1 Cahier des charges

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

La date de remise du projet est :
— mercredi 16 décembre 2020 à 18h00

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

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 *skapin_subrenat_projet*, suivi des extensions classiques (i.e. *".tar.gz"*).
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 *master*, des *workers* et des *clients*.

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

Il n'y a qu'un seul code source, et donc un seul exécutable, pour les *clients*, mais on peut en lancer plusieurs en même temps ; ils se font donc concurrence.

De même, il n'y a qu'un seul code source pour les *workers*, mais ils sont lancés en plusieurs exemplaires, et ils formeront un pipeline.

Un *client* s'adresse exclusivement au *master* qui se décharge alors sur le pipeline de *workers* pour les calculs, avant de renvoyer la réponse au *client*.

Le but est de déterminer si un nombre est premier en se basant sur le crible d'Ératosthène, et plus précisément sur la version pipeline : le crible de Hoare [1].

Chaque *worker* gère un étage du pipeline et est connecté au *worker* précédent, au *worker* suivant ainsi qu'au *master*.

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

1. https://fr.wikipedia.org/wiki/Crible_d%27%C3%89ratosth%C3%A8ne

3 Fonctionnement détaillé

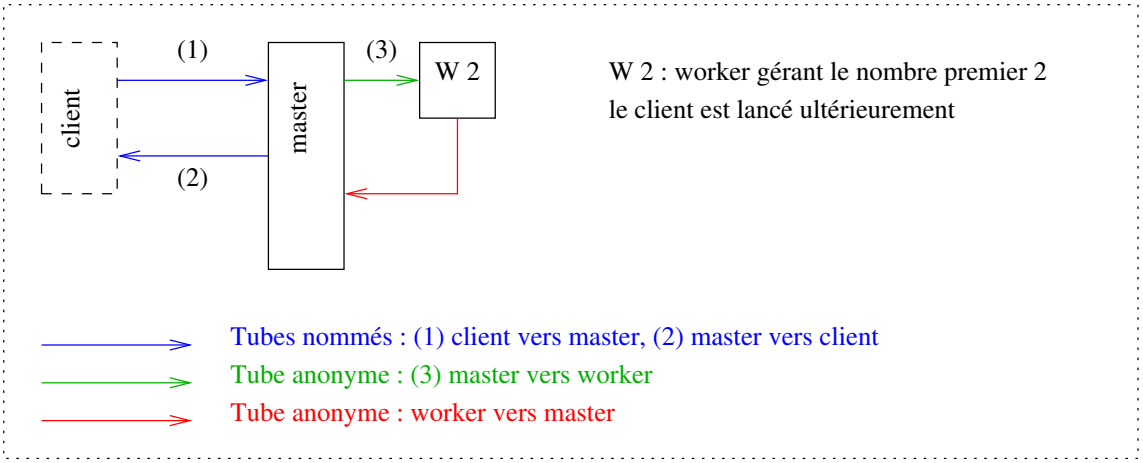
3.1 IPC et autres communications

Le *master* est exécuté en premier et lance, dès sa création, le premier étage du pipeline : un *worker* via un *fork/exec*.

Le *master* et le premier *worker* (associé au premier nombre premier, à savoir 2) communiquent par deux tubes anonymes pour un échange bidirectionnel.

Un *client* et le *master* communiquent :
- par une paire de tubes nommés (*mkfifo*, *open*, ...) pour échanger données et résultats,
- deux sémaphores IPC (*semget*, *semop*, ...) pour se synchroniser.

Voici le schéma au début de la vie du *master* :



Le *client* envoie une demande de calcul au *master* sur un tube nommé et attend la réponse sur l'autre tube nommé.

Il y a des synchronisations et des sections critiques (accès restreints à des portions de code) qui utiliseront obligatoirement les sémaphores IPC (*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 *master*, pour obtenir une communication bidirectionnelle entre *clients* et *master*.

Dans un premier temps :
- Le *client* envoie une demande au *master* (sur le premier tube). Il y a 4 ordres possibles :
  1. arrêt du *master*
  2. demander si un nombre est premier
  3. demander combien de nombres premiers ont été calculés
  4. demander le plus grand nombre premier calculé
- Dans le cas du deuxième ordre, le *client* envoie le nombre dont il faut tester la primalité.
- Le *client* attend et reçoit la réponse sur le deuxième tube ; dans le cas du premier ordre, il reçoit un accusé de réception du *master*.
- Le *client* débloque le *master* grâce à un sémaphore ; en effet, le *master*, avant de s'occuper d'une nouvelle demande, doit s'assurer que le *client* a complètement terminé.

