Faculté des Sciences
Département d'Informatique

INFO-F-201 – Systèmes d'exploitation

Projet 2 de programmation système

Chat201 – édition thread & réseau

2024 – 2025

Suite à la réalisation à succès d'un premier chat en local, vous avez décidé de voir plus grand et de passer au chat en réseau ! Avec vos nouvelles connaissances en programmation système, vous avez également décidé de repenser votre chat pour y utiliser des threads et éviter les accès concurrents.

1 Détails du projet

Le projet est à réaliser par groupe de **deux** ou **trois** étudiants. Vous avez la possibilité de changer de groupe, mais vous devez gérer ces changements par vous-même. Vous devez réaliser :

- la création du programme C ou C++ « `chat` » permettant à deux utilisateurs de discuter par l'intermédiaire d'un serveur,
- la création du programme C ou C++ « `serveur-chat` » prenant le rôle du serveur et établissant la communication entre les différents clients et
- la création du script Bash « `chat-auto` » qui facilite l'utilisation du programme `chat`.

Pour vous aider, une solution du premier projet sera publiée sur l'UV. Vous pouvez reprendre tout ou partie de cette solution.

2 Serveur `serveur-chat`

2.1 Objectif

Il s'agit du serveur mettant en contact des clients de messagerie. Il doit :

- être écrit en C ou en C++,
- avoir son code dans le dossier *src/serveur/*,
- être compilé à l'aide d'un Makefile (vous pouvez modifier celui fourni),
- communiquer à l'aide de *sockets* TCP — voir Section 2.2,
- gérer simultanément des clients de la façon de votre choix, choix à détailler dans le rapport — voir Section 2.3 et
- gérer les signaux SIGPIPE et SIGINT — voir Section 2.4.

2.2 Communication via *sockets*

Pour communiquer avec les clients, le serveur utilise des sockets avec le protocole TCP. La connexion est configurée ainsi :

Adresse IP toutes les adresses IP disponibles sur la machine (c.-à-d., `INADDR_ANY`)

Port

- si la variable d'environnement `PORT_SERVEUR` est définie et contient un nombre compris entre 1 et 65 535, alors le port est le nombre contenu dans cette variable,
- sinon, le port est 1234 par défaut.

Longueur de la file d'attente pour les nouvelles connexions entrantes 5

Nombre maximal de clients connectés simultanément 1000 [1]

2.3 Gestion des clients simultanément

Vous devez supposer que le serveur et le client peuvent être sur des machines différentes.

Vous devez pouvoir traiter plusieurs clients en même temps. Vous ne pouvez donc pas attendre qu'un client se déconnecte pour commencer à prendre en compte les requêtes des autres clients. Il n'y a pas de restriction sur la manière de gérer cela (processus, threads, fonctions non-bloquantes, polling...).

Expliquez dans le rapport quelle méthode vous avez choisie pour traiter les clients simultanément. Comparez ce choix (efficacité, aisance d'utilisation, impact sur votre implémentation...) avec une alternative qui aurait été possible.

Pour ce projet, il sera possible de discuter avec plusieurs autres utilisatrices et utilisateurs en même temps (voir Section 3.4). Tout le fonctionnement du `serveur-chat` doit être réalisé de sorte à ce que les accès concurrents aux ressources soient synchronisés (p.ex., via *mutex*, sémaphores, atomiques...).

2.4 Gestion des signaux

Le programme `serveur-chat` doit au moins prendre en compte les signaux `SIGPIPE` et `SIGINT`.

Vous êtes libres de gérer d'autres signaux si c'est pertinent mais précisez et justifiez-le dans le rapport.

Lorsque le `serveur-chat` reçoit le signal `SIGINT`, il devra se terminer proprement en fermant notamment toutes les connexions des différents clients qui lui sont connectés.

En fonction de vos choix d'implémentation, vous aurez peut-être besoin de l'Annexe A, le traitement des signaux avec les threads ajoutant quelques difficultés.

