Tubes (Pipes)

Abdelkader Gouaïch

IUT de Montpellier

2014-2015

Les objectifs du cours sont :

1. Définition des tubes (pipes) :
   - tubes-processus
   - tubes-anonymes
   - tubes-nommés
2. Maitriser les fonctions de gestion des tubes
3. Maitriser les techniques :
   - pipe/fork
   - pipe/fork-exec

Introduction

- Nous avons présenté dans le chapitre sur les signaux un mécanisme simple de communication entre processus
- Il s'agit de transférer une information codée sous forme d'un entier (le numéro du signal)
- Dans ce chapitre nous allons voir un mécanisme de communication plus élaboré entre processus.

Définition d'un tube (pipe)

- Un tube (ou pipe en anglais) est un mécanisme qui permet le transfert de données entre deux processus
- Le tube permet la circulation des données entre deux espaces d'adressage indépendants (les deux processus)
- Nous avons déjà rencontré cette notion avec l'opérateur `|` (Chapitre : Shell)

Les tubes processus (Process Pipe)

- Pour créer des tubes-processus on utilise les fonctions `popen` et `pclose`
- Avec ces fonctions on peut transférer facilement des données entre processus

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

- cmd : la commande qui sera lancée
- mode : le mode d'ouverture du tube-processus : "r" ou "w"

popen

```
#include <stdio.h>
FILE *popen(const char *cmd, const char * mode);
```

`popen` permet

- de lancer un autre programme comme un nouveau processus
- de transférer les données entre les processus (i) initiateur et (ii) nouvellement créé

les modes d'ouverture des flux

- Si le mode est "r" :
  - Les sorties du processus nouvellement créé sont disponibles à la lecture au processus initiateur
  - On utilise la fonction `fread` pour lire sur flux renvoyé par la fonction `popen`
- Si le mode est "w" :
  - Le processus initiateur peut envoyer des données au nouveau processus en écrivant dans le flux
  - Pour écrire dans le flux on utilise la fonction `fwrite`
  - Le nouveau processus trouvera les données en lisant sur l'entrée standard (`stdin`)
- On remarque que le mode est soit ≪r≫ soit ≪w≫
  - Cela signifie qu'on ne peut pas écrire *et* lire dans le flux retourné par `popen`

pclose

```
#include <stdio.h>
int pclose(FILE *flux);
```

- Quand le processus nouvellement créé se termine on peut fermer le flux avec la fonction `pclose`
- La fonction `pclose` est bloquante et retourne quand le processus créé se termine.

Exemple 1

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

#define BUFSIZ 1024

int main()
{
FILE *lecture_flux;
char buffer[BUFSIZ + 1];
int nbr_chars;
memset(buffer, '\0', sizeof(buffer));
lecture_flux = popen(``uname -a'', ``r'');
if (lecture_flux != NULL) {
nbr_chars = fread(buffer, sizeof(char), BUFSIZ, lecture_flux);
if (nbr_chars > 0) {
printf(``La sortie du programme etait:\n%s\n'', buffer);
}
pclose(lecture_flux);
exit(EXIT_SUCCESS);
}
exit(EXIT_FAILURE);
}
```

Exemple 2

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

#define BUFSIZ 1024
int main()
{
FILE *ecriture_flux;
char buffer[BUFSIZ + 1];
sprintf(buffer, "Ceci est ma phrase...\n");
ecriture_flux = popen("od -c", "w");
if (ecriture_flux != NULL)
{
fwrite(buffer, sizeof(char), strlen(buffer), ecriture_flux);
pclose(ecriture_flux);
exit(EXIT_SUCCESS);
}
else
{
exit(EXIT_FAILURE);
}
}
```

