écrivant ce programme, prenez grand soin de fermer, dans tous les processus, les descripteurs de fichiers inutilisés, en particulier ceux des tubes. Mais il est aussi intéressant de faire l'expérience de ne pas les fermer : que se passe-t-il alors ? pourquoi ?

6.3 ANNEAU À JETON

Les réseaux à jeton et en anneau sont un moyen simple de synchronisation entre plusieurs stations opérant en parallèle. Plusieurs réseaux commerciaux reposent sur ce type de protocole. Nous allons ici réaliser un simulateur d'un tel anneau.

6.3.1 Structure de l'anneau (`ring`)

Un réseau en anneau a la structure représentée sur la figure 6.1. Il est composé de N stations $S_0, S_1, ..., S_{N-1}$ ($N = 5$ sur la figure). Pour nous, une station correspondra à l'exécution d'un fichier binaire indépendant, nommé `station`, qui prend un unique argument, son numéro d'ordre ($i = 0, 1, ..., N - 1$). Les stations sont conectées à leurs deux voisines par l'intermédiaire de N tubes (p_i sur la figure). Le flux de communication dans les tubes est orienté comme indiqué sur la figure et les tubes sont branchés sur l'entrée et la sortie standard de chaque station. Toutes les stations sont (potentiellement [2]) actives en parallèle.

Vous en savez assez pour écrire le programme, disons `ring`, qui met en place cette structure de processus et les connexions correspondantes. Ce programme recevra un unique

1. Le point d'exclamation dans la commande qui suit, permet d'éviter à **zsh** de râler quand on tente d'écraser un fichier existant.

2. En effet vous constaterez plus loin que le protocole choisi confère une stricte séquentialité au fonctionnement de ce système parallèle.

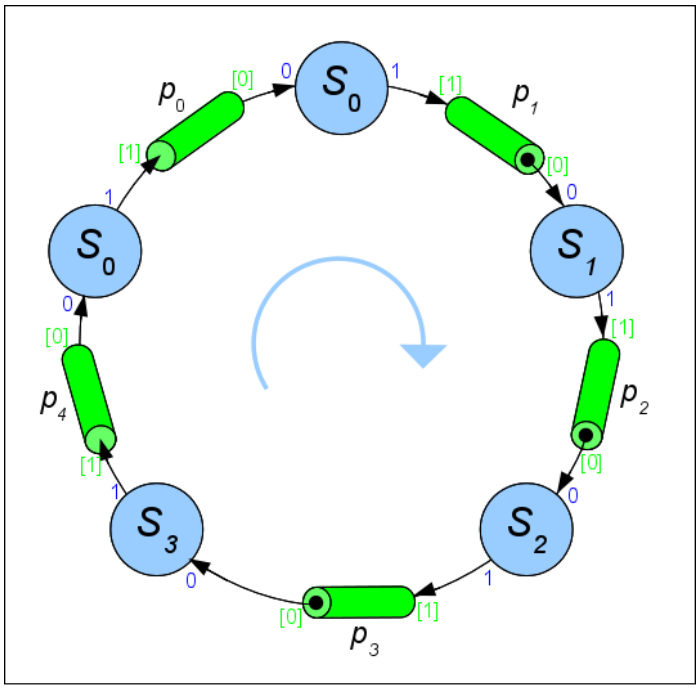

FIGURE 6.1 – Schéma de l'anneau à jeton (pour $N = 5$).

argument : le nombre N de stations de l'anneau (en pratique, moins d'une dizaine). Pour tester votre programme, vous pouvez vous contenter d'une version de `station` où l'on recopie sur la sortie standard ce qu'on a lu sur l'entrée (tout en l'affichant sur l'erreur standard, `stderr`).

> **Suggestion... et même un peu plus !**
>
> Le programme `ring` va bien sûr créer N processus fils, un pour chaque station. Soyez vigilants sur les deux points suivants :
> — le processus père doit impérativement attendre la fin de tous ses fils (`wait()`) ;
> — les processus fils et le père lui-même doivent fermer (`close()`) tous les descripteurs de fichiers qui leur sont inutiles (en particulier les descripteurs des tubes).
>
> Si vous ne respectez pas ces deux points, vous risquez d'avoir quelques problèmes pour arrêter votre réseau (en envoyant `^C` à `ring`, par exemple).

6.3.2 Données circulant sur l'anneau (paquet)

Sur l'anneau tout entier circule un seul paquet à la fois. Ce paquet unique est composé des éléments suivants (figure 6.2) :

— un jeton, qui a trois valeurs possibles : `FREE` (libre), `ACK` (accusé de réception, et `BUSY` (occupé) ;
— les numéros (de 0 à $N - 1$) des stations émettrice et réceptrice du message ;
— le message lui-même, une chaîne de caractères quelconque de longueur fixe `MSG_LEN`.

