Faculté des Sciences
Département d'Informatique

INFO-F-201 – Systèmes d'exploitation

Projet 1 de programmation système

Chat – édition processus & pipe

1 Détails du projet

Ce projet est à réaliser par groupe de **trois** étudiants et comporte différentes parties :

- La création du programme C ou C++ « `chat` » permettant à deux utilisateurs de discuter à l'aide de pipes nommés ;
- La création du script Bash « `chat-bot` » qui pourra remplacer un utilisateur et effectuer certaines requêtes de son interlocuteur.

2 Programme C ou C++ : `chat`

2.1 Objectif

Le but de ce programme est de créer une messagerie qui doit :

- Être écrite en C ou en C++ ;
- Avoir son code dans le dossier *src/* ;
- Être compilé à l'aide d'un Makefile (vous pouvez modifier celui fourni) ;
- Communiquer à l'aide de *pipes* nommés – voir Section 2.4 ;
- Gérer les signaux SIGPIPE et SIGINT – voir Section 2.7.

La communication se fera à l'aide de deux pipes nommés, un pour envoyer des messages et un pour en recevoir.

2.2 Gestion de paramètres

Les paramètres du programme `chat` suivront le format suivant :

```
chat pseudo_utilisateur pseudo_destinataire [--bot] [--manuel]
```

où :

- `pseudo_utilisateur` : est le pseudonyme (max 30 octets) que l'utilisateur lançant le programme utilisera pour sa communication ;
- `pseudo_destinataire` : est le pseudonyme (max 30 octets) de la personne avec qui les messages seront échangés ;
- `--bot` : si ce paramètre (optionnel) est présent, le texte ne sera pas coloré ni souligné (voir la Section 2.3.2) ;
- `--manuel` : si ce paramètre (optionnel) est présent, les messages ne seront pas affichés automatiquement. Leur arrivée sera notifiée par un bip et ne seront affichés que dans certaines situations. Voir Section 2.3.3 pour plus de détails.

Les paramètres correspondant aux pseudonymes seront toujours écrits en premier et deuxième. Les paramètres optionnels peuvent être ensuite présents dans n'importe quel ordre (ou être ab-

sents) mais ils ne seront jamais écrits en 1re ni 2e position (s'ils le sont, ils seront traités comme des pseudonymes).

S'il manque les pseudonymes comme paramètres, le programme devra se terminer avec un code de retour de 1 en affichant, sur la sortie standard d'erreur (`stderr`), le message « *chat pseudo_utilisateur pseudo_destinataire [--bot] [--manuel]* ».

Si au moins un des pseudonymes est plus long que 30 caractères, terminez le programme avec le code de retour 2 et en affichant un message d'erreur, adapté, de votre choix sur `stderr`.

Si au moins un des pseudonymes contient un ou plusieurs des caractères suivants, terminez le programme avec le code de retour 3 et en affichant un message d'erreur adapté, de votre choix, sur `stderr`.

```
/ - [ ]
<slash> <trait-d'union> <crochet-ouvrant> <crochet-fermant>
```

Faites de même si le pseudo est « `.` » ou « `..` ».

Vous pouvez choisir comment traiter des paramètres autres que ceux spécifiés dans cette section.

2.3 Affichage des messages

2.3.1 Affichage par défaut

Par défaut, lorsqu'un message est envoyé ou reçu, celui-ci sera affiché sur la sortie standard (`stdout`) sous la forme suivante :

```
[PSEUDO] MESSAGE
```

avec « `PSEUDO` » remplacé par le pseudonyme de l'émetteur et « `MESSAGE` » par le message envoyé ou reçu.

Les pseudonymes sont, pour l'affichage par défaut, soulignés (mais pas les crochets qui les entourent). Pour souligner du texte sur un terminal, vous pouvez ajouter « \x1B[4m » avant le texte à souligner et « \x1B[0m » après le texte à souligner. Par exemple, pour souligner le texte « LINUX » dans un `printf`, il suffit de faire :

```
printf("\x1B[4mLINUX\x1B[0m");
```

Ce qui affichera « LINUX » sur le terminal.

