INF2610

Chapitre 4: Communication interprocessus

4.1 Deux processus sont connectés par un canal bi-directionnel (un tube anonyme dans chaque direction). Le processus A demande au processus B la traduction d'un texte. Différentes stratégies sont possibles pour A. Le processus B lit 128 caractères à la fois et une fois un paragraphe complet reçu, il le traduit et le renvoie à A avant de poursuivre sa lecture de A. Pour chacune des stratégies pour A, dites si elle peut fonctionner (jamais, parfois, toujours...) et pourquoi.

1. Le processus A envoie tout le contenu du texte vers B et ensuite lit de B le texte traduit.
2. Le processus A envoie 128 caractères du texte vers B et ensuite lit la réponse de B.
3. Le processus A utilise un fil d'exécution pour envoyer le texte à B et un autre pour lire la réponse de B.

Dans le premier cas, à mesure que A envoie les données, elles sont lues par B. Toutefois, lorsque B aura retourné 4K de texte traduit sans que A ne le lise, B restera bloqué sur cette écriture. Ensuite, A pourrait remplir aussi jusqu'à 4K dans le tube vers B avant de bloquer. Ainsi, sauf pour de très petits textes (e.g. moins de 8K), il y aura un interblocage. Dans le second cas, si les 128 caractères ne constituent pas un paragraphe, B sera toujours en attente alors que A se mettra aussi en attente, encore en interblocage. Le troisième cas fonctionne très bien.

4.2 Les sorties stderr et stdout sont toutes deux redirigées par défaut vers le terminal. Quelle est l'utilité d'avoir deux sorties ainsi séparées?

Cette séparation permet par exemple de rediriger stdout vers un fichier ou dans un tube vers un autre processus, tout en obtenant les messages d'erreur sur le terminal. En outre, les paramètres par défaut de stdout et stderr sont différents; souvent stderr n'utilise que peu ou pas de tampon, perdant en performance mais s'assurant que l'usager est averti de l'erreur le plus tôt possible.

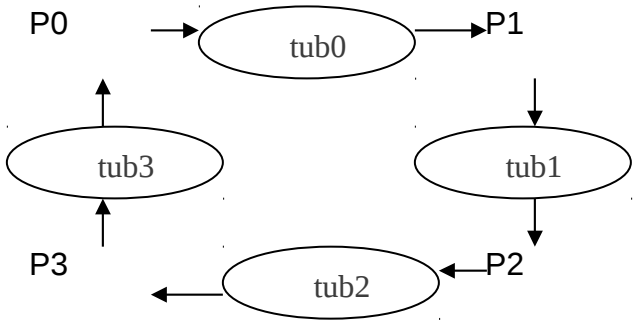
4.3 Quel est l'intérêt d'avoir une table de descripteurs de fichiers pour chaque processus? Simplement avoir un pointeur qui réfère au inode, périphérique et à la position ne suffirait-il pas?

La table de descripteurs permet un niveau d'indirection qui donne un certain degré de flexibilité, par exemple pour connecter ensemble des processus à travers stdin et stdout. De plus, l'ouverture et la fermeture explicites permettent de savoir quels sont les fichiers en usage et d'empêcher de les détruire.

4.4 Est-il possible d'utiliser l'appel système mmap et des signaux comme mécanisme de communication inter-processus? L'appel mmap seulement?

Lorsque deux processus utilisent mmap pour un même fichier, cela leur fournit un espace de mémoire partagée. Il est donc facile de communiquer à condition d'avoir un mécanisme de synchronisation. Les signaux peuvent facilement fournir un tel mécanisme (e.g. pour qu'un processus signifie à l'autre qu'il a terminé de remplir un tampon en mémoire). Sans les signaux, il faut utiliser un autre mécanisme de synchronisation, par exemple un mutex offert par le système d'exploitation ou la librairie standard, ou même un mutex programmé soi-même à l'aide d'opérations atomiques.