3 Client `chat`

3.1 Objectif

Ce client de messagerie doit :

- être écrit en C ou en C++,
- avoir son code dans le dossier *src/chat/*,

1. Cette hypothèse est là pour vous faciliter la vie, mais vous pouvez augmenter cette limite si vous voulez.

— être compilé à l'aide d'un Makefile (vous pouvez modifier celui fourni),
— communiquer à l'aide d'un *socket* — voir Section 3.5,
— gérer les signaux `SIGPIPE` et `SIGINT` — voir Section 3.7.

La communication se fait à l'aide de *sockets*. Étant donné que le client reste très similaire à celui réalisé au projet 1, certaines sections ou sous-sections sont reprises telles quelles, précédées de la mention « [Inchangée] » pour vous aider à identifier facilement les nouvelles contraintes et exigences.

3.2 Gestion de paramètres

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

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

où :

— `pseudo_utilisateur` : est le pseudonyme (max 30 octets) que l'utilisateur lançant le programme utilise pour sa communication ;
— `--bot` : si ce paramètre (optionnel) est présent, le texte n'est pas coloré ni souligné (voir la Section 3.3.2) ;
— `--manuel` : si ce paramètre (optionnel) est présent, les messages ne sont pas affichés automatiquement. Leur arrivée est notifiée par un bip et ne sont affichés que dans certaines situations. Voir Section 3.3.3 pour plus de détails.

Le paramètre correspondant au pseudonyme est toujours écrit en premier. Les paramètres optionnels peuvent être ensuite présents dans n'importe quel ordre (ou être absents) mais ils ne sont jamais écrits en 1ère position (s'ils le sont, ils sont traités comme des pseudonymes).

Si le programme est lancé sans arguments, il doit se terminer avec un code de retour de 1 en affichant, sur la sortie standard d'erreur (`stderr`), le message « *chat pseudo_utilisateur [--bot] [--manuel]* ».

Si le pseudonyme 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 le pseudonyme 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. Par exemple, vous pouvez ajouter des options si vous le souhaitez afin d'activer des ajouts personnalisés. Cependant, aucun changement du comportement du programme n'est autorisé sans l'utilisation d'une option personnalisée pour les activer.

3.3 Affichage des messages

3.3.1 [Inchangée] Affichage par défaut

Par défaut, lorsqu'un message est envoyé ou reçu, celui-ci est 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 affiche « 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 [2] si vous êtes curieux.

3.3.2 [Inchangée] 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`.

3.3.3 Mode `--manuel`

Lorsque l'option --manuel est activée, les messages de l'interlocuteur ne sont plus affichés au moment de leur réception. Ils ne sont 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 [3].

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

Attention, les accès concurrents doivent être bien gérés cette fois.

3.3.4 Accès concurrents lors de l'affichage

Vous devez garantir que l'affichage des messages soit synchronisé de telle sorte que les messages sur `stdout` ne s'entrelacent pas.

3.4 Envoi des messages

Le `chat` de ce projet ne se contente pas de communiquer avec un seul destinataire et il peut donc y avoir plusieurs conversations à la fois avec différents destinataires. Afin de connaître le destinataire du message écrit par l'utilisatrice ou utilisateur, chaque message sur `stdin` doit être précédé du pseudonyme de son destinataire. Si ce pseudonyme contient des espaces, ces espaces sont remplacées par des tirets (p.ex., le pseudo « chat chat » s'écrira « chat-chat »).

Ainsi, si l'utilisatrice « alice » souhaite envoyer à « bob » le message « Bonjour Bob ! », elle doit écrire sur `stdin` :

2. https://en.wikipedia.org/wiki/ANSI_escape_code
3. Vous n'avez donc pas à conserver plus de 4096 octets en mémoire avant affichage.

```
bob Bonjour Bob !
```

Si le message ne respecte pas ce format (c.-à-d., il n'y a pas au moins deux mots séparés par une espace), alors le message doit être ignoré par `chat`. Si le destinataire n'est pas connecté, le serveur doit l'indiquer au `chat` de l'utilisatrice « alice » qui affichera alors sur `stderr` :

```
Cette personne (bob) n'est pas connectée.
```

De manière plus générale, il faudra remplacer « bob » dans ce message par le pseudonyme du destinataire pour lequel l'envoi a échoué.