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 communication (i.e. tous les points sauf le dernier) 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 *master*.

De même, le sémaphore de synchronisation en fin de *client* a été créé et initialisé par le *master*.

Dans le code fourni (fichier *client.c*), une proposition d'organisation de l'implémentation est proposée mais non imposée.

3.3 *client* (bis)

Cette partie est complètement indépendante du reste du projet.

On rajoute une fonctionnalité au *client* : faire un test de primalité complètement local (i.e. sans le *master*) en mode multi-thread.

Voici l'algorithme demandé :
- soit N le nombre à tester
- on crée un tableau de booléens dont les cases sont numérotées de 2 à *N-1* (ou mieux de 2 à $\sqrt{N}$)
- on lance autant de threads qu'il y a de cases, chaque thread étant responsable d'une et une seule case
- soit T_i le thread s'occupant de la case i : il regarde si N est divisible par i ; si c'est le cas il met la i^{me} case à *false* (N n'est pas premier), sinon il la met à *true* (N est peut-être premier)
- lorsqu'on tous les threads ont fini, on regarde si toutes les cases sont à *true*

3.4 *worker*

Bien que décrit ici, il est préférable d'implémenter le code du *worker* une fois les codes du *client* et du *master* fonctionnels.

Le premier *worker* est issu d'un *fork/exec* du *master* dès le lancement de ce dernier. Ensuite les *workers* se créent en cascade : le dernier *worker* créé est celui qui créera le suivant.

Un *worker* est responsable d'un seul nombre premier (nommé P) ; autrement dit il y aura autant de *workers* s'exécutant que de nombres premiers calculés.

En résumé le rôle du *worker* est le suivant :
- il reçoit un nombre (nommé N) via un tube anonyme
- si N est divisible par P alors le *worker* indique au *master* que N n'est pas premier
- sinon si N est égal à P alors le *worker* indique au *master* que N est premier
- sinon le *worker* transmet N au *worker* suivant dans le pipeline.
- il recommence au premier point

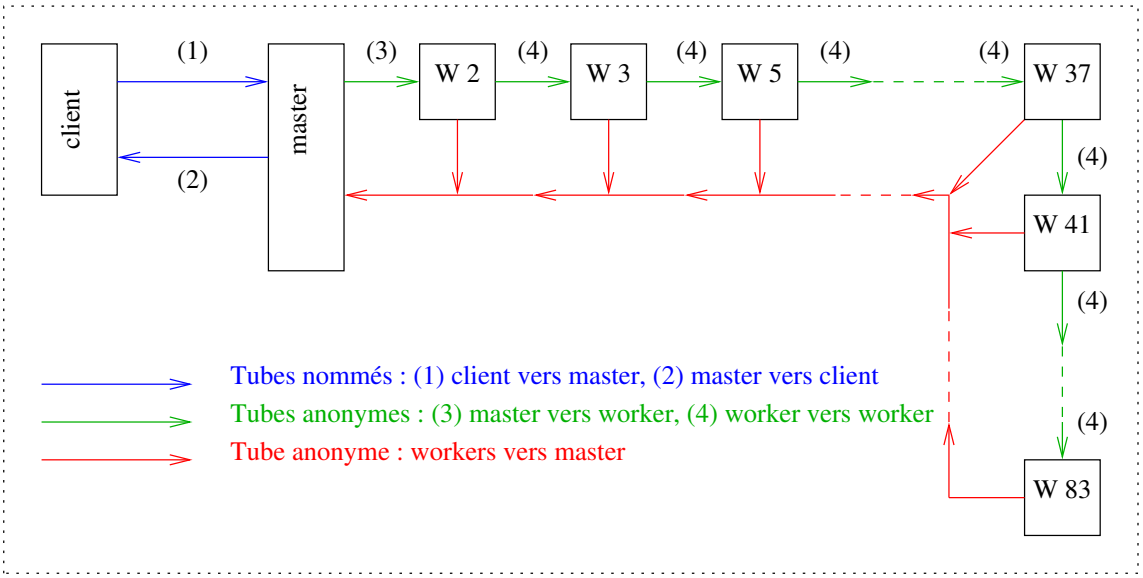
Plus précisément, un *worker* effectue les opérations suivantes :
- c'est un processus indépendant lancé par un *fork/exec*, soit du *master* pour le premier *worker*, soit par le *worker* précédent dans le pipeline.
- il reçoit en ligne de commande (i.e. *argv*) le nombre premier (P) dont il a la charge, ainsi que les *file descriptors* pour communiquer avec le précédent et le *master*.
  Note : lorsqu'un *worker* vient d'être créé il est par définition le dernier du pipeline.
- répéter à l'infini
  - attendre un ordre du précédent worker [2]
  - si c'est un ordre d'arrêt :
    - transmettre cet ordre au worker suivant (s'il existe)
    - attendre la fin de ce dernier (s'il existe)
    - sortir de la boucle
  - sinon c'est un nombre N à tester :
    - si N égale P alors envoyer un succès au *master* (via le tube anonyme)
    - sinon si P divise N alors envoyer un échec au *master*
    - sinon s'il y un *worker* suivant alors lui transmettre N
    - sinon créer un nouveau *worker* qui aura la charge de N (qui est donc premier, mais c'est le nouveau *worker* qui le signalera au *master*).
- se terminer proprement