4.5 Considérez N processus qui communiquent au moyen de tubes de communication non nommés (unnamed pipe). Chaque processus partage deux tubes (un avec le processus de droite et un autre avec le processus de gauche). Par exemple pour N=4, les processus communiquent selon le schéma suivant :



1. Complétez le code ci-après de manière à implémenter cette architecture de communication des N processus créés. L'entrée standard et la sortie standard de chaque processus Pi sont redirigées vers les tubes appropriés. Par exemple, pour le processus P0, l'entrée et la sortie standards deviennent respectivement les tubes tub3 et tub0.

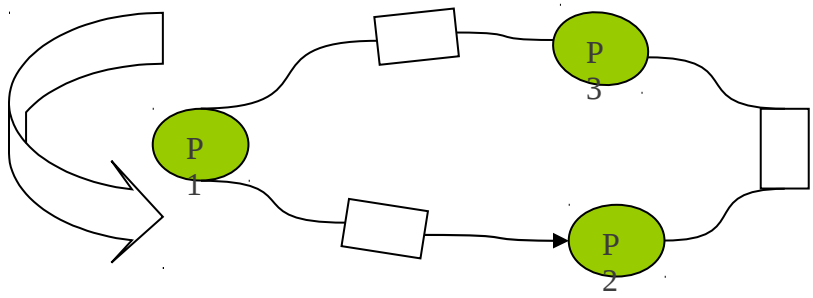
```
#include ″EntetesNecessaires.h″
#define N 4
void proc( int ) ;
int main ( )
{       int i ;
```

```
        for( i=0 ; i <N; i++)
                if ( fork() ==0)
              {                 proc(i);   exit(0);          }
        exit(0) ;
}
#include  ″codedeproc″
```

Attention : Vous ne devez pas écrire le code de la fonction proc.

```
#include ″EntetesNecessaires.h″
#define N 4
void proc( int ) ;
int main ( )
{         int i ;
        int fd[N][2];
        for( i=0 ; i <N; i++)  pipe(fd[i]);
        for( i=0 ; i <N; i++)
                if ( fork() ==0)
              {
                        dup2(fd[i][1],1);
                        dup2(fd[(N+i-1)%N][0],0);
                        for(int j=0 ; j <N; j++) {close (fd[j][0]); close(fd[j][1]);}
                        proc(i);   exit(0);
              }
        exit(0) ;
}
#include  ″codedeproc″
```

4.6 On veut établir, en utilisant les tubes anonymes (pipes), une communication de type anneau unidirectionnel entre trois processus fils. Pour ce faire, la sortie standard de l'un doit être redirigée vers l'entrée standard d'un autre, selon le schéma suivant :



Complétez le programme suivant en y ajoutant le code permettant de réaliser les redirections nécessaires à la création d'un tel anneau.

```
/*0*/
int main ()
{
        /*1*/
        if (fork()) // création du premier processus
        {
                if(fork())
                {
                        /*2*/
                        if(fork())
                        { /*3*/
                                while (wait(NULL)>0);
                                /*4*/
                        } else
                        { // processus P3
                                /*5*/
                                execlp("program3", "program3",NULL);
                                /*6*/
                        }
                } else
                { // processus P2
                        /*7*/
```

```
                execlp("program2", "program2",NULL);
                /*8*/
        }
    } else
    { //processus P1
        /*9*/
        execlp("program1","program1", NULL);
        /*10*/
    }
    /*11*/
}

/*0*/
int main ()
{
    /*1*/
    int fd12[2], fd23[2], fd31[2];
    pipe(fd12); pipe(fd23); pipe(fd31);
    if (fork()) // création du premier processus
    {
        if(fork())
        {
            /*2*/
            if(fork())
            { /*3*/
                while (wait(NULL)>0);
                /*4*/
            } else
            { // processus P3
                /*5*/
                dup2(fd23[0],0); dup2(fd31[1],1);
                close(fd12[1]); close(fd12[0]);
                close(fd23[1]); close(fd23[0]);
                close(fd31[1]); close(fd31[0]);
                execlp("program3", "program3",NULL);
                /*6*/
            }
        } else
        { // processus P2
            /*7*/
            dup2(fd12[0],0); dup2(fd23[1],1);
            close(fd12[1]); close(fd12[0]);
            close(fd23[1]); close(fd23[0]);
            close(fd31[1]); close(fd31[0]);
```

```
                execlp("program2", "program2",NULL);
                /*8*/
        }
    } else
    { //processus P1
        /*9*/
        dup2(fd12[1],1); dup2(fd31[0],0);
        close(fd12[1]); close(fd12[0]);
        close(fd23[1]); close(fd23[0]);
        close(fd31[1]); close(fd31[0]);
        execlp("program1","program1", NULL);
        /*10*/
    }
    /*11*/
}
```