De plus, la longueur maximale du contenu d'un message **est désormais limitée à 1024 octets** (p.ex., le contenu dans l'exemple ci-dessus est « Bonjour Bob ! », ceci n'incluant donc pas l'espace après le pseudonyme du destinataire « bob »). Si un message est envoyé au `serveur-chat` alors que son contenu est plus long que cette limite, l'émetteur **doit** être déconnecté par le `serveur-chat`.

3.5 Connexion via un *socket*

Le programme `chat` doit se connecter, via un *socket* TCP, au programme `serveur-chat`. La connexion est établie selon modèle du client-serveur où le programme `chat` joue le rôle du client et `serveur-chat` celui du serveur.

La connexion du `chat` au `serveur-chat` est configurée ainsi :

Adresse IP

- si la variable d'environnement `IP_SERVEUR` est définie et contient une adresse IPv4 valide (c.-à-d., respecte le format « w.x.y.z » avec $w, x, y, z \in \{0, ..., 255\}$), alors l'adresse IP est celle contenue dans cette variable d'environnement,
- sinon, l'adresse IP est 127.0.0.1 par défaut.

Port

- si la variable d'environnement `PORT_SERVEUR` est définie et contient un nombre compris entre 1 et 65 535, alors le port est le nombre contenu dans cette variable d'environnement,
- sinon, le port est 1234 par défaut.

3.6 Threads

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

1. le *thread* d'origine : lancé initialement en exécutant la fonction `main` du programme `chat` ;
2. le second *thread* : créé par le *thread* d'origine.

Le *thread* d'origine doit lire les messages sur l'entrée standard (`stdin`) et les transmettre au destinataire via le *socket*.

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

Cependant, si l'option --manuel est activée, vous pouvez choisir quel *thread* se charge d'afficher les messages en fonction de la situation dans laquelle l'affichage a été demandé.

De plus, lorsque l'option --manuel est activée, les messages destinés à l'utilisateur doivent également être conservés en mémoire jusqu'à leur affichage.

3.7 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` peut varier durant l'exécution du programme. Pour ce signal, les processus le traitent ainsi :

- tant que la connexion (fonction `connect()`) n'a pas été établie, 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 la connexion a été établie et que l'option --manuel est activée, alors la réception du signal `SIGINT` par le *thread* d'origine doit désormais afficher les messages en attente conservés en mémoire.

En fonction de vos choix d'implémentation, vous aurez peut-être besoin de l'Annexe A, le traitement des signaux avec les threads rajoutant quelques difficultés.

3.8 Fin du programme

Le programme doit se terminer dans les différentes circonstances suivantes :

1. en cas de paramètres invalides (voir Section 3.2) ;
2. en cas de fin normale du programme (avec alors un code de retour de 0) :
   - si l'entrée standard (`stdin`) est fermée,
   - si la connexion avec le `serveur-chat` est interrompue.
3. dans certains cas à la réception d'un `SIGINT` (voir Section 3.7) ;
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.

4 Script Bash : `chat-auto`

4.1 Objectif

L'objectif du script Bash `chat-auto` est de faciliter l'utilisation du programme `chat`. En effet, il est pénible de réécrire systématiquement le pseudonyme du destinataire. Ce script vise à ne plus avoir à le réécrire à chaque fois. Pour ce faire :

1. il commence en demandant le destinataire des prochains messages ;
2. il retranscrit ensuite les messages via le `chat` en ajoutant le pseudonyme au début ;
3. il permet de changer le pseudonyme du destinataire lorsque la combinaison Ctrl + D est utilisée (retour au point 1).

4.2 Détails de fonctionnement

Ce script fonctionne en lançant le programme `chat` et en altérant les valeurs écrites sur `stdin` pour y ajouter le pseudonyme du destinataire. Tant qu'un Ctrl + D n'a pas été fait, le script continue d'ajouter le même pseudonyme à chaque message.

Lorsqu'un Ctrl + D est effectué (le `read` de Bash retourne alors 1), le script attend en entrée un pseudonyme. Si, à ce moment-là, aucun pseudonyme n'est spécifié (ligne vide) ou qu'un autre Ctrl + D est réalisé, le script `chat-auto` finit. Sinon, il considère que le texte entré est le

pseudonyme du prochain destinataire (la gestion des espaces n'est pas prise en compte, il faut donc entrer le pseudonyme avec des tirets).