Vous pouvez également colorer les pseudonymes si vous le souhaitez (mais il n'y aura pas de points bonus si c'est fait). Vous pouvez regarder du côté des séquences d'échappement ANSI [1] si vous êtes curieux.

2.3.2 Mode `--bot`

Lorsque l'option --bot est activée, l'ajout du soulignement des pseudonymes (ainsi que des couleurs, si vous avez décidé d'en utiliser) est désactivé. De plus, les messages envoyés par l'utilisateur (lus sur `stdin`) ne sont pas affichés sur `stdout`.

1. https://en.wikipedia.org/wiki/ANSI_escape_code

2.3.3 Mode `--manuel`

Lorsque l'option --manuel est activée, les messages de l'interlocuteur ne seront plus affichés au moment de leur réception. Ils ne seront affichés que dans un des 3 cas suivants :

- si le signal `SIGINT` est reçu (utilisation de Ctrl + C) ;
- si un message est envoyé par l'utilisateur, les messages en attente sont alors affichés après l'affichage du message envoyé (ou directement si l'option --bot est activée) ;
- si plus de 4096 octets sont en attente d'être affichés [2].

Lorsqu'un message est reçu, vous devrez cependant écrire directement le caractère `'\a'` sur `stdout`, ce qui émettra un bip sonore (si ce comportement n'est pas désactivé dans les options du terminal).

2.4 Gestion des pipes

Pour échanger des messages, le programme `chat` **doit** utiliser des *pipes* nommés (les autorisations d'accès aux *pipes* sont `0666`). Le nom (c.-à-d., chemin d'accès) du *pipe* dépendra du pseudonyme des deux utilisateurs qui s'échangeront les messages. Le format sera le suivant :

```
/tmp/ECRIVAIN-LECTEUR.chat
```

où `ECRIVAIN` correspond au pseudonyme de l'utilisateur qui envoie un message et `LECTEUR` correspond au pseudonyme de l'utilisateur qui reçoit un message.

Supposons que nous soyons « alice » et que nous souhaitions discuter avec « bob ». Nos messages seront donc écrits sur le *pipe* nommé ouvert en `/tmp/alice-bob.chat` et nous pourrons, par conséquent, lire les messages de « bob » via le *pipe* nommé ouvert en `/tmp/bob-alice.chat` (car c'est là que « bob » y écrira ses messages nous étant destinés).

Lorsque la conversation est terminée, les *pipes* nommés doivent être supprimés.

2.5 Processus

Afin de permettre à l'utilisateur d'envoyer et recevoir des messages simultanément, le programme `chat` utilisera **exactement 2 processus** :

1. le processus d'origine : lancé initialement en exécutant le programme `chat` ;
2. le second processus : créé par le processus d'origine.

Le processus d'origine doit lire les messages sur l'entrée standard (`stdin`) et les transmettre au destinataire via le *pipe* nommé adéquat (voir Section 2.4).

Le second processus, créé par celui d'origine, lit sur le *pipe* nommé adéquat les messages reçus de l'autre utilisateur et se charge de les afficher.

Cependant, si l'option --manuel est activée, vous pouvez choisir quel processus se charge d'afficher les messages en fonction de la situation dans laquelle l'affichage a été demandé. **Justifiez dans le rapport quel processus affiche les messages en fonctions des 3 situations d'affichage énumérées à la Section 2.3.3.**

De plus, lorsque l'option --manuel est activée, les messages destinés à l'utilisateur devront également être conservés en mémoire partagée jusqu'à leur affichage. Plus d'informations à la Section 2.6.

2. Vous n'avez donc pas à conserver plus de 4096 octets en mémoire avant affichage.

2.6 Mémoire partagée

Lorsque l'option --manuel est activée, les messages reçus qui n'ont pas encore été affichés doivent être conservés dans une mémoire partagée. Cette mémoire partagée doit être accessible uniquement par le processus d'origine et son enfant (c.-à-d., le second processus).

Pour rappel, les conditions pour afficher les messages de l'utilisateur (lorsque l'option --manuel est activée) sont énumérées à la Section 2.3.3.