4.7 Écrivez le « main » d'un processus qui permet de simuler un pipe ('|'). Ce code doit utiliser un tube nommé et doit avoir exactement le même comportement qu'un pipe ('|') dans un shell.

Exemples :

bash$> my_pipe who wc –l (equivaut à bash$> who | wc –l)

bash$> my_pipe ls grep "my_pipe" (équivaut à bash$> ls | grep "my_pipe")

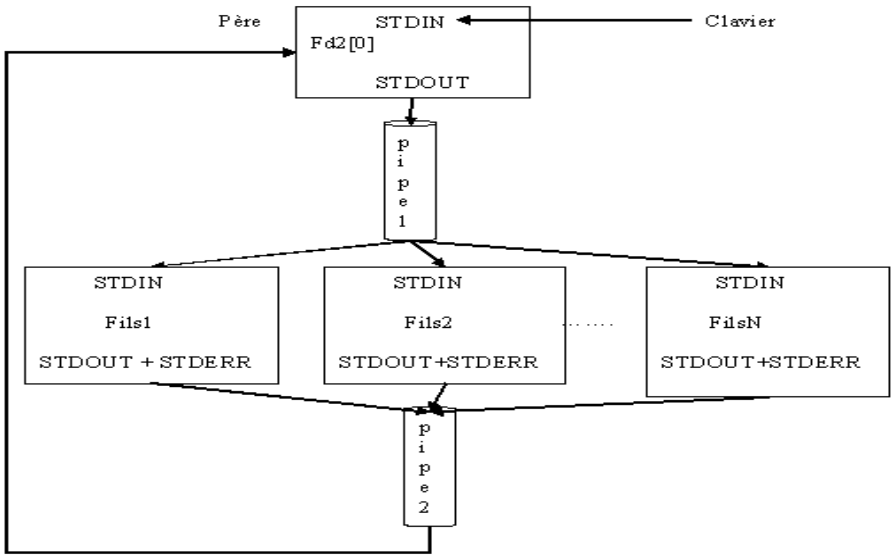
```
/*  Code testé sur Linux */
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/stat.h>
#include <sys/errno.h>
extern  int errno;

#define FIFOA "fifoa"

int main(int argc, char* argv[])
{
    int writefd, readfd;
```

```
        if (mknod(FIFOA, S_IFIFO | 0666, 0) <  0) /* create fifo */
                perror("FIFOA open failed");

        if (!fork())
        {
                if ((writefd = open(FIFOA, O_WRONLY)) < 0)
                {
                        perror("Open FIFOA write");
                        exit(1);
                }

                dup2(writefd, 1);
                close(writefd);
                argv[2] = NULL;
                execvp(argv[1], &argv[1]);
                perror("Error EXECVP");
                exit(0);
        }

        if (!fork())
        {
                if ((readfd = open(FIFOA, O_RDONLY)) < 0)
                {
                        perror("Open FIFOA read");
                        exit(1);
                }
                dup2(readfd, 0);
                close(readfd);
                execvp(argv[2], &argv[2]);
                perror("Error EXECVP");
                exit(0);
        }

 wait(NULL);
 wait (NULL);
 unlink(FIFOA);
 exit(0);
}
```

