PS 02 Programmation Système (Mini-Shell)

A faire TD 3 et TD 4.

Table des matières

1 Résumé fork/wait

Dédoubler un processus

- `pid = fork()`

Attendre la fin d'un processus

- `waitpid(pid, &status, options)`
- `pid = wait(&status)`

Consultation du code de retour

- `retcode = WEXITSTATUS(status)`

2 Les appels "exec"

2.1 Exemple "execl"

```
if (fork()==0) {
    execl("/bin/ls", "ls", "-l", NULL);
}
int s;
wait( &s );
```

Déroulement du processus fils :

- charge et exécute le programme `/bin/ls`,
- "`ls` et"`-l`" dans `argv[0]` et `argv[1]`,
- 2 dans `argc`

2.2 Fonctionnement de "exec"

- Le code du programme chargé **remplace** celui du processus
- quand le programme s'arrête, fin du processus
- la valeur transmise par `exit()` est le code de retour du processus fils

2.3 La famille exec

- Par commodité, plusieurs fonctions avec un rôle similaire
  - execl, execv, execvl, execvp
- Paramètres transmis
  - sous forme d'une liste : execl
  - sous forme d'un tableau : execv
- Programme indiqué
  - par un chemin absolu
  - par un chemin relatif (résolu par le PATH)

2.4 Exemples

Liste de paramètres terminée par NULL

```
execl ("/bin/ls", "ls", "-l", NULL);
execlp(     "ls", "ls", "-l", NULL);
```

Tableau de paramètres

```
char * arg[] = { "ls", "-l", NULL};

execv ("/bin/ls", arg);
execvp(     "ls", arg);
```

3 Application dans le shell

3.1 Lancement de commandes par le shell

Intégrer le lancement direct d'un programme par le shell

```
si premier mot est exit, help, cd ... (commande interne)
   l'exécuter
sinon {
   faire un fork
     - le fils construit un tableau d'arguments
       et appelle execvp(premier mot, tableau)
     - le père attend la fin (waitpid)
}
```

3.2 Lancement de commandes en arrière plan

Si le dernier "mot" de la commande est "&"

- ne pas attendre le processus fils
- afficher son numéro de pid

```
12> emacs truc.txt &
[1234] emacs truc.txt
13>
```

Ajouter aussi une commande `wait<pid>` qui se bloque en attendant la fin d'un processus

3.3 Gestion des travaux

Ajouter une table de correspondance entre

- numéros des processus fils lancés en arrière-plan
- commande correspondante

La commande "`jobs`" permettra de faire afficher cette table

```
18>jobs
[1234] emacs truc.txt
[1236] firefox
19>
```

3.4 Gestion des travaux (suite)

- **Objectif** : quand un processus fils se termine,
  - prévenir l'utilisateur
  - mettre à jour la table des "jobs"
- **Moyen** : quand un processus se termine
  - le système envoie un **signal** `SIGCHLD` à son père
  - le père peut associer une action à ce signal

Appel système

```
signal(SIGCHLD, une_fonction);
```

3.5 Exemple

```
void fin_fils(int sig) {
   pid_t p = wait(nullptr);
   printf("fin de %d !n", p);
}

int main()
{
   signal(SIGCHLG, fin_fils);

   if (fork() == 0) {
       ...
   }
}
```

3.6 Mise en oeuvre sur l'exemple

Par exemple, la réception d'un signal

- fera afficher la ligne de commande concernée
- retirera le processus de la liste de travaux en arrière plan

4 Les Signaux

- mécanisme de communication primitif, ne transmet qu'un numéro de signal
- permet à un processus de réagir à un évènement.
  - `SIGQUIT`,
  - `SIGCHLD`,
  - `SIGSTOP`,
  - `SIGCONT`,
  - `SIGSEGV`,
  - `SIGUSR1`, ...