Voici le principe de création d'un nouveau *worker* :

2. ou du *master* dans le cas du premier *worker* : nous ne préciserons désormais plus ce cas particulier, d'autant plus qu'un worker n'a aucune idée de qui lui envoie des informations.

— création d'un tube anonyme
— *fork/exec* en passant, en ligne de commande, au nouveau *worker* : son nombre (i.e. N), le file descriptor pour recevoir des données du *worker* courant, le file descriptor du tube vers le *master*.
— on en déduit qu'il n'y qu'un seul tube anonyme vers le *master* et que tous les *workers* écrivent dedans ; mais comme dans le pipeline, pour un calcul donné, un seul *worker* a la possibilité d'écrire, ce n'est pas un problème.

Le schéma complet et général de communication est donc le suivant :



Et donc les étages se rajoutent au fur et à mesure.

Dans le code fourni (fichier *worker.c*), une proposition d'organisation de l'implémentation est proposée mais non imposée.

3.5 *master*

Le principe général est le suivant sur une boucle "infinie" :
— attendre la connexion d'un *client*
— analyser la demande du *client*
— éventuellement faire intervenir les *workers*
— renvoyer le résultat au *client*

De manière plus précise ;
— initialisations diverses dont création des tubes nommés et des deux sémaphores
— lancement du premier *worker* avec les deux tubes anonymes
— boucle infinie de travail :
  — ouvrir les tubes de communication avec le futur *client*
  — attendre l'envoi d'un ordre d'un *client*
  — si ordre de fin
    — envoi ordre de fin au premier *worker*
    — attendre la fin du premier *worker*
    — envoyer accusé de réception au *client*
    — sortir de la boucle
  — si test de primalité
    — envoi du nombre au premier *worker* (cf. ci-dessous)
    — attendre la réponse d'un des *workers*
    — transmettre le résultat au *client*
  — si demande du nombre de nombres premiers trouvés, ou si demande du plus grand nombre premier trouvé, envoyer le résultat au *client*
  — fermer les tubes nommés de communication
  — attendre le déblocage du *client*
— fin répéter

— libération des ressources
— destruction des tubes

Attention dans le crible d'Hoare, les nombres doivent être envoyés un par un dans le pipeline.

Plaçons nous dans le cas suivant :
— le dernier nombre envoyé dans le pipeline est 42
— le *client* demande le nombre 65

Avant d'envoyer 65 dans dans le pipeline, le *master* doit d'abord envoyer tous les nombres de 43 à 64.

Dans le code fourni (fichier *master.c*), une proposition d'organisation de l'implémentation est proposée mais non imposée.

4 Travail à rendre

Documents à rendre :
— le code du projet (chaque fichier créé doit comporter en commentaire vos noms et prénoms),
— Notez qu'il y a *Makefile* pour compiler le projet. Si vous avez ajouté des fichiers, il faudra le compléter.
— 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 refusées.

