1 Cahier des charges

Ce devoir est à faire en binôme de préférence, pas de trinôme.

La date de remise du projet est :
— jeudi 13 décembre 2018 à 18h15

Le projet sera déposé sur la plate-forme UPDAGO, sous forme d'un fichier archive au format *tar* compressé avec l'utilitaire *bzip2*.

Les enseignants se réservent le droit de convoquer les étudiants à un oral pour des précisions sur le travail.

Le nom de l'archive sera IMPÉRATIVEMENT composé de vos noms de famille (ou d'un seul nom en cas de monôme) en minuscules dans l'ordre lexicographique, d'un underscore, du mot "projet", par exemple *alayrangues_subrenat_projet*, suivi des extensions classiques (i.e. *".tar.bz2"*).
Le désarchivage devra créer, dans le répertoire courant, un répertoire s'appelant : *PROJET*.
Ces directives sont à respecter SCRUPULEUSEMENT (à la minuscule/majuscule près). Un script-shell est à votre disposition pour vérifier votre archive.

Le langage utilisé est obligatoirement le C. Un programme doit compiler, sans erreur ni warning, sous le Linux des machines des salles de TP avec les options suivantes :
-Wall -Wextra -pedantic -std=c99 (voire *-Wconversion* si vous avez le courage).
De même un programme doit s'exécuter avec *valgrind* sans erreur ni fuite mémoire.

Vous n'êtes pas autorisés à utiliser des bibliothèques ou des composants qui ne sont pas de votre cru, hormis les bibliothèques système. En cas de doute, demandez l'autorisation.

Il vous est demandé un travail précis. Il est inutile de faire plus que ce qui est demandé. Dans le meilleur des cas le surplus sera ignoré, et dans le pire des cas il sera sanctionné.

2 Présentation générale du projet

Le but est d'implémenter un *démon*, des *services* et des *clients*.

Le *démon* est un programme qui tourne en permanence. Il attend que des *clients* le contactent pour leur rendre des *services*.
Les *clients* sont des programmes indépendants.
Les *services* sont des programmes indépendants, mais lancés par le *démon*.

Donc il y a un exécutable distinct pour chaque *service* ; et il y a autant d'exécutables *clients* qu'il y a d'exécutables *services*.

Un *client* s'adresse exclusivement, en premier lieu, au *démon* qui se décharge alors sur un des *services* de son choix. À partir de ce moment le *client* échangera directement avec le *service*, indépendamment du *démon*.

Il y a une série préétablie de services pouvant être rendus, i.e. ils sont codés en dur dans des fichiers sources distincts. On rappelle que chaque *service* est un exécutable indépendant.
Un fichier de configuration, utilisé uniquement par le *démon*, liste les services disponibles.

Tous ces processus utilisent, pour communiquer, les IPC et les tubes nommés fournis par les bibliothèques du C.

3 Fonctionnement détaillé

3.1 IPC et autres communications

Un *client* et le *démon* communiquent :
- par une paire de tubes nommés (*mkfifo*, *open*, ...) pour amorcer la discussion,
- des sémaphores IPC (*semget*, *semop*, ...) pour se synchroniser.

Une fois la demande du *client* validée, le *démon* :
- crée une paire de tubes nommés pour la communication entre le *client* et le *service*,
- lance le *service* dans un processus fils (*fork*, *exec*), pour que le *service* soit autonome et que le *démon* puisse traiter immédiatement un autre *client*.

Le *client* et le *service* sont alors complètement indépendants du *démon* et communiquent via cette nouvelle paires de tubes qui disparaîtront avec la fin du traitement.

Il y a des synchronisations et des sections critiques (accès restreints à des portions de code) qui utiliseront obligatoirement les sémaphores (*semget, ...)* pour une gestion entre processus lourds (cf. détails ci-dessous).

Les entrées-sorties seront effectuées avec les fonctions de bas niveau (*open, write, ...*).

3.2 *client*

Il y a deux tubes nommés, pré-créés par le *démon*, pour obtenir une communication bidirectionnelle.