4.1 Déclenchement/réception d'un signal

- Déclenchement par
  - évènement matériel (violation d'accès mémoire, ...)
  - évènement du système (un processus s'est terminé, ...)
  - appel à la fonction `kill(pid,sig)`
- Réception par un processus
  - comportement par défaut : dépend du signal (ignorer, mettre fin au processus, le stopper, le relancer)
  - modification par : `signal(sig, handler)`

4.2 Exemple : SIGINT

- On tape "controle-C" pendant l'exécution d'un programme :
  - le shell l'intercepte
  - il envoie le signal SIGINT au processus qui tourne
- Lorsque le "handler" est lancé,
  - le comportement par défaut est rétabli
  - réarmer le signal ?

4.3 Exercice

- Faire un programme qui
  - incrémente et affiche un compteur quand il reçoit SIGUSR1
  - s'arrête de lui-même au bout de 30 secondes
- À découvrir :
  - la commande "`kill`"
  - fonction `alarm()`
  - attendre à ne rien faire : `while(true} sleep(1);`
  - ou mieux `pause();`

5 Résumé fork/exec/wait/signal

Processus

- lancer un autre processus : `fork()`
- faire exécuter un programme : `exec()`
- attendre leur fin : `waitpid()`, `wait()`

Signaux

- notifier des évènements : `kill()`, `raise()`
- les traiter : `signal()`

6 Ce qu'on va voir maintenant

Rediriger l'entrée/la sortie d'un programme :

```
ls -l > sortie.txt
```

Echanger des données entre programmes à l'aide d'un tuyau :

```
du -s . | sort -n
```

7 Redirections

Si on fait exécuter

```
int main()
{
    execl("/bin/ls", "ls", "-l", NULL);
}
```

la commande "ls" hérite de la table de descripteurs ouverts

numéro	nom	fichier ouvert
0	STDIN_FILENO	clavier
1	STDOUT_FILENO	fenêtre
2	STDERR_FILENO	fenêtre

7.1 Idée : rediriger la sortie vers `sortie.txt`

Ce qu'on veut :

numéro	nom	fichier ouvert
0	STDIN_FILENO	clavier

numéro	nom	fichier ouvert
1	**STDOUT_FILENO**	**fichier `sortie.txt` en écriture <=**
2	STDERR_FILENO	fenêtre

Étapes :

1. obtenir un descripteur `fd` en ouvrant le fichier
2. le copier dans celui de `STDOUT_FILENO`
3. fermer `fd`

7.2 Étape 1 : ouvrir le fichier de sortie (`open`)

```
int fd = open("sortie.txt",           // pathname
              O_CREAT | O_WRONLY,     // flags
              0644);                  // mode = permissions
```

Après :

numéro	nom	fichier ouvert
0	STDIN_FILENO	clavier
1	STDOUT_FILENO	fenêtre
2	STDERR_FILENO	fenêtre
=> 3		**fichier `sortie.txt` <=**

7.3 Étape 2 : dupliquer le descripteur (`dup2`)

```
dup2(fd, STDOUT_FILENO);  // duplicate fd to STDOUT_FILENO
```

Après :

numéro	nom	fichier ouvert
0	STDIN_FILENO	clavier
=> 1	STDOUT_FILENO	**fichier `sortie.txt` <=**
2	STDERR_FILENO	fenêtre
3		fichier `sortie.txt`

7.4 Étape 3 : fermer le descripteur inutile (`close`)

```
close(fd);        // close file descriptor
```

Après :

numéro	nom	fichier ouvert
0	STDIN_FILENO	clavier
1	STDOUT_FILENO	**fichier `sortie.txt`**
2	STDERR_FILENO	fenêtre
—	—	—

7.5 En résumé

```
int main()
{
   int fd = open("sortie.txt",
```

```
                O_CREAT | O_WRONLY,
                0644) ;
    dup2(fd, STDOUT_FILENO) ;
    close(fd) ;

    execl("/bin/ls",
          "ls", "-l", NULL) ;
}
```

7.6 Paramètre d'appel pour `open()`

```
int fd = open(chemin_d_accès, indicateurs, mode) ;
```

Indicateurs : combinaison avec

- **obligatoirement** un des modes d'accès `O_RDONLY`, `O_WRONLY`, `O_RDWR`
- `O_CREAT`, créer le fichier si il n'existe pas
- `O_APPEND`, écrire à la fin du fichier existant
- `O_TRUNC`, vider avant d'y écrire
- ...