Tous les retours des appels système doivent être testés. Une assertion fera amplement l'affaire.

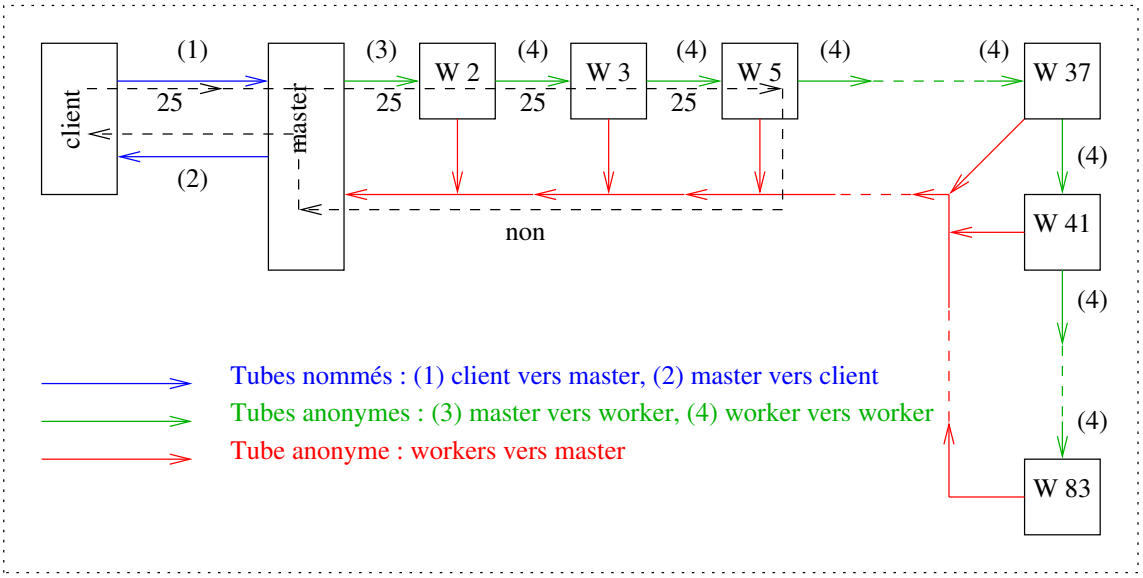
Attention, dans l'archive, ne mettez que les fichiers sources. Tous les autres fichiers (*.o* et autres cochonneries) NE doivent PAS être dans l'archive.

Tournez SVP ...

5 Schémas explicatifs

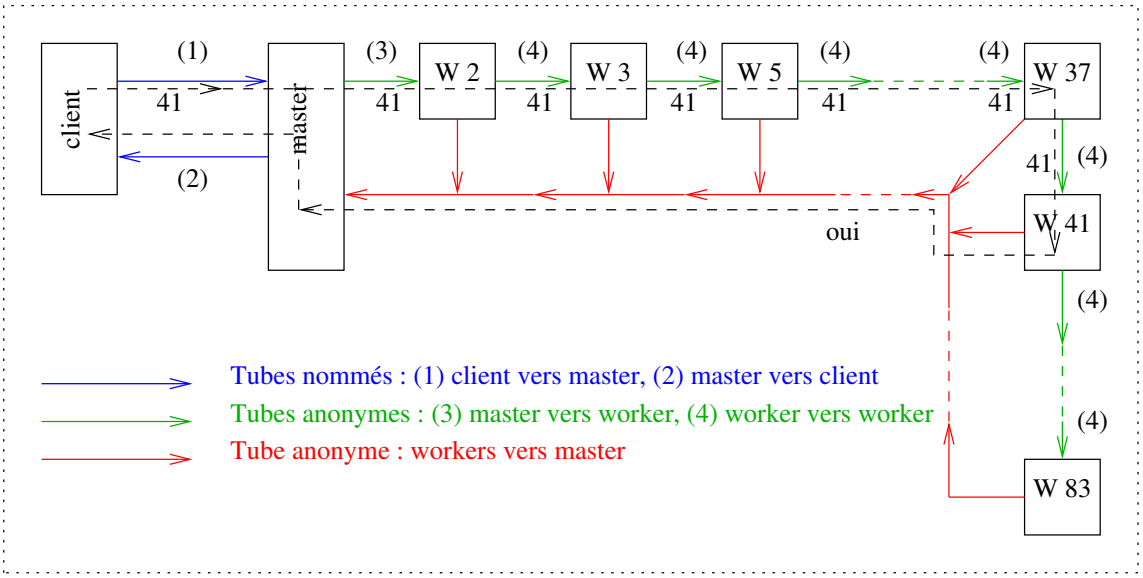
5.1 Cheminement du nombre 25

Ce nombre transite de *worker* en *worker*, jusqu'au *worker* 5 qui détecte que ce n'est pas un nombre premier ; un message négatif est envoyé au *master* :



