

# Projet `minishell` - Rapport

Vianney HERVY - 1ASN Groupe H

May 24, 2024

## Contents

<b>1</b>	<b>Architecture</b>	<b>2</b>
<b>2</b>	<b>Choix et spécificités de conception</b>	<b>2</b>
<b>3</b>	<b>Méthodologie des tests</b>	<b>3</b>
<b>4</b>	<b>Étapes</b>	<b>4</b>
	Étape 1 (Testez le programme) . . . . .	4
	Étape 2 (Lancement d'une commande) . . . . .	4
	Étape 3 (Enchaînement séquentiel des commandes) . . . . .	5
	Étape 4 (Lancement de commandes en tâche de fond) . . . . .	6
	Étape 5 (Rendu) . . . . .	7
	Étape 6 (Traitement du signal <code>SIGCHLD</code> ) . . . . .	7
	Étape 7 (Utilisation de <code>SINGINT</code> pour traiter la terminaison des processus fils) . . . . .	7
	Étape 8 (Attendre un signal : <code>pause</code> ) . . . . .	7
	Étape 9 (Suspension et reprise d'un processus en arrière plan) . . . . .	8
	Étape 10 (Affichage d'un message indiquant le signal reçu) . . . . .	8
	Étape 11 (Rendu) . . . . .	9
	Étape 12 (Test de la frappe au clavier de <code>ctrl-c</code> et <code>ctrl-z</code> ) . . . . .	9
	Étape 13 (Gestion de la frappe au clavier de <code>ctrl-c</code> et <code>ctrl-z</code> ) . . . . .	9
	Étape 13.1 (Changer le traitement des signaux <code>SIGINT</code> et <code>SIGTSTP</code> ) . . . . .	9
	Étape 13.2 (Ignorer les signaux <code>SIGINT</code> et <code>SIGTSTP</code> ) . . . . .	10
	Étape 13.3 (Masquer les signaux <code>SIGINT</code> et <code>SIGTSTP</code> ) . . . . .	10
	Étape 14 (Détacher les processus fils en arrière plan) . . . . .	11
	Étape 15 (Rendu) . . . . .	12
	Étape 16 (Redirections) . . . . .	12
	Étape 17 (Rendus 2) . . . . .	14
	Étape 19 (Tubes simples) . . . . .	14
	Étape 20 (Pipelines) . . . . .	14
	Étape 21 (Rendu) . . . . .	16

<b>5</b>	<b>Codes complémentaires</b>	<b>16</b>
5.1	copier.c . . . . .	17
5.2	tube1.c . . . . .	17
5.3	tube2.c . . . . .	17
5.4	tube3.c . . . . .	17
5.5	tube4.c . . . . .	17
5.6	tube5.c . . . . .	17

# 1 Architecture

L'architecture de ce projet est simple étant donné qu'elle nous est presque imposée par le sujet. La majorité du code utile se trouve dans le fichier `minishell.c`. Le package formé par `readcmd.c` et `readcmd.h` est utilisé pour extraire de la commande utilisateur les informations telles que la mise en arrière plan, les redirections, les tubes etc.

## 2 Choix et spécificités de conception

J'ai rapidement vu que la gestion d'erreur en C n'était pas une mince affaire. Pour éviter les répétitions de code, j'ai créé 5 fonctions "sûres" qui gèrent les erreurs de manière uniforme. Ces fonctions sont `safeopen`, `safeclose`, `safedup2`, `safeexecvp` et `safepipe`.

```
1  /* Ouvrir le fichier 'nom' dans un descripteur de fichier en assurant la bonne
2     execution. Retourne le descripteur de fichier associe. */
3  int safeopen(char *nom, int flags, mode_t mode) {
4      int desc_open;
5      if (mode) {
6          desc_open = open(nom, flags, mode);
7      } else {
8          desc_open = open(nom, flags);
9      }
10     if (desc_open == -1) {
11         fprintf(stderr, "Erreur a l'ouverture de %s", nom);
12         exit(EXIT_FAILURE);
13     }
14     return desc_open;
15 }
16 /* Fermer le fichier de descripteur 'desc' en verifiant la bonne fermeture. 'nom'
17     est utilise en cas d'erreur. */
18 void safeclose(int desc, char *nom) {
19     int desc_close = close(desc);
20     if (desc_close == -1) {
21         fprintf(stderr, "Erreur a la fermeture du descripteur %s", nom);
22         exit(EXIT_FAILURE);
23     }
24 }
25 /* Dupliquer (cf 'dup2') en assurant la bonne execution. 'nom' est utilise en cas d'
26     erreur. */
27 void safedup2(int oldfd, int newfd, char *nom) {
28     if (dup2(oldfd, newfd) == -1) {
29         fprintf(stderr, "Erreur au dup %s", nom);
30         exit(EXIT_FAILURE);
31     }
32 }
33 /* Executer 'cmd0' avec 'execvp' en s'assurant de la bonne execution. */
34 void safeexecvp(char *cmd0, char *const argv[]) {
35     if (execvp(cmd0, argv) == -1) {
36         fprintf(stderr, "Commande inconnue : %s", cmd0);
37         exit(EXIT_FAILURE);
38     }
39 }
```

```

40
41 /* Creer un tube en s'assurant du succes de l'operation */
42 void safepipe(int tube[2]) {
43     if (pipe(tube) == -1) {
44         perror("Erreur lors de la creation du tube");
45         exit(EXIT_FAILURE);
46     }
47 }