7.7 Mode = permissions d'accès

- droits d'accès pour **utilisateur**, **groupe**, **autres**
- on les écrit souvent en **octal** (nombre commençant par **0**)
- constantes symboliques

```
S_IRUSR  00400 user has read permission
S_IWUSR  00200 user has write permission
S_IXUSR  00100 user has execute permission
...
```

(voir `man 2 open`)

7.8 Exercice

Écrire un programme C : `min2maj f1 f2`

- prend un fichier texte en entrée,
- fabrique un fichier de sortie,
- les minuscules ont été converties en majuscules.

Indication : en langage de commande, on utiliserait `/usr/bin/tr`

```
tr [a-z] [A-Z] < entree  > sortie
```

Amélioration

- sortie standard si `f2` absent ou égal à "-"
- entrée standard si `f1` absent ou égal à "-"

7.9 Remarque : indicateur `O_CLOEXEC` (*close on exec*)

- un descripteur marqué `O_CLOEXEC` n'est pas transmis par `exec()`
- l'indicateur `O_CLOEXEC` n'est pas copié par `dup2()`

```
int main()
{
    int fd = open("sortie.txt",
                O_CREAT | O_WRONLY | O_CLOEXEC,
                0644) ;
```

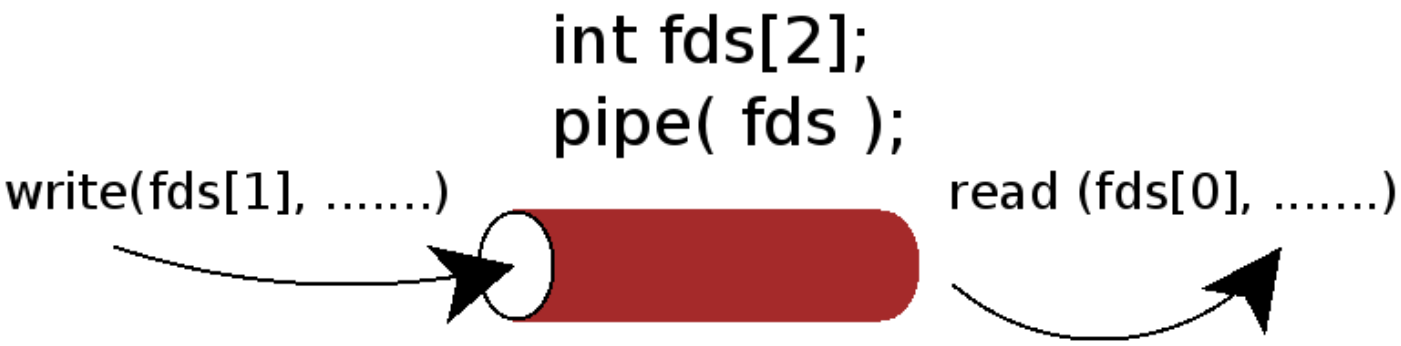

FIG. 1 : tuyau

```
    dup2(fd, STDOUT_FILENO);          // close(fd);
    execl("/bin/ls",
          "ls", "-l", NULL);
}
```

7.10 Travail : préparer l'intégration dans le shell

Ecrire une fonction

```
void do_command(struct Shell *this, const struct StringVector *args);
```

qui lance un programme en tenant compte des redirections

Tests :

```
args.strings = { "ls" , "-l" };
args.strings = { "ls" , "-l", ">", "r.txt"};
args.strings = { "<", "f.cc", "wc", "-l", ">", "r.txt"};
```

8 Tuyaux

8.1 Tuyau = mécanisme de communication

L'appel système `pipe()` retourne deux descripteurs de fichiers, connectés à un tampon de données.