Dans un premier temps :
- *client* envoie une demande au *démon* (sur le premier tube), cette demande étant le numéro du service souhaité (un numéro particulier indiquera au *démon* de s'arrêter),
- *client* reçoit en retour (sur le deuxième tube) un code indiquant si le *démon* accepte ou non de répondre à la demande,
- si la demande est acceptée, *client* reçoit (sur le deuxième tube) les noms des deux tubes nommés par lesquels vont désormais transiter toutes les autres communications (i.e. entre le *client* et le *service*).
- dans tous les cas le *client* prévient le *démon* de la fin de la transaction (pour que le *démon* puisse gérer un autre client immédiatement) ; le *démon* est donc en attente, et sera débloqué grâce à un sémaphore.

Attention, cette phase est délicate et il ne faudrait pas que deux *clients* se télescopent. Le plus simple est que la portion de code gérant cette première communication soit exécutée par un seul *client* à la fois (en mettant le code en section critique). Cette exclusivité est sous la responsabilité des *clients*, mais le mutex a été préalablement créé par le *démon.*.

Les créations des tubes nommés sont à la charge du *démon*. Leurs destructions sont à la charge du *service*.
Une paire de tubes nommés, entre un *client* et un *service*, n'existe que le temps de satisfaire la demande.

Dans un second temps :
- *client* envoie les données à traiter au *service* créé à cette occasion par le *démon* (par exemple deux nombres, ou une image, ...),
- *client* reçoit le résultat,
- dans tous les cas le *client* envoie un code de fin de communication.

À part le dernier point qui est commun à tous les *clients*, les deux autres ont un protocole complètement libre qui dépend du service demandé (on pourrait imaginer plusieurs aller-retours entre le *client* et le *service*, même si ce n'est pas nécessaire dans ce projet).

Dans le cas particulier du *client* qui indique au *démon* de se terminer, ce "second temps" n'existe pas.

Dans le code fourni (répertoire *CLIENTS*), une proposition d'organisation de l'implémentation est proposée mais en aucun cas obligatoire.

3.3 *service*

Un *service* est issu d'un *fork/exec* du *démon* et une fois lancé est complètement indépendant de ce dernier. Le seul rôle du *service* est de répondre à un *client* et de se terminer.

Un *service* effectue les opérations suivantes :
- récupérer (sur le tube 1) les données à traiter,
- calculer le résultat,
- envoyer (sur le tube 2) le résultat.
- attendre le code fin provenant du *client*.
- détruire les deux tubes.

Comme indiqué dans la section précédente, seuls les deux derniers points sont communs à tous les *services* ; les trois premiers points peuvent être aussi compliqués que nécessaire.

Dans le code fourni (répertoire *SERVICES*), une proposition d'organisation de l'implémentation est proposée mais en aucun cas obligatoire.

3.4 Services à implémenter

On impose un minimum de 4 services.
Dans le code fourni (fichier *SERVICES/0README*), il y a des explications détaillées,

3.4.1 Service 1 : somme de deux *float*

Les données à traiter sont deux *float*, le résultat est un *float*.

3.4.2 Service 2 : miroir

La donnée à traiter est une chaîne de caractères, le résultat est une chaîne de caractères qui est le miroir de la chaîne d'entrée.

3.4.3 Service 3 : compression

La donnée à traiter est une chaîne de caractères, le résultat est une chaîne de caractères qui est la chaîne d'entrée privée de ses voyelles.

3.4.4 Service 4 : binarisation

Le but est de transformer une image en niveaux de gris en une image en noir et blanc. Il y a un seuil ; tous les niveaux en dessous du seuil deviennent noirs, ceux en dessus du seuil deviennent blanc, et ceux égaux au seuil deviennent gris "moyen".

Techniquement (et surtout pour se simplifier la vie) l'image sera stockée dans un tableau unidimensionnel (les lignes de pixels sont mises les unes derrière les autres).

En outre on impose une résolution multi-threads.

Les données à traiter sont l'image, le seuil et le nombre de threads (choisi par le *client* donc). Le résultat est l'image modifiée.

Cf. fichier fourni pour plus de détails.

3.5 *démon*

3.5.1 Fichier de configuration

Le fichier de configuration a une structure fixe (et même rigide) :
- les lignes vides sont autorisées et ignorées,

- les lignes commentaires sont autorisées et ignorées (un commentaire est une ligne commençant par `#`),
- *ligne 1 :* nombre de *services* disponibles,
- puis une ligne par *service* :
  - numéro du *service* (numérotation commençant à 1)
  - mot indiquant si le service est ouvert ou fermé
  - nom de l'exécutable du *service*

Ce fichier n'est lu et connu que par le *démon*. Notamment un *service* ne sait pas s'il est ouvert ou fermé. S'il est fermé il ne sera simplement jamais sollicité par le *démon*.