```

Listing 1: Code des fonctions "sûres"

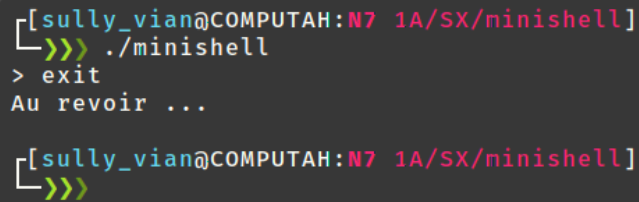
Dans les listings, je pourrais ne pas montrer ces processus de gestion d'erreur afin de garder un code concis. Toutefois, je les ai utilisé autant que possible dans le code de mon projet.

### 3 Méthodologie des tests

Pour les étapes où des changements majeurs sont apportés au code, j'ai pris soin de vérifier que toutes les fonctionnalités précédentes fonctionnaient toujours. Par exemple, après l'étape 20, j'ai vérifié que les redirections classiques (`cmd < file` et `cmd > file`) étaient bien effectuées. De même, j'ai testé que les redirections et pipelines en arrière plan fonctionnaient correctement.

## 4 Étapes

### Étape 1 (Testez le programme)



```
[sully_vian@COMPUTAH:N7 1A/SX/minishell]
>>> ./minishell
> exit
Au revoir ...

[sully_vian@COMPUTAH:N7 1A/SX/minishell]
>>>
```

Figure 1: Test du programme

### Étape 2 (Lancement d'une commande)

C'est le fils qui exécute la commande.

```
1 /* ... */
2 case 0: /* code du fils */
3     execvp(cmd[0], cmd);
4     break;
5 /* ... */
```

Listing 2: Code de la question 2

Sur la figure 3, on voit un exemple de recouvrement: le processus père, responsable de l'affichage du prompt, n'attend pas la terminaison du fils (ici la commande `ls`) pour écrire `>`.

```
[sully_vian@COMPUTAH:N7 1A/SX/minishell]
>>> ./minishell
> ls
> complements  minishell.o      readcmd.c
french.h       _minted-rapport    readcmd.h
Makefile       rapport.pdf      readcmd.o
minishell      rapport.synctex.gz resources
minishell.c    rapport.tex      test_readcmd.c
```

Figure 2: Exemple du lancement de la commande `ls`

### Étape 3 (Enchaînement séquentiel des commandes)

Pour que le père attende la terminaison du fils, il suffit d'ajouter la commande bloquante `wait(null)` dans le code du père tel qu'indiqué dans le listing 1.

```
1 /* ... */
2 default: /* code du pere */
3     wait(null); // attendre terminaison du fils
4     break;
5 /* ... */
```

Listing 3: Ajout pour la question 3

La figure 3 montre un exemple de lancement de la commande `ls` avec un père patient.

```
[sully_vian@COMPUTAH:N7 1A/SX/minishell]
>>> ./minishell
> ls
complements  minishell.o      readcmd.c
french.h     _minted-rapport  readcmd.h
Makefile     rapport.pdf      readcmd.o
minishell    rapport.synctex.gz resources
minishell.c  rapport.tex      test_readcmd.c
> █
```

Figure 3: Exemple du lancement de la commande `ls` avec un père patient

#### Étape 4 (Lancement de commandes en tâche de fond)

Pour ne pas attendre la terminaison d'un fils mis en arrière plan, il suffit de vérifier si le champ `backgrounded` de la structure `commande` est `NULL`. Si c'est le cas, la commande est en avant plan et on l'attend comme implémenté dans le listing 2.

```
1 default: /* code du pere */
2     if (commande->backgrounded == NULL) {
3         // on n'attend pas la terminaison d'un fils mis en arriere plan
4         wait(NULL);
5     }
```

Listing 4: Ajout pour la question 4

Pour vérifier que le père n'attend pas la terminaison d'un fils en arrière plan, on peut lancer la commande `sleep 5 &` et voir que le prompt est affiché directement, sans attendre avant la fin du sommeil.

