Les primitives `dup` et `exec`

Exercice 1. Que se passe-t'il pendant l'exécution du programme suivant ?

progr1.c

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <unistd.h>
 4 : #include <sys/wait.h>
 5 : /* #include <sys/types.h> */
 6 :
 7 : int main(void)
 8 : {
 9 :
10 :     int calu, tube[2], fd;
11 :
12 :     pipe(tube);
13 :
14 :     if (fork())                /* père */
15 :     {
16 :         close(STDOUT_FILENO);
17 :         fd = dup(tube[1]);     /* Il écrit dans le tube */
18 :         close(tube[0]);
19 :         close(tube[1]);
20 :         printf("%d\n", fd);
21 :
22 :         while ((calu = getchar()) != '.')
23 :         {
24 :             if (calu >= 'a' && calu <= 'z')
25 :                 calu -= 'a' - 'A';
26 :             printf("%c", calu);
27 :         }
28 :         fflush(stdout);
29 :         close(STDOUT_FILENO);
30 :         wait(NULL);
31 :         exit(EXIT_SUCCESS);
32 :     } else                     /* fils */
33 :     {
34 :         close(STDIN_FILENO);
35 :         dup(tube[0]);          /* Il lit du tube */
36 :         close(tube[0]);
37 :         close(tube[1]);
38 :         while ((calu = getchar()) != EOF)
39 :             printf("[fils] : >%c<\n", calu);
40 :         printf("Fin du fils\n");
41 :         exit(EXIT_SUCCESS);
42 :     }
43 : }
```

Exercice 2. Décrire précisément ce que fait le programme suivant.

progr2.c

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <unistd.h>
 4 : #include <assert.h>
 5 :
 6 : int main(void)
 7 : {
 8 :     int tube[2];
 9 :
10 :     pipe(tube);
11 :
12 :     if (fork())                /* père */
13 :     {
14 :         close(STDIN_FILENO);
15 :         dup(tube[0]);
16 :         close(tube[0]);
17 :         close(tube[1]);
18 :         execlp("wc", "Wordcount", "-l", NULL);
19 :         /* Jaimais atteint sauf erreur */
20 :         perror("Erreur dans exec wc");
21 :     } else                     /* fils */
22 :     {
23 :         close(STDOUT_FILENO);
24 :         dup(tube[1]);
25 :         close(tube[0]);
26 :         close(tube[1]);
27 :         execlp("ls", "ls", "-l", NULL);
28 :         /* Jaimais atteint sauf erreur */
29 :         perror("Erreur dans exec ls");
30 :     }
31 :     /* Jaimais atteint sauf erreur */
32 :     exit(EXIT_FAILURE);
33 : }
```

Exercice 3. Il y a des erreurs dans le traitement des erreurs du programme précédant. Quelle sont-elles ?

Exercice 4. Écrire un programme qui prend comme paramètre une commande shell et l'exécute en remplaçant les caractères minuscules par des majuscules sur la sortie standard.

Exercice 5. Écrire un programme qui exécute la commande shell `ls -l | grep   \.c$ | wc -l`.

Exercice 6. Le programme suivant s'exécute comme prévu ? Pourquoi ?

progr6.c

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <unistd.h>
 4 :
 5 : int main(void)
 6 : {
 7 :     printf("Bonjour ");
 8 :     execl("/bin/echo", "echo", "à tous !", NULL);
 9 :     exit(EXIT_FAILURE);
10 : }
```

Exercice 7. Écrire la fonctions `execvp` en utilisant la fonction `execv`.

Exercice 8. Un peu d'anglais : voici la description POSIX de la fonction `system` (appartenant à la bibliothèque standard C) :

```
#include <stdlib.h>

int system(const char *command);

DESCRIPTION

If command is a null pointer, the system() function shall determine
whether the host environment has a command processor. If command is
not a null pointer, the system() function shall pass the string
pointed to by command to that command processor to be executed in
an implementation-defined manner; this might then cause the program
calling system() to behave in a non-conforming manner or to terminate.

The environment of the executed command shall be as if a child process
were created using fork(), and the child process invoked the sh
utility using execl() as follows:

execl(<shell path>, "sh", "-c", command, (char *)0);

where <shell path> is an unspecified pathname for the sh utility.

The system() function shall ignore the SIGINT and SIGQUIT signals,
and shall block the SIGCHLD signal, while waiting for the command to
terminate. If this might cause the application to miss a signal that
would have killed it, then the application should examine the return
value from system() and take whatever action is appropriate to the
application if the command terminated due to receipt of a signal.

The system() function shall not affect the termination status of any
child of the calling processes other than the process or processes it itself creates.

The system() function shall not return until the child process has terminated.

RETURN VALUE

If command is a null pointer, system() shall return non-zero to
indicate that a command processor is available, or zero if none is
available.

If command is not a null pointer, system() shall return the
termination status of the command language interpreter in
the format specified by waitpid(). The termination status shall
be as defined for the sh utility; otherwise, the termination status
is unspecified. If some error prevents the command language
interpreter from executing after the child process is created,
the return value from system() shall be as if the command language
interpreter had terminated using exit(127) or _exit(127). If a child
process cannot be created, or if the termination status for the
command language interpreter cannot be obtained, system() shall
return -1 and set errno to indicate the error.
```

1. Écrire une implementation de la fonction `system`.
2. Écrire un analogue de la fonction system, qui ne se sert pas du shell.