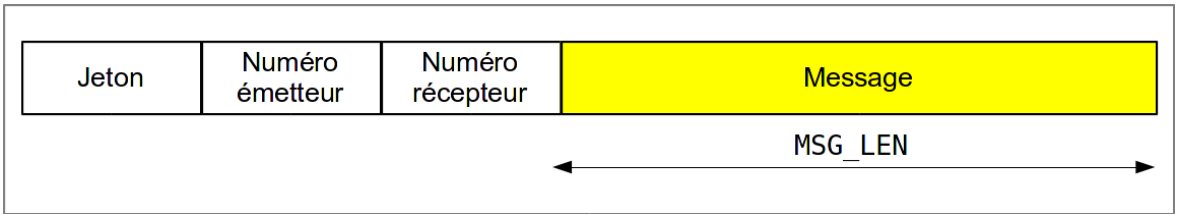

FIGURE 6.2 – Schéma du paquet circulant sur l'anneau.

6.3.3 Protocole de l'anneau (code de `station`)

Le principe du protocole de l'anneau à jeton est très simple, dans les grandes lignes. Quand une station voit passer le paquet avec un jeton `FREE`, elle peut lui attacher un message, marquer le jeton `BUSY` et propager le paquet. Quand la station destinatrice reçoit le paquet, elle le « consomme », marque le jeton en accusé de réception (`ACK`) et le propage à nouveau. Le paquet va donc faire un tour complet pour revenir à son émetteur qui prendra acte de l'`ACK`, remettra le jeton dans l'état `FREE`, et passera le paquet à la station suivante. Bien entendu, ceci ne fonctionne que si chaque station propage à sa voisine les paquets qui ne la concernent pas. En détails, chaque station déroule l'algorithme suivant :

1. La station se met en attente (bloquante) du paquet sur son entrée standard ;
2. Quand le paquet arrive, la station prépare une nouvelle configuration du paquet de la manière suivante :
   (a) Si le jeton est `FREE`, cela signifie qu'il n'y a pas de message valide. La station engage un dialogue au terminal avec l'utilisateur, en lui demandant si il a un message à transmettre ? si oui, lequel et à quelle station ?
   Si l'utilisateur veut envoyer un message, la station prépare un paquet en marquant le jeton `BUSY`, en renseignant les numéros des stations et en y copiant le message. En revanche si l'utilisateur ne souhaite pas envoyer de message, le paquet reçu n'est pas modifié.
   (b) Si le jeton n'est pas `FREE`, cela signifie que le paquet contient un message valide. Il y a alors plusieurs cas, dont les deux premiers ne sont pas forcément exclusifs [3] :
   — la station courante est la destinatrice de ce message : dans ce cas, elle marque le jeton en `ACK` ;
   — la station courante était l'émettrice du message : dans ce cas, si le jeton est un `ACK`, cela signifie que son message a été reçu cor- rectement par son destinataire et que le paquet a fait un tour complet ; la station remet alors le jeton dans l'état `FREE` ;
   — Si aucune des deux conditions précédentes n'est vraie, le paquet est inchangé.
3. Dans tous les cas, le nouveau paquet élaboré précédemment est envoyé sur la sortie standard, vers la station suivante.

6.3.4 Initialisation de l'anneau

Il faut bien qu'une station envoie le premier paquet (`FREE`) sur le réseau. Nous supposerons ici que c'est le rôle de la station de numéro 0.

3. En particulier, vous devrez traiter correctement le cas où une station s'envoie un message à elle-même.

Dans un véritable anneau à jeton, l'initialisation et la regénération du jeton en cas de perte, sont beaucoup plus délicates.

6.3.5 Réalisation de `station`

Maintenant, votre mission est bien entendu d'écrire `station` conformément aux spécifications précédentes, et de tester votre réseau en mettant ensemble `ring` et cette version complète de `station`.

Dialoque avec l'utilisateur

L'entrée et la sortie standard de chaque station sont occupées à assurer la connectivité du réseau. Elles ne peuvent donc plus être utilisées pour dialoguer au terminal ! Il reste l'erreur standard (`stderr`) mais si on peut y écrire, il n'est pas conseillé d'y lire.

Pour dialoguer avec l'utilisateur, chaque station ouvrira donc, par `fopen()`, le périphérique nommé `/dev/tty`, le *terminal de contrôle* (en fait le terminal tout court, pour vous !). Attention, pour éviter les problèmes, il convient d'ouvrir ce terminal deux fois, une fois en entrée, une autre fois en sortie. Comme vous utilisez `fopen()`, vous obtenez en retour des `FILE *`, qui sont directement utilisables avec `fprintf()` et `fscanf()`.

Traces du fonctionnement de la station

Vous avez intérêt à tracer le fonctionnement de vos stations. Les traces doivent correspondre aux différents cas énumérés ci-dessus. N'en mettez cependant pas trop ! Les messages de trace peuvent être envoyés sur `stderr`, ou sur le flux de sortie associé à `/dev/tty` mentionné ci-dessus.