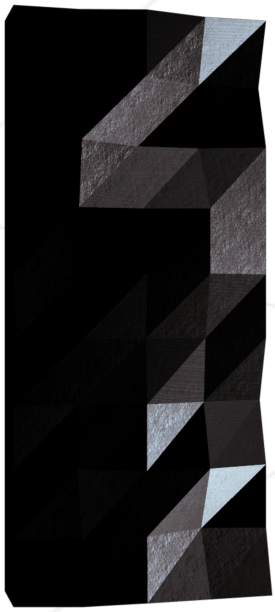

Pipex

Résumé: Ce projet est la découverte dans le détail et par la programmation d'un mécanisme d'UNIX que vous connaissez déjà.

Version: 2

Table des matières

Chapitre I

Préambule

Cristina : `"Allez danser la salsa quelque part :)"`

Chapitre II

Règles communes

- Votre projet doit être écrit en C.
- Votre projet doit être codé à la Norme. Si vous avez des fichiers ou fonctions bonus, celles-ci seront inclues dans la vérification de la norme et vous aurez 0 au projet en cas de faute de norme.
- Vos fonctions de doivent pas s'arrêter de manière inattendue (segmentation fault, bus error, double free, etc) mis à part dans le cas d'un comportement indéfini. Si cela arrive, votre projet sera considéré non fonctionnel et vous aurez 0 au projet.
- Toute mémoire allouée sur la heap doit être libéré lorsque c'est nécessaire. Aucun leak ne sera toléré.
- Si le projet le demande, vous devez rendre un Makefile qui compilera vos sources pour créer la sortie demandée, en utilisant les flags `-Wall`, `-Wextra` et `-Werror`. Votre Makefile ne doit pas relink.
- Si le projet demande un Makefile, votre Makefile doit au minimum contenir les règles `$(NAME)`, `all`, `clean`, `fclean` et `re`.
- Pour rendre des bonus, vous devez inclure une règle `bonus` à votre Makefile qui ajoutera les divers headers, librairies ou fonctions qui ne sont pas autorisées dans la partie principale du projet. Les bonus doivent être dans un fichier différent : `_bonus.{c/h}`. L'évaluation de la partie obligatoire et de la partie bonus sont faites séparément.
- Si le projet autorise votre `libft`, vous devez copier ses sources et son Makefile associé dans un dossier libft contenu à la racine. Le Makefile de votre projet doit compiler la librairie à l'aide de son Makefile, puis compiler le projet.
- Nous vous recommandons de créer des programmes de test pour votre projet, bien que ce travail **ne sera pas rendu ni noté**. Cela vous donnera une chance de tester facilement votre travail ainsi que celui de vos pairs.
- Vous devez rendre votre travail sur le git qui vous est assigné. Seul le travail déposé sur git sera évalué. Si Deepthought doit corriger votre travail, cela sera fait à la fin des peer-evaluations. Si une erreur se produit pendant l'évaluation Deepthought, celle-ci s'arrête.

- Le fichier exécutable doit être nommé `pipex`.
- Vous devez gérer les erreurs avec du bon sens. En aucun cas votre programme peut-il se fermer de manière inattendue (Segmentation défaut, bus error, double free, etc.). Si vous n'êtes pas sûr, gérer les erreurs comme la commande shell originale `< file1 cmd1 | cmd2 > file2`.
- Votre programme ne peut pas avoir de fuites mémoire.
- Vous êtes autorisé à utiliser les fonctions suivantes :
  - `access`
  - `open` **OK**
  - `unlink`
  - `close` **OK**
  - `read` **OK**
  - `write` **OK**
  - `malloc` **OK**
  - `waitpid` **Wait for specific PID**
  - `wait` **wait(NULL) == wait for one child; you can also use the wait(int_of_your_choice) to get the output status of the child process. You can then pass that in into a macro like WIFEXITED or WEXITSTATUS WIFEXITED returns true if the child terminated normally and WEXITSTATUS return the exit status of the child**
  - `free` **OK**
  - `pipe` **Use this to create a link between functions or processes; pipe before forking**
  - `dup` **You can use dup(file) to duplicate the fd It will return another file descriptor which will point to the same thing as the first on you could use on or another for read or write**
  - `dup2` **You can use dup2(file, 1) This function takes 2 parametes The first is fd we actually want to clone The second one is the value we one the new FD to have so if we use dup2(file, 1) instead of creating a new value its gonna take a look at fd1 and close this stream (stdout) then its gonna open it again to our ping result This way we finally duplicate the new created FD into the stdout**
  - `execve` **Use to open an executable with its absolute path, Execve("parampath", vector1, env) the vector1 is composed of the executable , then the arguments that you would pass to that executable. The executable overruns the actual c program**
  - `fork`
  - `perror`
  - `strerror`
  - `exit`

Chapitre III

Objectifs

Votre objectif est de coder le programme Pipex.
Il doit être exécuté de cette manière :

```
$> ./pipex file1 cmd1 cmd2 file2
```

file1 et file2 sont des noms de fichiers.
cmd1 et cmd2 sont des commandes shell avec leurs paramètres.

L'exécution du programme pipex a le même effet que la commande shell suivante :

```
$> < file1 cmd1 | cmd2 > file2
```

III.1 Exemples

```
$> ./pipex infile ``ls -l'' ``wc -l'' outfile
```

devrait être le même que "< infile ls -l | wc -l > outfile"

```
$> ./pipex infile ``grep a1'' ``wc -w'' outfile
```

devrait être le même que "< infile grep a1 | wc -w > outfile"

Chapitre IV

Bonus part

Les bonus ne seront évalués que si votre partie obligatoire est PARFAIT. Par PARFAIT, nous entendons naturellement qu'il doit être complete, qu'il ne peut échouer, même en cas d'erreurs comme de mauvaises utilisations, etc. Cela signifie que si votre partie obligatoire n'obtient pas TOUS les points lors de la notation, vos bonus seront entièrement IGNORÉ..

- Gérer plusieurs pipes :

```
$> ./pipex file1 cmd1 cmd2 cmd3 ... cmdn file2
```

Doit être équivalent à :

```
< file1 cmd1 | cmd2 | cmd3 ... | cmdn > file2
```

- Supporte « et » lorsque le premier paramètre est "here_doc"

```
$> ./pipex here_doc LIMITER cmd cmd1 file
```

Doit être équivalent à :

```
cmd << LIMITER | cmd1 >> file
```

Chapitre V

Rendu et peer-évaluation

Rendez-votre travail sur votre dépot `GiT` comme d'habitude. Seul le travail présent sur votre dépot sera évalué en soutenance.