## Étape 5 (Rendu)

Rien à faire.

## Étape 6 (Traitement du signal SIGCHLD)

```
1  /* ... */
2  void traitement(int sig) {
3      switch (sig) {
4          case SIGCHLD:
5              printf("Un fils vient de se terminer\n");
6              break;
7          default:
8              printf("Signal inconnu\n");
9              break;
10     }
11 }
12 /* ... */
13 int main(void) {
14     struct sigaction action;
15     action.sa_handler = traitement;
16     sigemptyset(&action.sa_mask);
17     action.sa_flags = SA_RESTART;
18     sigaction(SIGCHLD, &action, NULL);
19 }
20 /* ... */
```

Listing 5: ajout de la question 6

## Étape 7 (Utilisation de SINGINT pour traiter la terminaison des processus fils)

```
1  /* ... */
2  void traitement(int sig) {
3      pid_t pid;
4      switch (sig) {
5          case SIGCHLD:
6              pid = waitpid(-1, NULL, WNOHANG | WUNTRACED | WCONTINUED);
7              printf("sortie du processus de pid = %d\n", pid);
8              break;
9          default:
10             printf("Signal inconnu\n");
11             break;
12     }
13 }
14 /* ... */
```

Listing 6: ajout de la question 7

## Étape 8 (Attendre un signal : pause)



```

1 /* ... */
2 default: /* code du pere */
3     pause(); // attendre un signal
4     break;
5 /* ... */

```

Listing 7: ajout de la question 8

```

[sully_vian@COMPUTAH:N7 1A/SX/minishell]
>>> ./minishell
> sleep 10 &
> sleep 50

sortie du processus de pid = 201214
>
sortie du processus de pid = 201303

```

Figure 4: Démonstration de la terminaison en arrière plan du premier `sleep`

## Étape 9 (Suspension et reprise d'un processus en arrière plan)

Lors de l'envoi du signal `SIGSTOP` à un processus en arrière plan, ce processus est mis en pause et passe à l'état "stopped". Lors de l'envoi du signal `SIGCONT`, le processus reprend son exécution et passe de nouveau à l'état "running".

## Étape 10 (Affichage d'un message indiquant le signal reçu)

```

1 /* ... */
2 void traitement(int sig) {

```

```

3  pid_t pid;
4  switch (sig) {
5      case SIGCHLD:
6          pid = waitpid(-1, null, WNOHANG | WUNTRACED | WCONTINUED);
7
8          if (WIFEXITED(status)) {
9              printf("\nsortie du processus de pid = %d\n", pid);
10             }
11             if (WIFSIGNALED(status)) {
12                 printf("\ntermination du processus de pid = %d par le signal %d\n",
pid, sig);
13             }
14             if (WIFSTOPPED(status)) {
15                 printf("\ninterruption du processus de pid = %d\n", pid);
16             }
17             if (WIFCONTINUED(status)) {
18                 printf("\nreprise du processus de pid = %d\n", pid);
19             }
20             break;
21
22             default:
23                 printf("autre signal\n");
24                 break;
25         }
26     }
27     /* ... */

```

Listing 8: ajout de la question 10

## Étape 11 (Rendu)

Rien à faire.

## Étape 12 (Test de la frappe au clavier de ctrl-c et ctrl-z)

Lorsque le minishell est lancé, les signaux SIGINT et SIGTSTP causent l'arrêt du processus père ainsi que celui de chacun de ses fils.