- on peut écrire par un descripteur
- on peut lire par l'autre

8.2 Précisions

- La **capacité** du tampon est **limitée**
  - POSIX : au moins 512 octets
  - Linux : de 4 à 64 Ko
- La lecture et l'écriture sont
  - **atomiques** : deux écritures simultanées ne peuvent pas se mélanger
  - **bloquantes** : écriture dans tampon trop plein, lecture tampon vide

8.3 Exemple

- Un processus produit des messages (nombres)
- Le processus père les lit, et les affiche

```
function main                  fonction produire
  creer pipe                     pour i de 1 à 10
  lancer produire                  |  envoyer i
```

```
  consommer                      |  attendre 1 sec

            fonction consommer
              tant que c'est possible
              |  lire un nombre
              |  l'afficher
```

8.4 Code du producteur

```
struct Message {   // structure de taille fixe
     int data;
};

void produire(int sortie)
{
  for (int i = 1; i<=10; i++) {
    struct Message m;
    m.data = i;
    write(sortie, &m, sizeof(m));
  }
  close(sortie);
}
```

8.5 Code du consommateur

```
void consommer(int entree)
{
  while (true) {
    struct Message m;
    int lus = read(entree, &m, sizeof(m));
    if (lus != sizeof(m)) {
         break;
    }
    printf("%dn", m.data);
  }
  close(entree);
}
```

8.6 Le programme (main)

```
    int fds[2];
    pipe(fds);
    int entree = fds[0], sortie = fds[1];

    if (fork() == 0) {  // processus fils
        close(entree);
        produire(sortie);
        exit(EXIT_SUCCESS);
    }
    close(sortie);      // processus père
    consommer(entree);
    wait(nullptr);
    exit(EXIT_SUCCESS);
```

8.7 Question

Que se passe-t'il si le processus père ne fait pas `close(sortie)` ?

9 Résumé tuyaux

- appel `pipe(fds)` pour créer un tuyau
- retourne deux descripteurs dans tableau `int fds[2] ;`
- lecture dans `fds[0]`, écriture dans `fds[1]`

La lecture détecte une fin de fichier si

- il n'y a plus de données à lire,
- toutes les copies du descripteur `fds[1]` sont fermées

9.1 Résumé tuyau (écriture)

`write(fd, adresse, nombre d'octets)`

- **Paramètres :**
  - descripteur
  - adresse du premier octet à transmettre
  - nombre d'octets à transmettre

9.2 Résumé tuyau (lecture)

`n = read(fd, adresse, nombre d'octets)`

- **Paramètres**
  - descripteur
  - adresse du tampon de réception
  - taille maximum
- **retourne** le nombre d'octets *effectivement* lus
  - 0 en fin de fichier
  - -1 si erreur

10 Exercice : pipeline

10.1 Sujet

- Ecrire un "pipe-line" enchainant plusieurs actions simples
  - produire les entiers de 1 à 10
  - sélectionner ceux qui ne sont pas multiples de 3
  - les multiplier par 100
  - faire afficher les résultats
- Chaque action aura en paramètre un descripteur d'entrée ou un descripteur de sortie, ou les deux.
- Le résultat devrait être 100, 200, 400, 500, 700, 800, 1000.

10.2 Répartition du travail

Objectif A la fin, chacun est capable de présenter le programme au nom du groupe (de 3)

1. chacun réalise le programme réduit "produire les entiers de 1 à 10, les afficher"
2. coordination : aidez-vous pour que **tout le monde** comprenne
3. chacun ajoute un "filtre" : sélection, ou transformation
4. coordination : mettre en commun les filtres
5. intégrer l'autre filtre
6. coordination : tout le monde a un programme qui marche