Vous pouvez choisir quel type de mémoire partagée utiliser et en quelle quantité. **Justifiez ces deux choix dans le rapport.**

Vous ne devez pas gérer les risques d'accès concurrents à la mémoire partagée.

2.7 Gestion des signaux

Le programme `chat` doit au moins prendre en compte les signaux `SIGPIPE` et `SIGINT` (vous êtes libres d'en gérer d'autres si c'est pertinent mais **précisez-le dans le rapport**).

Le traitement du signal `SIGINT` pourra varier durant l'exécution du programme. Pour ce signal, les processus le traiteront ainsi :

- tant que les *pipes* nommés n'ont pas été ouverts, le signal `SIGINT` doit terminer proprement `chat` (et donc à la fois le processus d'origine et le second) avec le code de retour 4 ;
- si les *pipes* nommés ont été ouverts et que l'option --manuel est activée, alors la réception du signal `SIGINT` par le processus d'origine devra désormais afficher les messages en attente conservés en mémoire partagée.

Pour le second processus, le signal `SIGINT` doit toujours être ignoré, peu importe les options activées.

2.8 Fin du programme

Le programme doit se terminer dans les différentes circonstances suivantes :

1. en cas de paramètres invalides (voir Section 2.2) ;
2. en cas de fin normale du programme (avec alors un code de retour de 0) :
   - si l'entrée standard (`stdin`) est close ;
   - si le *pipe* nommé pour envoyer des messages est clos (déconnexion de l'interlocutrice ou interlocuteur).
3. dans certains cas à la réception d'un `SIGINT` (voir Section 2.7) ;
4. en cas d'échec critique d'un appel système essentiel ou d'allocation d'une ressource (à vous de déterminer si l'erreur indiquée est critique ou non), avec alors un code de retour de votre choix.

2.9 Illustration de `chat` et ses IPC

L'illustration est disponible à la Figure 1.