Dans le code fourni (répertoire *CONFIG*) il y a :
- un exemple de fichier de configuration correct
- plusieurs fichiers incorrects
- un programme permettant de tester les fichiers de configuration
- le fichier *0README* qui montre les sorties écran du programme de test

Pour lire et exploiter un fichier de configuration il y a une API imposée. Les deux fichiers sont dans le répertoire principal :
- *config.h* : signature des fonctions de l'API, vous n'avez pas le droit de modifier ce fichier.
- *config.c* : implémentation de l'API qui est à votre charge.

Le programme de test utilise cette API.

Seul le *démon* (et le programme de test) utilisera cette API.

3.5.2 processus *démon*

Le principe général est le suivant sur une boucle "infinie" :
- attendre la connexion d'un *client*
- analyser la demande du *client*
- si elle est correcte, lancer le *service* et le laisser gérer le *client*

De manière plus précise ;
- initialisations diverses (dont lecture du fichier de configuration)
- répéter (jusqu'à ordre de fin)
  - se connecter sur les tubes de communication
  - attendre l'envoi d'un numéro de *service* par un *client*
  - si le numéro est correct
    - créer deux tubes dédiés pour ce *client*
    - indiquer au *client* les noms des tubes
    - lancer le *service* dans un processus fils (*fork*/*exec*)
  - se déconnecter des tubes principaux
  - attendre l'ordre du *client* pour continuer
- libération des ressources

Dans le code fourni (répertoire principal), une proposition d'organisation de l'implémentation est proposée mais en aucun cas obligatoire.

4 Travail à rendre

Documents à rendre :
- le code du projet (chaque fichier créé doit comporter vos noms et prénoms),
- Notez qu'il a des scripts-shell pour compiler les programmes et non des *Makefile* ; vous pouvez laisser comme cela.
- un rapport au format pdf, nommé "rapport.pdf" (disons 2 pages hors titre et table des matières) qui contient :
  - l'organisation de votre code : liste des fichiers avec leurs buts,
  - et surtout tous les protocoles de communication.

  N'hésitez pas à préciser ce qui ne marche pas correctement dans votre code.
  Soignez l'orthographe, la grammaire, ...

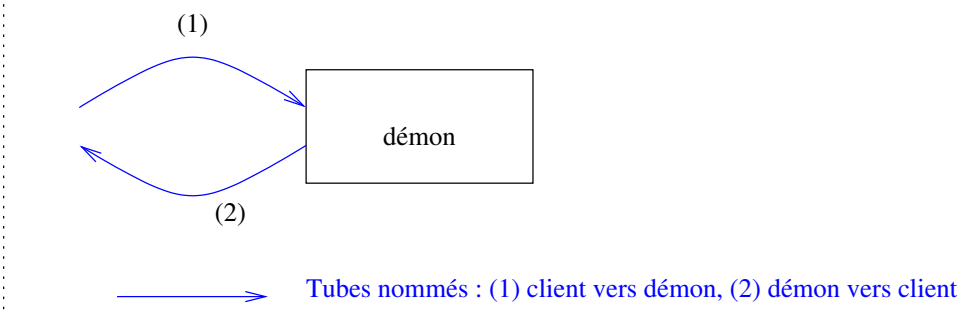
Le fichier “rapport.pdf” doit être directement dans le dossier *PROJET* (racine de votre archive).
Le code du projet sera dans un sous-répertoire nommé *src*.
La hiérarchie des répertoires fournie pour le code doit être conservée (dans *src* donc).
Rappelez-vous que vous avez à disposition un script-shell de vérification, et que les archives ne passant pas avec succès cette vérification seront refusés.

Attention, dans l'archive, ne mettez que les fichiers sources. Tous les autres fichiers (*.o* et autres cochonneries) NE doivent PAS être dans l'archive.

5 Schémas explicatifs

5.1 Le *démon* au repos

Voici l'état des programmes lorsqu'aucun *client* n'est en action, donc seul le *démon* tourne.



5.2 Ensemble des processus avec *clients* en action

Voici l'état des programmes lorsque deux *clients* sont en action et un troisième établit la connexion avec le *démon*.