## Étape 13 (Gestion de la frappe au clavier de ctrl-c et ctrl-z)

### Étape 13.1 (Changer le traitement des signaux SIGINT et SIGTSTP)

```

1  /* ... */
2  void traitement(int sig) {
3      switch (sig) {
4          case SIGINT:
5              printf("\n[SIGINT]\n");
6              break;
7
8          case SIGTSTP:
9              printf("\n[SIGTSTP]\n");
10             break;
11     }
12     /* ... */

```

Listing 9: ajout de la question 13.1

```
[sully_vian@COMPUTAH:N7 1A/SX/minishell]
>>> ./minishell
> sleep 10
^C
[SIGINT]
>
terminaison du processus de pid = 245043 par le signal 17
sleep 10
^Z
interruption du processus de pid = 245617

[SIGSTP]
>
```

Figure 5: Test de la frappe au clavier de `ctrl-c` et `ctrl-z`

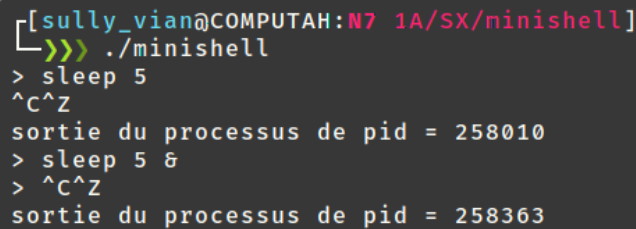
Étape 13.2 (Ignorer les signaux SIGINT et SIGTSTP)

Étape 13.3 (Masquer les signaux SIGINT et SIGTSTP)

```
1  int main(void) {
2      /* ... */
3      sigset_t mask;
4      sigemptyset(&mask);
5      sigaddset(&mask, SIGINT);
6      sigaddset(&mask, SIGTSTP);
7      sigprocmask(SIG_BLOCK, &mask, NULL);
8      /* ... */
9  }
```

Listing 10: ajout de la question 13.3

Sur la figure 6, on voit que le `minishell` ne réagit plus aux signaux `SIGINT` et `SIGTSTP`.



```
[sully_vian@COMPUTAH:N7 1A/SX/minishell]
>>> ./minishell
> sleep 5
^C^Z
sortie du processus de pid = 258010
> sleep 5 &
> ^C^Z
sortie du processus de pid = 258363
```

Figure 6: Test de la frappe au clavier de `ctrl-c` et `ctrl-z`

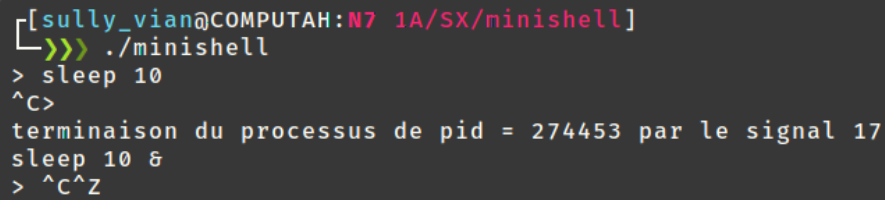
## Étape 14 (Détacher les processus fils en arrière plan)

En enlevant le masquage réalisé juste au dessus (je l'ai commenté dans mon code) et en détachant les processus fils en arrière plan (à l'aide de `setpgrp`), les signaux `SIGINT` et `SIGTSTP` n'atteignent plus le processus père et ses fils en arrière plan.

```
1 /* ... */
2 case 0: /* code du fils */
3     if (commande->backgrounded != NULL) {
4         // changer de groupe si en arriere plan
5         setpgrp();
6     }
7 /* ... */
```

Listing 11: ajout de la question 14

Dans la figure 7, on voit qu'un processus en avant plan est interrompu mais qu'un processus en arrière plan n'est pas affecté par les signaux `SIGINT` et `SIGTSTP`. Le `minishell`, lui, continue de fonctionner normalement.

A terminal window with a dark background. The prompt is [sully\_vian@COMPUTAH:N7 1A/SX/minishell]. The user enters >>> ./minishell. The prompt changes to >. The user enters sleep 10. The prompt changes to ^C>. The terminal displays 'terminaison du processus de pid = 274453 par le signal 17'. The user enters sleep 10 &. The prompt changes to >. The user enters ^C^Z.

```
[sully_vian@COMPUTAH:N7 1A/SX/minishell]
>>> ./minishell
> sleep 10
^C>
terminaison du processus de pid = 274453 par le signal 17
sleep 10 &
> ^C^Z
```

Figure 7: Test de la frappe au clavier de `ctrl-c` et `ctrl-z`

## Étape 15 (Rendu)

Rien à faire.