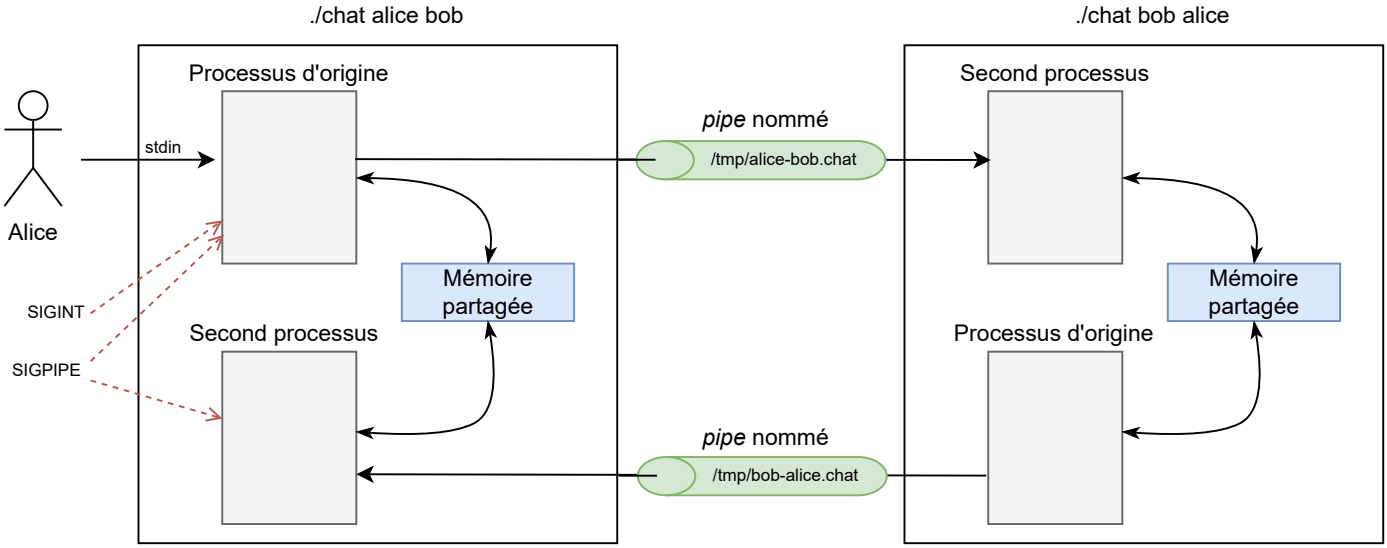

Figure 1 – Illustration récapitulative des différents processus et leurs moyens de communication pour une discussion entre l'utilisatrice « alice » (partie à gauche) et l'utilisateur « bob » (partie à droite, un peu moins détaillée). Attention, la position sur l'image des processus est inversée chez « bob » par rapport à « alice ».

3 Programme Bash : `chat-bot`

3.1 Objectif

Le script `chat-bot` a pour objectif de créer un petit robot conversationnel rudimentaire. Pour ce faire il :

- utilise le programme `chat` (avec l'option --bot activée) pour communiquer ;
- redirige les flux standard de `chat` pour envoyer et recevoir les messages ;
- exécute les commandes demandées par son interlocutrice ou interlocuteur.

L'Annexe A vous sera utile pour réaliser ce script.

3.2 Paramètres

Le script `chat-bot` prend 1 ou 2 paramètres :

1. le pseudonyme du destinataire des réponses du robot ;
2. le pseudonyme utilisé par le robot (optionnel, vaut « bot » par défaut s'il a été omis).

Si le nombre de paramètres est incorrect, le script doit se terminer avec la valeur de terminaison 1 et en affichant sur la sortie standard d'erreur : « chat-bot destinataire [pseudo] ».

Notez que l'ordre des pseudonymes est inversé par rapport au programme `chat`.

3.3 Interface avec `chat`

Pour fonctionner, le script `chat-bot` doit exécuter le programme `chat`. Vous devrez trouver un moyen de rediriger l'entrée standard et la sortie standard de ce programme pour pouvoir simuler un utilisateur normal avec votre script.

Le comportement du programme `chat` avec l'option --bot doit s'en tenir **strictement** à ce qui a été défini dans la Section 2 et ses sous-sections. Vous n'êtes, par exemple, pas autorisés à

écrire des redirections de flux directement en C dans le code source de `chat` parce que l'option --bot a été activée.

La Figure 2 illustre comment le script `chat-bot` s'interface avec le programme `chat` pour communiquer avec une utilisatrice « alice ».

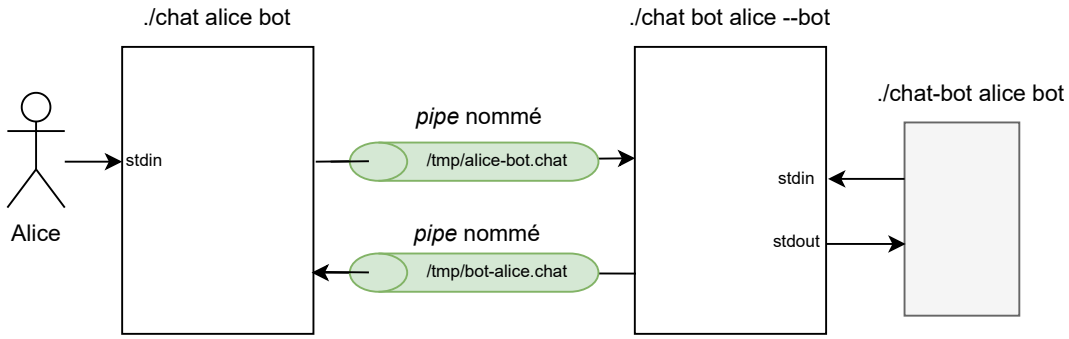

Figure 2 – Illustration de l'interfaçage entre le robot conversationnel `chat-bot` (partie à droite) et l'utilisatrice « alice » (partie à gauche). Une représentation interne plus détaillée du programme `chat` est disponible à la Figure 1.

