TD : communication par tubes anonymes

Exercice 1.

filtre.c

```
 1 : #include <stdlib.h>
 2 : #include <stdio.h>
 3 : #include <unistd.h>
 4 : #include <sys/wait.h>
 5 :
 6 : int p[2];
 7 :
 8 : void fils1(void)
 9 : {
10 :     char c;
11 :
12 :     close(p[0]);
13 :     printf("début fils1 (taper 0 pour fin)\n");
14 :     while ((c = getchar()) != '0')
15 :         if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'))
16 :         {
17 :             if ((c >= 'a') && (c <= 'z'))
18 :                 c -= 32;
19 :             write(p[1], &c, 1);
20 :             printf("Le fils1 envoie >%c<\n", c);
21 :         }
22 :     close(p[1]);
23 :     exit(EXIT_SUCCESS);
24 : }
25 :
26 : void fils2(void)
27 : {
28 :     char c;
29 :
30 :     close(p[1]);
31 :     printf("debut fils2\n");
32 :     while (read(p[0], &c, 1) > 0)
33 :         printf("fils2 reçoit >%c<\n", c);
34 :     close(p[0]);
35 :     exit(EXIT_SUCCESS);
36 : }
37 :
38 : int main(void)
39 : {
40 :     if (pipe(p) != 0)
41 :     {
42 :         printf("pb ouverture pipe \n");
43 :         exit(EXIT_FAILURE);
44 :     }
45 :     if (fork() == 0)
46 :         fils1();
47 :     if (fork() == 0)
48 :         fils2();
49 :     close(p[0]);
50 :     close(p[1]);
51 :     wait(NULL);
52 :     wait(NULL);
53 :     printf("fin du père\n");
54 :     exit(EXIT_SUCCESS);
55 : }
```

Que se passe-t'il pendant l'exécution de `filtre.c` ?

Exercice 2.

tubonacci.c

```
 1 : #include <unistd.h>
 2 : #include <stdlib.h>
 3 : #include <stdio.h>
 4 :
 5 : void parent(int entree, int sortie)
 6 : {
 7 :     unsigned char n;
 8 :     n = 0;
 9 :     write(sortie, &n, 1);
10 :     n = 1;
11 :     write(sortie, &n, 1);
12 :     do
13 :     {
14 :         read(entree, &n, 1);
15 :         printf("%u ", n);
16 :         write(sortie, &n, 1);
17 :     }
18 :     while (n <= 200);
19 :     close(sortie);
20 :     wait(NULL);
21 :     exit(EXIT_SUCCESS);
22 : }
23 :
24 : void enfant(int entree, int sortie)
25 : {
26 :     unsigned char p, q;
27 :     read(entree, &p, 1);
28 :     for (;;)
29 :     {
30 :         write(sortie, &p, 1);
31 :         if (read(entree, &q, 1) != 1)
32 :             exit(EXIT_SUCCESS);
33 :         p += q;
34 :     }
35 : }
36 :
37 : int main(void)
38 : {
39 :     int tube_pe[2];
40 :     int tube_ep[2];
41 :     int pid;
42 :
43 :     printf("[%d] début du père.\n", getpid());
44 :
45 :     pipe(tube_pe);              /* tube Parent -> Enfant */
46 :     pipe(tube_ep);              /* tube Enfant -> Parent */
47 :
48 :     if ((pid = fork()) == -1)
49 :         exit(EXIT_FAILURE);
50 :     if (pid > 0)                /* parent */
51 :     {
52 :         close(tube_pe[0]);
53 :         close(tube_ep[1]);
54 :         parent(tube_ep[0], tube_pe[1]);
55 :     } else                      /* enfant */
56 :     {
57 :         close(tube_pe[1]);
58 :         close(tube_ep[0]);
59 :         enfant(tube_pe[0], tube_ep[1]);
60 :     }
61 :     printf("[%d] : terminaison\n", getpid());
62 :     exit(EXIT_SUCCESS);
63 : }
```

Que ce passe-t'il pendant l'exécution de `tubonacci.c` ?

Exercice 3. Modifier `tubonacci.c` pour calculer les ternes de la suite définie par $u_0 = 2$, $u_1 = 3$, $u_{n+2} = 2u_{n+1} + 3u_n$.

Exercice 4. Que ce passe-t'il si l'on supprime la ligne 19 de `tubonacci.c` ?

Exercice 5 : chasse aux erreurs. Deviner le comportement du programme qui suit et en corriger les erreurs d'implémentation.

chasseerreur.c

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <unistd.h>
 4 : #include <signal.h>
 5 :
 6 : static int n = 0;
 7 :
 8 : void erreur(const char *errmess)
 9 : {
10 :     perror(errmess);
11 :     exit(EXIT_FAILURE);
12 : }
13 :
14 : void interruption(int signum)
15 : {
16 :     printf("No choisi : %d\n", n);
17 :     exit(EXIT_SUCCESS);
18 : }
19 :
20 : void init_iteration(int d_ecriture)
21 : {
22 :     write(d_ecriture, &n, sizeof(int));
23 : }
24 :
25 : void iteration(int d_lecture, int d_ecriture)
26 : {
27 :     while (1)
28 :     {
29 :         read(d_lecture, &n, sizeof(int));
30 :         n++;
31 :         write(d_ecriture, &n, sizeof(int));
32 :     }
33 : }
34 :
35 : int main(void)
36 : {
37 :     int pf[2], fp[2];           /* Communication bidirectionnelle */
38 :     struct sigaction action;
39 :     action.sa_flags = 0;
40 :     sigemptyset(&action.sa_mask);
41 :
42 :     if (pipe(pf) == -1 || pipe(fp) == -1)
43 :         erreur("pipe");
44 :
45 :     switch (fork())
46 :     {
47 :     case -1:
48 :         erreur("fork");
49 :     case 0:                     /* Fils */
50 :         close(fp[0]);
51 :         close(pf[1]);
52 :         iteration(pf[0], fp[1]);
53 :     default:                    /* Père */
54 :         action.sa_handler = interruption;
55 :         sigaction(SIGINT, &action, NULL);
56 :         close(pf[0]);
57 :         close(fp[1]);
58 :         iteration(fp[0], pf[1]);
59 :     }
60 :     exit(EXIT_SUCCESS);
61 : }
```

Exercice 6 : tri distribué. Écrire un programme où un processus père crée deux fils ; chaque fils attend un couple d'entiers (x, y) du père et le renvoie dans l'ordre croissant.
Le père demande quatre entiers a, b, c, d à l'utilisateur et ordonne les couples (a', b') et (c', d'). Puis il ordonne les couples (a', c') et (b', d'), il obtient les couples (a'', c'') et (b'', d''). Finalement, il ordonne le couple (c'', b'') il obtient le

couple (c''', b''') et affiche (a'', c''', b''', d'').