Projet de programmation Système
Rapport d'architecture

Dupont Robin et Stiers Nathan

Nous allons utiliser une structure nommée Programme

```
typedef struct{
        int id;
        char nomFichier[256];
        int erreurCompil;
        int nbrExec;
        long dureeExecTotal;
} Programme;
```

qui contient les informations relatives à un programme donné. Ces structures se trouveront dans une liste allant de 0 à 999 (Programme listeProg[1000];) comme indiqué dans l'énoncé. Cette même liste se trouvera dans la mémoire partagée afin d'être accessible par les différentes applications.

Le sémaphore ne bloquera l'accès qu'en cas de modification d'une donnée dans la structure. C'est-à-dire après l'exécution d'un fichier source afin d'y écrire les données calculées.

Les messages sur le réseau se feront par une structure type Message (avec un entier afin d'en alléger le travail et un buffer de 255 caractères). Cette seule structure permettra de lire le code même s'il fait plus de 255 caractères à l'aide de l'appel système shutdown.

```
typedef struct {
        char MessageText[256];
        int code;
        int numeroDeProgramme ;
        varchar sender ;
} Message;
```

Au niveau des signaux, il ne faut pas gérer les arrêts brutaux. Il ne faudra donc gérer que l'utilisation du ctrl+C lors de la fermeture du serveur. L'argument delay du fils minuterie ne sera pas un signal d'alarme mais sera géré par un simple sleep. On aura donc de base seulement un handler de signaux qui devra se charger d'attendre que les clients aient fini leurs tâches avant de fermer le programme.

Nous utiliserons les select dans les classes serveur afin de ne pas faire attendre inutilement les clients si l'un d'entre eux décidait de ne plus répondre.

Au niveau de la séparation du code en modules, notre logiciel sera composé :

- Du côté serveur avec comme fichiers
    - gstat.c
    - maint.c
    - serveur.c
    - serveurUtils.c
    - serveurUtils.h

- Le côté client aura comme fichiers
    - client.c
    - clientUtils.c
    - clientUtils.h

- Ensuite, le côté utilitaire
    - utils.c
    - utils.h
    - Makefile

- Et enfin, le dossier Applications qui contiendra les .c

Comme on peut le voir, le serveur tout comme le client possèdent un fichier utils qui leur est propre. Ces fichiers serviront à faire les tâches répétitives telles que la connexion au serveur, accepter un client, les constantes de port, etc.

Le fichier utils se trouvant dans le « package » utilitaire aura comme méthode les vérifications d'erreurs, les fork_and_exec, etc. qui seront donc applicables à tous les exécutables. Voir annexe pour une première vue du Makefile et donc de l'architecture

Au niveau du travail en équipe, nous utiliserons un repository GitHub et travaillerons à l'aide de GitKraken. Robin travaillant sur Ubuntu et Nathan sur Windows (à l'aide de l'application permettant d'utiliser les commandes Linux sur Windows), il faudra vérifier de temps en temps que des problèmes de compatibilité ne surviennent pas.

ANNEXE MAKEFILE

```
CFLAGS = -std=c11 -pedantic -Wall -Werror -Wvla  -D_DEFAULT_SOURCE

EXEC = -o $@ $^
OBJECT = -c $<

all : gstat maint serveur client
```

```
#### SERVEUR ####

gstat : gstat.o utils.o serveurUtils.o
	cc $(CFLAGS) $(EXEC)

gstat.o : gstat.c utils.h serveurUtils.h
	cc $(CFLAGS) $(OBJECT)

maint : maint.o utils.o serveurUtils.o
	cc $(CFLAGS) $(EXEC)

maint.o : maint.c utils.h serveurUtils.h
	cc $(CFLAGS) $(OBJECT)

serveur : serveur.o utils.o serveurUtils.o
	cc $(CFLAGS) $(EXEC)

serveur.o : serveur.c utils.h serveurUtils.h
	cc $(CFLAGS) $(OBJECT)

#### CLIENT ####

client : client.o utils.o clientUtils.o
	cc $(CFLAGS) $(EXEC)

client.o : client.c utils.h clientUtils.h
	cc $(CFLAGS) $(OBJECT)

#### UTILS ####

utils.o : utils.c utils.h
	cc $(CFLAGS) $(OBJECT)

serveurUtils.o : serveurUtils.c serveurUtils.h
	cc $(CFLAGS) $(OBJECT)

clientUtils.o : clientUtils.c clientUtils.h
	cc $(CFLAGS) $(OBJECT)

.PHONY : clean

clean :
	@rm -f *.o
	rm -f gstat
	rm -f maint
	rm -f serveur
	rm -f client
```