Notez que vous aurez besoin de rediriger le `stdin` du `chat` pour arriver à vos fins. De plus, de cette manière, le Ctrl + D n'affectera pas directement le `stdin` du `chat` car, si son `stdin` est redirigé, ce n'est plus le même que celui du script `chat-auto`.

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.

Vous êtes autorisés à compiler avec des versions plus récentes des langages C et C++ que celles demandées mais indiquez-le dans le rapport.

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
```

ou (ces options étant malheureusement incompatibles, vous empêchant d'analyser à la fois la mémoire et les accès concurrents) :

```
-g -fsanitize=thread,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.

5.1 Pondération

- Script Bash **/1.5**
- Ce projet de programmation système va principalement évaluer votre compétence à manier correctement les outils liés aux systèmes d'exploitation (processus, *threads*, *sockets*, signaux, synchronisation, ...) 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. **/10.5**

— 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++ :
  — de *serveur-chat* dans le dossier *src/serveur/* ;
  — de *chat* dans le dossier *src/chat/*.
— les éventuels scripts Bash que vous utilisez (donc au moins `chat-auto`);
— un seul Makefile pour compiler le *serveur-chat* et le `chat` ;
— votre rapport au format PDF avec le nom des membres du groupe et leur matricule.

Vous devez soumettre votre projet sur l'**Université Virtuelle** pour le **22 décembre 2024 23h59** au plus tard.

Retards

Tout retard sera sanctionné d'un point par tranche de 4h de retard, avec un maximum de 24h 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 Gestion des signaux dans un contexte multithread

A.1 Signaux dans un contexte multithread

Lorsqu'un signal est envoyé à un processus ayant plusieurs *threads*, n'importe lequel de ceux-ci qui n'a pas bloqué le signal est susceptible d'être interrompu pour exécuter le gestionnaire de signaux choisi.

Attention également, dans un contexte multithreading, si vous aviez plusieurs fonctions bloquantes en cours au moment de l'arrivée d'un signal, seule la fonction bloquante du thread réceptionnant le signal sera interrompue. Si vous souhaitez retransmettre le signal aux autres threads, la Section A.3 pourrait vous être utile...

Vous êtes libres de l'utiliser ou d'utiliser une autre alternative équivalente.

A.2 Réception du signal dans les bons threads

Pour rappel, les signaux sont délivrés aléatoirement à un thread qui ne bloque pas ni n'ignore ce signal. Si vous souhaitez que seuls certains threads aient la possibilité de recevoir un signal, vous pouvez bloquer (`SIG_BLOCK`) ces signaux dans les autres threads. Par exemple, pour bloquer les signaux SIGINT et SIGUSR1 dans un thread, vous pouvez utiliser :

```
sigset_t set;

sigemptyset(&set);         // Ensemble vide de signaux
sigaddset(&set, SIGINT);   // Ajouter le signal SIGINT
sigaddset(&set, SIGUSR1); // Ajouter le signal SIGUSR1

if (pthread_sigmask(SIG_BLOCK, &set, NULL) != 0) {
   /* Erreur (errno mis à jour) */
}
```

Vous pouvez les débloquer après en utilisant `SIG_UNBLOCK` à la place de `SIG_BLOCK` dans le code ci-dessus. Les threads créés héritent du blocage des signaux de leur thread "parent" (mais vous pouvez changer ensuite ceux qui sont bloqués de manière indépendante).

A.3 Envoyer un signal à un thread spécifique

En fonction de vos choix d'implémentation, il pourrait s'avérer judicieux de pouvoir envoyer un signal à des threads spécifiques. Il existe justement une fonction pour cela : `pthread_kill()`. Elle fonctionne de manière similaire à `kill()` sauf que vous devez préciser l'identifiant du thread (un `pthread_t`) qui recevra le signal au lieu d'un PID. Sa signature est la suivante :

```
#include <signal.h>
int pthread_kill(pthread_t thread, int sig);
```

Supposons que vous ayez un thread référencé par `th` et que vous souhaitiez lui envoyer le signal `SIGUSR1`, vous feriez :

```
pthread_kill(th, SIGUSR1);
```