4.8 Considérez le programme suivant :

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define N 5
void gestion(){ /* traitement */ };

/*0*/
int main( )
{       pid_t pid[N];
        int i;
        /*1*/
        for ( i=0; i<N;i++)
            if ((pid[i]=fork())==0)
            {   /* 2*/
                execlp("traitement", "traitement", NULL);
                exit(1);
            }
          /*3*/
        gestion( );
          return 0;
}
```

1. On veut faire communiquer le processus père avec ses fils au moyen de deux tubes anonymes (unnamed pipe) selon le schéma suivant :



La sortie standard du père est redirigée vers le pipe1 qui devient l'entrée standard des fils. Les sorties standards et erreurs des fils sont redirigées vers le pipe2.

Complétez le code de manière à établir ces canaux de communication (supposez que le fork n'échoue jamais).

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define N 5
void gestion(){ /* traitement */ };

/*0*/  int fd1[2], fd2[2];
int main( )
{         pid_t pid[N];
           int i;
          /*1*/  pipe(fd1); pipe(fd2);
          for ( i=0; i<N;i++)
               if ((pid[i]=fork())==0)
               {    /* 2*/   dup2(fd1[0],0) ; dup2(fd2[1],1) ; dup2(fd2[1],2) ;
                             close(fd1[0]); close(fd1[1]); close(fd2[1]); close(fd2[0]);
                    execlp("traitement", "traitement", NULL);
                    exit(1);
               }
             /*3*/ dup2(fd1[1],1); close(fd1[0]); close(fd1[1]); close(fd2[1]);
          gestion(  );
            return 0;
}
```

4.9 On dispose d'une fonction F() d'une bibliothèque qui écrit une certaine quantité de données sur la sortie standard (descripteur de fichier 1). On aimerait récupérer, en utilisant un tube anonyme, ces données pour les traiter.

```
#define Taille  1024
int main
{
         char Data[Taille];
```

```
        /*1*/
        F ( );
        /*2*/
        utiliser_resultat (Data);
}
```

1. Insérez du code en amont et en aval de F( ) afin que tous les caractères émis par F( ) sur la sortie standard soient récupérés dans Data. Vous pouvez utiliser des variables et appels système supplémentaires, mais vous ne pouvez pas utiliser de fichiers ni de processus supplémentaires.
2. Que se passe-t-il, si la taille des données émises par F dépasse celle de Data ?
3. Que se passe-t-il, si la taille des données émises par f dépasse celle du tube ?
4. Proposez une deuxième version corrigeant ces problèmes. Pour ce faire, vous pouvez utiliser un processus supplémentaire.

1.
```
int main()
{       char Data [Taille];
        int fd[2];
        pipe (fd);
        dup2 (fd[1], 1);
        close (fd[1]);
        F ();
        close(1) ;
        read (fd[0], Data, Taille);
        close (fd[0]);

        utiliser_résultat (Data);
}
```

2. Si la taille des données émises par F dépasse celle de Data, il y aura perte de données car il y a une seule lecture du pipe et le nombre de caractères lus ne peut pas dépasser Taille.

3. Si la taille des données redirigées est trop importante, le processus se bloquera lorsque F() tentera d'écrire dans un tube plein, et sans jamais pouvoir être réveillé puisque les lectures sur le tube ne pourront avoir lieu qu'à la sortie de F().

4. int main ()
```
{       char Data[Taille];
        int fd[2];

        pipe (fd);
        if (fork () == 0)
        {       /* le fils */
                close (fd[0]);
                dup2 (fd[1], 1);
                close (fd[1]);
                F ();
                close (1);
                exit (0);
        }
        else {
                close (fd[1]);
                while (read (fd[0], Data, Taille) > 0)
                        utiliser_resultat (Data);
                close (fd[0]);
                wait (NULL);
                }
}
```