## Étape 16 (Redirections)

Pour cette étape, il faut ajouter à la fois la gestion d'entrée et de sortie. Ces deux étapes sont similaires, il suffit de remplacer le descripteur de fichier standard d'entrée (`STDIN_FILENO`) par le descripteur du fichier d'entrée et le descripteur de sortie standard (`STDOUT_FILENO`) par le descripteur du fichier de sortie.

Dans les deux cas, il faut :

1. ouvrir le fichier avec `open`
2. dupliquer le descripteur de fichier sur celui voulu (entrée / sortie) avec `dup2`
3. fermer le descripteur de fichier ouvert avec `close`

À chacune de ces étapes, on vérifie que l'opération s'est bien déroulée pour éviter les mauvaises surprises.

```
1 case 0: /* code du fils */
2     /* ... */
3     /* remplacer l'entree standard par commande->in */
4     char *in = commande->in;
5     if (in != NULL) { /* cmd < file */
6         int in_desc;
7         if ((in_desc = open(in, O_RDONLY)) == -1) {
8             fprintf(stderr, "Erreur a l'ouverture de %s", in);
9             exit(EXIT_FAILURE);
10        }
11        if (dup2(in_desc, STDIN_FILENO) == -1) {
12            fprintf(stderr, "Erreur au dup in");
13            exit(EXIT_FAILURE);
14        }
15        if (close(in_desc) == -1) {
16            fprintf(stderr, "Erreur a la fermeture du descripteur in");
17            exit(EXIT_FAILURE);
18        }
19    }
20
21    /* remplacer la sortie standard par commande->out */
22    char *out = commande->out;
23    if (out != NULL) { /* cmd > file */
24        int out_desc;
25        if ((out_desc = open(out, O_WRONLY|O_CREAT|O_TRUNC, 0644)) == -1) {
26            fprintf(stderr, "Erreur a l'ouverture de %s", out);
27            exit(EXIT_FAILURE);
28        }
29        if (dup2(out_desc, STDOUT_FILENO) == -1) {
30            fprintf(stderr, "Erreur au dup out");
31            exit(EXIT_FAILURE);
32        }
33        if (close(out_desc) == -1) {
34            fprintf(stderr, "Erreur a la fermeture du descripteur out");
35            exit(EXIT_FAILURE);
36        }
37    }
```

Listing 12: gestion des redirections

Voyez dans la figure 8 un exemple de ces redirections utilisant les commandes `echo` et `cat`.

```
[sully_vian@COMPUTAH:N7 1A/SX/minishell]
>>> ./minishell
> echo [texte] > f1
sortie du processus de pid = 317522
> cat < f1 > f2
sortie du processus de pid = 317765
> cat f2
[tecte]
sortie du processus de pid = 318074
> █
```

Figure 8: Test de la redirection de l'entrée et de la sortie

Le texte [texte] est bien écrit dans `f1` à la première commande, puis, le contenu de `f1` est écrit dans `f2` à la deuxième commande. La troisième commande vérifie que le contenu de `f2` est bien celui prévu.

### Étape 17 (Rendus 2)

Rien à faire.

### Étape 19 (Tubes simples)

Voir l'étape 20.

### Étape 20 (Pipelines)

Une pipeline est une liste de commandes séparées par des tubes (`|`). La sortie de chaque commande est redirigée vers l'entrée de la commande suivante. Pour implémenter cela, il suffit de savoir si la commande actuelle est la première ou la dernière du pipeline et de mémoriser les descripteurs de fichier du tube d'une commande à l'autre.

Il est facile de savoir si une commande est la dernière d'un pipeline, le champ `seq` de `struct cmdline` se termine toujours par NULL. On teste donc la nullité de la commande suivante.

En revanche, pour savoir si une commande est la première d'un pipeline, j'ai introduit une nouvelle variable entière `dernier_lu`. Cette variable garde en mémoire le descripteur de fichier de la sortie de la commande précédente.