Exemple 3 : Lecture de données volumineuses

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

#define BUFSIZ 100
int main()
{
FILE *read_flux;
char buffer[BUFSIZ + 1];
int chars_read;
memset(buffer,'\0', sizeof(buffer));
read_fp = popen(``ps ax'', ``r'');
if (read_flux != NULL) {
chars_read = fread(buffer, sizeof(char), BUFSIZ, read_flux);
while (chars_read > 0)
{
buffer[chars_read -1]= '\0';
printf(``Lecture de: %s\n'', buffer);
chars_read = fread(buffer, sizeof(char), BUFSIZ, read_flux);
}
pclose(read_flux);
exit(EXIT_SUCCESS);
}
else
{
exit(EXIT_FAILURE);
}
}
```

Comment popen est réalisée ?

- Popen exécute la commande `cmd` en utilisant le Shell
- A chaque appel de `popen` un nouveau processus est créé avec :
  - le Shell comme programme
  - comme argument la commande `cmd`

Les tubes anonymes

- Nous allons étudier des fonctions de gestion de tubes sans invoquer le Shell pour interpréter une commande
- Ces tubes sont appelés des tubes anonymes et permettent de contrôler les transferts de données entre les processus

la fonction `pipe`

Le prototype de la fonction qui permet de créer un tube anonyme est :

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

- La fonction `pipe` attend en paramètre un tableau d'entiers contenant deux cases
- Cette fonction va remplir les deux cases avec :
  - la case d'indice 0 : le descripteur d'un fichier utilisé pour lire les données du tube
  - la case d'indice 1 : le descripteur d'un fichier utilisé pour écrire les données dans le tube

Les descripteurs du tube anonyme

- Evidement, les deux descripteurs de fichiers retournés par `pipe` sont connectés de sorte que :
  - Chaque donnée écrite dans la case d'indice 1 (*descripteurs*[1])
  - Sera lue en utilisant le descripteur de la case d'indice 0 (*descripteurs*[0])
- L'ordre de lectures/écritures est FIFO (First In First Out)

Remarque

- Attention : Il faut remarquer que la fonction `pipe` retourne des descripteurs de fichiers et non pas des flux
- Il faut dans ce cas utiliser les fonctions E/S de bas niveau comme `read`, `write` et `close`

Exemple

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
int nbre;
int file_pipes[2];
const char donnees[] = "TEST";
char buffer[BUFSIZ + 1];
memset(buffer, '\0', sizeof(buffer));
if (pipe(file_pipes) == 0) {
nbre = write(file_pipes[1], donnees, strlen(donnees));
printf("Ecriture de %d bytes\n", data_processed);
nbre = read(file_pipes[0], buffer, BUFSIZ);
printf("Lecture de %d bytes: %s\n", nbre, buffer);
exit(EXIT_SUCCESS);
}
exit(EXIT_FAILURE);
}
```

Introduction

- L'exemple précédant n'illustre pas l'utilité du pipe
- Pourquoi ?

Introduction

- Nous avons simplement un seul processus
- Ce processus écrit et lit dans son propre tube
- Notre but est de réaliser de IPC

Introduction

- Il nous faut créer un autre processus et établir une communication avec celui-ci
- Comment créer un processus ?

Exemple

```
int main(){
int nbre; pid_t resultat_fork;
int file_pipes[2];
const char donnees[] = "TEST";
char buffer[BUFSIZ + 1];
memset(buffer, '\0', sizeof(buffer));
if (pipe(file_pipes) == 0){
resultat_fork = fork();
if(resultat_fork == -1)//cas erreur
{
exit(EXIT_FAILURE);
}
if(resultat_fork == 0)//cas fils
{
nbre = read(file_pipes[0], buffer, BUFSIZ);
printf("Lecture de %d bytes: %s\n", nbre, buffer);
exit(EXIT_SUCCESS);
}
else //cas pere
{
nbre = write(file_pipes[1], donnees, strlen(donnees));
printf("Ecriture de %d bytes\n", data_processed);
exit(EXIT_SUCCESS);
}
}
exit(EXIT_FAILURE);
}
```

Que fait ce programme ?

- Le programme commence par créer le tube avec la fonction `pipe`
- Après l'appel de `pipe` le tableau contient les valeurs des deux descripteurs de lecture et d'écriture du tube
- On fait appel à `fork` pour créer un processus fils par duplication d'image
- Attention : avec le `fork` les variables sont copiées ; et donc le processus fils aura les mêmes descripteurs du tube.
- Le fils fait une lecture sur le pipe.
- Le père fait une écriture dans le pipe.

fork-exec

- Avec `fork` le partage des valeurs des descripteurs du tube était assez simple.
- Comment partager ces valeurs en utilisant la combinaison `fork` / `exec` ?