3.4 Commandes supportées

Le script `chat-bot` exécutera des commandes (sensibles à la casse) en fonction de ce qui lui est demandé. Les commandes supportées seront :

- « liste » : lister tous les fichiers du dossier de travail du robot ;
- « li FICHIER » : lire le contenu du fichier « FICHIER » (ou tout autre nom choisi) ou affiche une erreur en cas d'échec (p.ex., si le fichier est inexistant) ;
- « qui suis-je » : donne le pseudonyme du destinataire ;
- « au revoir » : termine le robot avec le code de retour 0.

Le résultat des commandes est toujours écrit comme un message vers le destinataire de la discussion. Si le résultat est constitué de plusieurs lignes, celui-ci peut être vu comme une suite de plusieurs messages écrits par le robot.

Si la commande n'est aucune de celles listées ci-dessus, le script doit consulter le fichier (fourni) `liste-bot.txt` pour essayer d'y trouver un mot correspondant. Ensuite :

- si la commande a été trouvée, le script répond avec la réponse associée à la commande ;
- si la commande n'a pas été trouvée, il répond le message (encodé en UTF-8) « 🤖 ? » (il y a une espace avant le « ? »).

Le fichier `liste-bot.txt` est structuré par ligne. Le premier mot de chaque ligne correspond à la commande et est suivi d'un ou plusieurs mots qui forment, eux, la réponse attendue. Un mot est vu comme une suite de caractères autres qu'une espace.

4 Ce qui est mis à votre disposition

Vous pouvez télécharger la base du projet sur l'Université Virtuelle. Ce répertoire contient :

- un fichier `Makefile` pour compiler `chat` ;
- un dossier `src/` contenant le fichier `main.c` et où le code source C ou C++ du programme `chat` devra se trouver (vous êtes autorisés à supprimer le fichier `main.c`) ;
- un script vide `chat-bot` pour écrire le code du script `chat-bot` ;

— le fichier `liste-bot.txt` avec les réponses pour `chat-bot`.

5 Critères d'évaluation

Si vous écrivez votre code en C, votre projet doit compiler avec `gcc` version 9.4 (ou ultérieure) et les options ci-dessous (présentes dans le Makefile fourni) :

```
FLAGS=-std=gnu11 -Wall -Wextra -O2 -Wpedantic
```

Si vous écrivez votre code en C++, votre projet doit compiler avec `g++` version 9.4 (ou ultérieure) et les options ci-dessous :

```
FLAGS=-std=gnu++17 -Wall -Wextra -O2 -Wpedantic
```

Si votre programme ne compile pas, vous recevrez une note de 0/20.

N'hésitez pas à utiliser les options pour les assainisseurs lors du développement de votre projet, ceci vous aidera à détecter vos erreurs plus facilement (ceci n'est pas obligatoire et pensez à bien recompiler tous vos fichiers lorsque vous ajoutez ou retirez ces options) :

```
-g -fsanitize=address,undefined
```

Assurez-vous également que **tous** vos appels systèmes ont leur valeur de retour correctement traitée (à vous de **gérer correctement les cas où une erreur est survenue**). Cela fait partie de l'évaluation. Un projet dont le code fonctionne mais ne prend pas compte des erreurs pouvant survenir perdra des points.

Pondération

— Tests automatiques **/3**
— Ce projet de programmation système va principalement évaluer votre compétence à manier correctement les outils liés aux systèmes d'exploitation (processus, pipes, signaux…) dans le langage C ou C++. La pertinence des outils utilisés ainsi que la manière dont ils sont utilisés (trop, pas assez, au mauvais endroit, trop longtemps…) sont évalués. Assurez-vous également que les codes d'erreurs sont bien traités. **/9**
— Ce projet doit contenir un rapport dont la longueur attendue est de deux à trois pages (max 5). Ce rapport décrira succinctement le projet, les choix d'implémentation si nécessaire, les difficultés rencontrées et les solutions originales que vous avez fournies et vos choix d'implémentation. **/6**
  — Orthographe
  — Structure
  — Légende des figures
— Votre code sera aussi évidemment examiné en termes de clarté, de documentation, de commentaires et de structure. **/2**

6 Remise du projet

Vous devez remettre un projet par groupe contenant un fichier zip contenant :

- les codes source C/C++ ;
- le ou les scripts Bash que vous utilisez ;
- un seul Makefile pour compiler le code ;
- votre rapport au format PDF avec le nom des membres du groupe et leur ULBID ;

Ne remettez pas le fichier `liste-bot.txt`.