Avec ces deux informations, il est facile de chaîner les commandes entre elles, en remplaçant simplement les entrées / sorties des commandes par les descripteurs de fichier des tubes de la même manière que pour les redirections de l'étape 16.

```

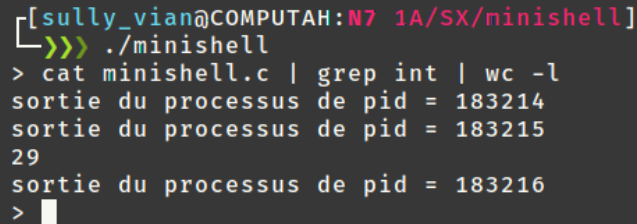
1 while((cmd = commande->seq[indexseq])) {
2     // tant qu'on n'est pas a la fin du pipeline
3     /* ... */
4     else {
5         int tube[2];
6         if (commande->seq[indexseq + 1] != NULL) {
7             // s'il y a une commande suivante, creer un tube
8             safepipe(tube);
9         }
10        switch (pid_fils = fork()) {
11            /* ... */
12            case 0: /* code du fils */
13                /* ... */
14                // redirection des tubes
15                if (dernier_lu != -1) {
16                    // si ce n'est pas la premiere commande, fixer l'entree standard
17                    // a la sortie du tube precedent et fermer la sortie du tube
18                    safedup2(dernier_lu, STDIN_FILENO, "in");
19                    safeclose(dernier_lu, "in");
20                }
21                if (commande->seq[indexseq + 1] != NULL) {
22                    // si ce n'est pas la derniere commande, fixer la sortie
23                    // standard a l'entree du tube et fermer l'entree du tube
24                    safeclose(tube[0], "dernier_lu2");
25                    safedup2(tube[1], STDOUT_FILENO, "tube[1]");
26                    safeclose(tube[1], "dernier_lu3");
27                }
28                /* ... */
29                break;
30            default: /* code du pere */
31                if (dernier_lu != -1) {
32                    // si un tube est ouvert, le fermer (le pere n'en a pas besoin)
33                    safeclose(dernier_lu, "dernier_lu4");
34                }
35                if (commande->seq[indexseq + 1] != NULL) {
36                    // s'il y a une commande suivante, fermer la sortie (pas besoin
37                    // dans le pere)
38                    safeclose(tube[1], "dernier_lu5");
39                    // sauvegarder la sortie du tube pour la prochaine commande
40                    dernier_lu = tube[0];
41                }
42                /* ... */
43            }
44        }
45    }
46 }

```

Listing 13: gestion des pipelines



Dans la figure 9, on voit un exemple de pipeline avec les commandes `cat`, `grep` et `wc -l`. On compte le nombre de lignes de `minishell.c` contenant le mot `int`. `cat` affiche le contenu du fichier, `grep` filtre les lignes contenant le mot `int` et `wc -l` compte le nombre de lignes.



```
[sully_vian@COMPUTAH:N7 1A/SX/minishell]
>>> ./minishell
> cat minishell.c | grep int | wc -l
sortie du processus de pid = 183214
sortie du processus de pid = 183215
29
sortie du processus de pid = 183216
> █
```

Figure 9: Test d'un pipeline

On compte donc 29 lignes différentes contenant au moins une fois le mot `int` dans le fichier `minishell.c`. On remarque aussi que les deux premières commandes s'arrêtent avant l'affichage du résultat. C'est cohérent avec l'implémentation réalisée.

## Étape 21 (Rendu)

Rien à faire.

## 5 Codes complémentaires

Dans le sujet, il nous est demandé de réaliser des codes complémentaires. J'ai placé ces codes dans le dossier `complement`.

## 5.1 `copier.c`

Ce programme est équivalent à la commande `cp` de Unix. Il se déroule en 3 étapes:

1. ouvrir le fichier source en lecture seule et le fichier destination en écriture seule.
2. boucler pour lire le contenu du fichier source fragment par fragment (de taille `BUFSIZE`) et écrire le fragment dans le fichier destination. Cette boucle s'arrête lorsqu'il n'y a plus rien à lire.
3. fermer les fichiers source et destination.

## 5.2 `tube1.c`

Ce programme ne peut pas fonctionner car le fils ne connaît pas le tube créé par le père. Le fils a un descripteur de fichier identique à celui du père au moment du `fork`, mais ces copies ne sont pas partagées.

## 5.3 `tube2.c`

Ce programme fonctionne correctement, le fils affiche bien l'entier écrit par le père.

## 5.4 `tube3.c`

Ce programme ne fonctionne pas comme espéré. Le père n'écrit plus dans le tube donc le fils n'a plus rien à y lire. Or, `read` est bloquant. Le fils reste donc bloqué dans la boucle. Le père non plus ne s'arrête pas car il attend un signal (`pause`).

## 5.5 `tube4.c`

Cette fois, le programme se termine bien parce que le père se termine après 10 secondes. En revanche, le fils n'affiche toujours pas de message puisque `read` est toujours bloquant. Il reste encore coincé dans la boucle.

## 5.6 `tube5.c`