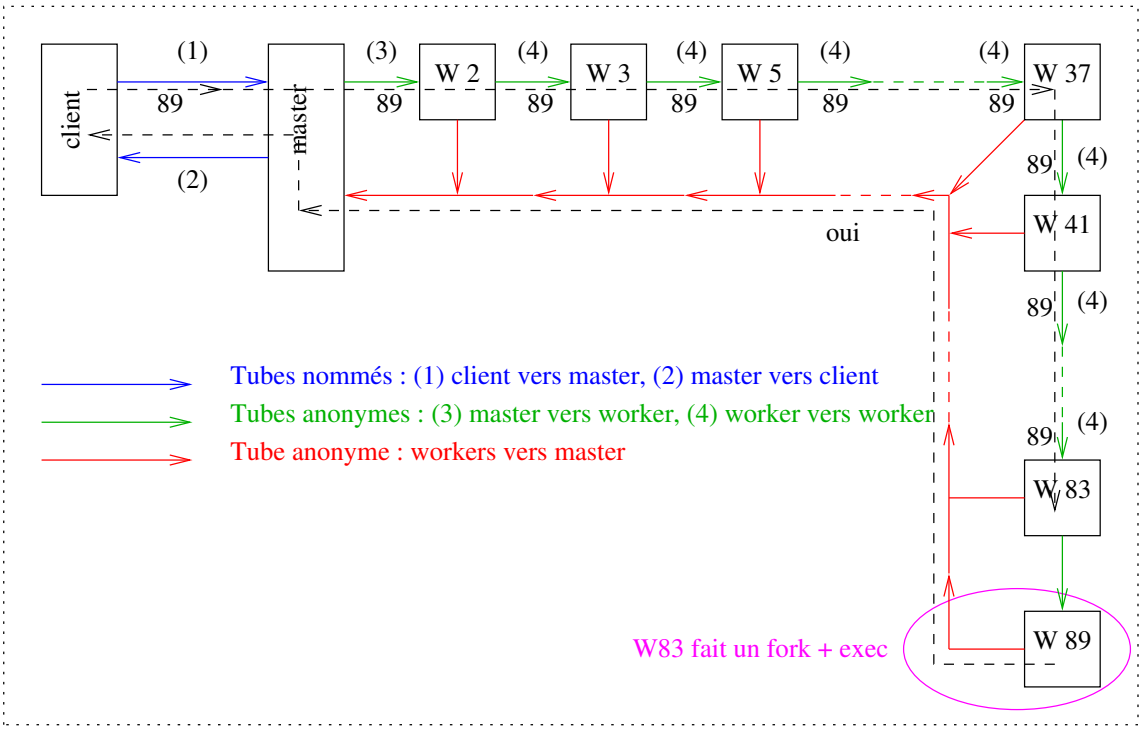
5.2 Cheminement du nombre 41

Ce nombre transite de *worker* en *worker*, jusqu'au *worker* 41 qui détecte que c'est un nombre premier ; un message positif est envoyé au *master* :



5.3 Cheminement du nombre 89

Ce nombre transite de *worker* en *worker*, jusqu'au *worker* 83 qui ne détecte aucun problème. Ce *worker* veut alors transmettre le 89 au *worker* suivant qui n'existe pas. Le *worker* 83 crée alors le *worker* 89 en le rajoutant au pipeline. Le *worker* 89, à son lancement, envoie automatiquement un message positif au *master*.


client
master
W 2
W 3
W 5
W 37
W 41
W 83
W 89
(1)
(2)
(3)
(4)
89
oui
Tubes nommés : (1) client vers master, (2) master vers client
Tubes anonymes : (3) master vers worker, (4) worker vers worker
Tube anonyme : workers vers master
W83 fait un fork + exec