Vous devez soumettre votre projet sur l'**Université Virtuelle** pour le **24 novembre 2024 23h59** au plus tard.

Retards

Tout retard sera sanctionné d'un point par tranche de 4 h de retard, avec un maximum de 24 h de retard, et le projet devra être soumis sur l'université virtuelle.

Questions

Vous pouvez poser vos questions par courriel à arnaud.leponce@ulb.be ou alexis.reynouarde@ulb.be en commençant l'objet du message par « [INFO-F201 projet] ».

A Conseils

A.1 Fonctions Bash

Vous aurez sûrement besoin d'organiser votre script `chat-bot` en sous-scripts ou fonctions.
Pour définir et appeler une fonction en bash, utilisez la syntaxe suivante :

```
function foo () {
    echo "Paramètre 1: $1"
    echo "Paramètre 2: $2"
}
foo "Hello World" "and goodbye"
```

Ceci affiche :

```
Paramètre 1: Hello World
Paramètre 2: and goodbye
```

Une fois définie, la fonction est utilisable dans le script comme tout autre programme. En particulier, vous pouvez effectuer des redirections sur des appels de fonctions :

```
foo Hello World > output
```

A.2 *Pipes* Bash

Imaginons, à tout hasard, que j'ai une fonction `foo` qui devrait

1. lire l'affichage d'un programme `bar` et
2. y réagir en lui envoyant du texte sur l'entrée standard de ce même programme.

Pour faire le premier point, il suffit d'écrire :

```
bar | foo
```

et pour le deuxième :

```
foo | bar
```

En revanche, faire les deux en même temps est plus complexe.
Pour y arriver, je vous encourage à utiliser la built-in bash `coproc`. Les autres méthodes sont plus difficiles à implémenter de manière sûre et propre.
La commande :

```
coproc BAR_PIPES { bar; }
```

exécute `bar` de façon asynchrone et crée deux variables. Chaque variable contient un *file descriptor* : `${BAR_PIPES[0]}` contient le *file descriptor* d'un *pipe* connecté à la sortie standard de `bar` et `${BAR_PIPES[1]}` est connecté de la même manière à l'entrée standard de `bar`.
De plus, il est possible de faire une redirection en donnant un *file descriptor* plutôt qu'un nom de fichier en écrivant `<&fd` (avec `fd` un *file descriptor*). Par exemple : `foo <&"${fd_var}"` remplace l'entrée standard de `foo` par ce qui est lu via le *file descriptor* contenu dans la variable `fd_var`. De même, `foo >&"${fd_var}"` remplace la sortie standard de `foo`.
Attention, si vous décidez d'utiliser `mkfifo` plutôt que `coproc`, vous devez vous assurer de choisir un chemin non utilisé et de supprimer les pipes à la sortie du script, même si celui-ci est fini par un `CTRL + C` ou une erreur.

A.3 *Buffering*

Pour que votre robot fonctionne, il faudra qu'il reçoive effectivement les messages. Tant que tous les messages affichés par votre programme `chat` se finissent par un retour à la ligne (`\n`), vous ne devriez pas avoir de problèmes. Si, malgré tout, votre bot semble ne pas recevoir les messages, vous avez peut-être un problème de *buffering*.

Comme ce fut mentionné au TP sur les processus, l'utilisation de `printf` en C n'envoi pas immédiatement le texte sur la sortie. Le texte est d'abord conservé dans un buffer. En général, ce buffer est envoyé sur la sortie lorsqu'un retour à la ligne y est ajouté. Mais si ce n'est pas le cas pour vous, le plus facile sera sûrement de forcer l'envoi du buffer en appelant `fflush(stdout)`.