fork-exec

```
int main()
{
int nbre;pid_t resultat_fork;
int file_pipes[2];
const char donnees[] = "TEST";
char buffer[BUFSIZ + 1];
memset(buffer, '\0', sizeof(buffer));
if (pipe(file_pipes) == 0){
resultat_fork = fork();
if(resultat_fork == -1)//cas erreur
{
exit(EXIT_FAILURE);
}
if(resultat_fork == 0)//cas fils
{ //recouvrement
sprintf(buffer, ''%d'', file_pipes[0]); //NEW
(void)execl(''pipe_prg'', ''pipe_prg'', buffer, (char *)0); //NEW
exit(EXIT_FAILURE); //NEW
}
else //cas pere
{
nbre = write(file_pipes[1], donnees, strlen(donnees));
printf(''%d - ecriture de %d bytes\n'', getpid(), nbre);//NEW
exit(EXIT_SUCCESS);
}
}
exit(EXIT_FAILURE);
}
```

pipe_prg.c

```
//pipe_prg.c
int main(int argc, char *argv[])
{
int n;
char buffer[BUFSIZ + 1];
int file_descriptor;
memset(buffer, '\0', sizeof(buffer));
sscanf(argv[1], ''%d'', &file_descriptor);
n = read(file_descriptor, buffer, BUFSIZ);
printf(''%d - lecture de %d bytes: %s\n'', getpid(), n, buffer);
exit(EXIT_SUCCESS);
}
```

Lecture des données

- Généralement, un programme ne connaît pas la taille des données à lire.
- La solution est de faire une boucle :
  - lecture des données dans une mémoire/variable tampon
  - traitement des données (avec stockage éventuel)
- jusqu'à épuisement des données à traiter

Fermeture d'un tube en écriture

- Un appel à `read` bloque le processus appelant en attendant la disponibilité de données
- Dans le cas d'un tube, que se passe-t-il si le descripteur d'écriture du tube a été fermé ?

Fermeture d'un tube en écriture

- Le processus qui demande la lecture avec `read` restera bloqué...
- C'est pourquoi, l'opération `read` sur un tube *qui n'est pas ouvert pour l'écriture* ne bloque pas et retourne la valeur 0.
- Ceci va permettre au processus qui lit sur le tube de détecter la fermeture de celui-ci.
- C'est "l'équivalent" de la détection de la fin d'un fichier (EOF) standard pour le processus qui fait un `read`.

Fermeture d'un tube en écriture

- Quel enseignement tirer ?
- Il est très important de fermer les extrémités des tubes non utilisées.
- Sinon le processus en lecture ne peut pas détecter la fin de la session de communication et bloquer sur la lecture

Fermeture d'un tube en écriture

- Exemple du `fork`
- dans le cas du `fork`, il faut que le père et le fils ferment le descripteur d'écriture pour que le tube soit considéré comme fermé.

Les tubes nommés

- Jusqu'à présent nous avons créé des tubes entre des processus qui partagent un ancêtre commun.
- Comment faire partager des données entre des processus complètement indépendants ?

Les tubes nommés

- Jusqu'à présent nous avons créé des tubes entre des processus qui partagent un ancêtre commun.
- Comment faire partager des données entre des processus complètement indépendants ?
- On fait cela en utilisant des tubes nommés (FIFOs)

La commande `mkfifo`

- On peut créer un tube nommé à partir du shell par la commande `mkfifo` *nom*
- On dispose également d'une fonction à utiliser à partir d'un programme C :

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

Accès à un tube nommé

- Ce qui est intéressant avec les tubes nommés c'est qu'on peut les utiliser comme des fichiers 'normaux'
- `cat < /tmp/mafifo`
- `echo ''Bonjour'' > /tmp/mafifo`

Accès à un tube nommé en C

- Une restriction : on ne peut pas ouvrir une FIFO en lecture *et* en écriture
- L'autre différence réside dans l'utilisation du flag : `O_NONBLOCK`

Accès à un tube nommé en C

- `open(const char *path, O_RDONLY);`
- `open` va bloquer et retournera quand un autre processus va ouvrir la FIFO en écriture.

Accès à un tube nommé en C

- `open(const char *path, O_RDONLY | O_NONBLOCK);`
- `open` va retourner immédiatement même si aucun autre processus n'a ouvert la FIFO en écriture.

Accès à un tube nommé en C

- `open(const char *path, O_WRONLY);`
- `open` va bloquer et retournera quand un autre processus va ouvrir la FIFO en lecture.

Accès à un tube nommé en C

- `open(const char *path, O_WRONLY | O_NONBLOCK);`
- `open` va retourner immédiatement même si aucun autre processus n'a ouvert la FIFO en lecture.
- Attention : Si aucun processus n'a ouvert la FIFO en lecture `open` retourne -1 et la FIFO ne sera pas ouverte.