Rapport de PSTL

Trac-Simulateur & IDE génériques pour OMicrob

Author :
Qiwei XIAN
Ruiwen WANG

Professeur :
Prof.Emmanuel Chailloux

31 mai 2020

Table des matières

1 Introduction

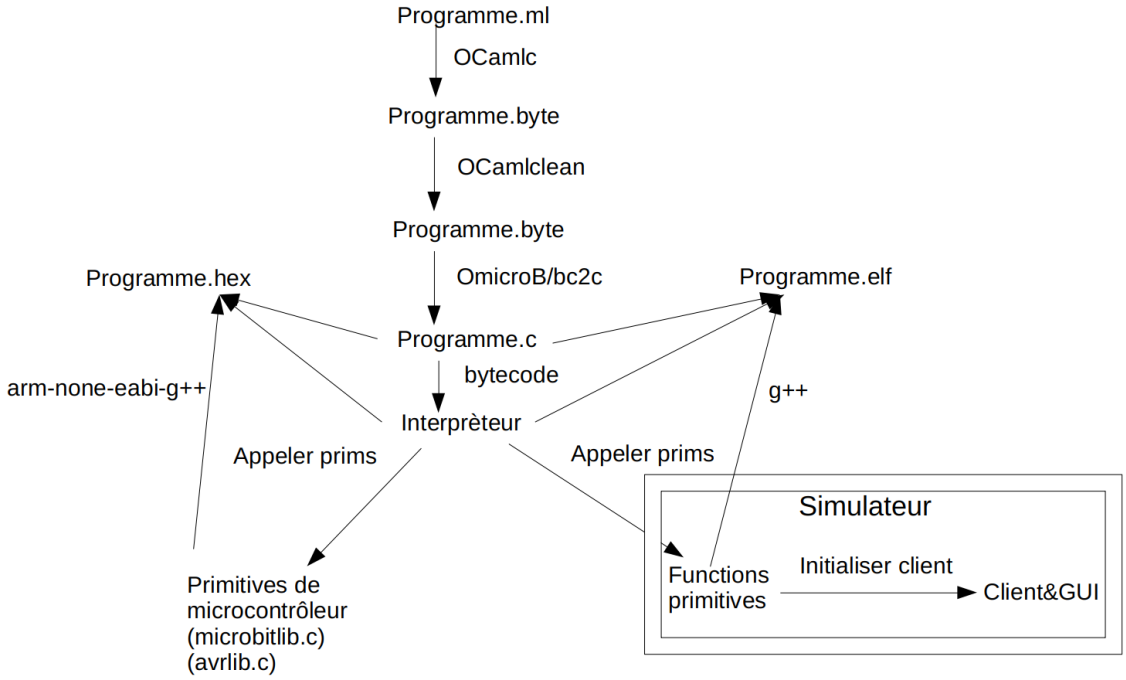
lien vers les codes sources : https://github.com/XIANQw/OMicroB/tree/microbit

La programmation des architectures à base de microcontrôleurs est difficile tant par les ressources limitées accessibles que par les modèles de programmation proposés. L'intégration électronique poussée d'un micro-contrôleur permet de diminuer la taille, la consommation électrique et le coût de ces circuits. La taille des programmes et la quantité de mémoire vive sont "faibles", le tas peut être de quelques kilo-octets seulement. Ils doivent communiquer directement avec les dispositifs d'entrées/sorties (capteurs, effecteurs, ...) via les pattes du circuit principal, et ne possèdent pas les périphériques classiques (souris, clavier, écran). La mise au point d'un programme devient plus difficile de par ce manque d'interaction classique.

On s'intéresse ici à une nouvelle version portable de la machinerie OCaml, appelée OMicroB, qui engendre un programme C contenant la version byte-code du programme OCaml ainsi que l'interprète de ce byte-code OCaml. Omicrob vient avec un environnement de développement incluant un simulateur permettant de décrire un montage et d'exécuter le programme sur l'ordinateur hôte avant de le transférer sur le microcontrôleur. Bien que portable, la version initiale de l'environnement de développement dont le simulateur a été principalement testée pour l'architecture Arduino. Le portage d'OMicrob vers d'autres architectures (Micro :bit Arm Corex-M0, PIC32) nécessite maintenant d'adapter son environnement de développement, principalement le simulateur. L'idée est d'ajouter une couche d'abstraction aux circuits utilisés pour pouvoir facilement passer d'un microcontrôleur à un autre. Par ailleurs une extension synchrone à flots de données, appelée OCaLustre, est particulièrement appropriée pour décrire les interactions externes et la concurrence interne à l'application. Le couple OCaLustre+OMicroB permet une programmation mixte (synchrone et multi-paradigme classique) avec une consommation parcimonieuse des ressources. L'intérêt de ces couches d'abstractions est de faciliter le développement d'applications fiables sur micro-contrôleurs

Ce projet cherche à améliorer cette mise au point de programmes en utilisant d'une part un langage de haut niveau, ici OCaml et son extension synchrone OCaLustre, et d'autre part en fournissant un simulateur et un IDE simple pour la mise au point de tels programmes.

2 Structure de compilation



Cet image montre chaque étape de compilation d'un programme OCaml par OMicroB.

1. **ocamlc** compile le fichier **.ml** avec la bibliothèque et génére un fichier **.byte**.
2. **ocamlclean** traite le fichier généré **.byte**.
3. **bc2c** est le compilateur de **OMicroB**, il permet de transférer le code binaire à un programme **.c**.
4. **g++** compile le **programme.c** et la bibliothèque de simulateur **sf-regs** puis génère le fichier exécutable **.elf**.
5. **arm-none-eabi-g++** compile le **programme.c** avec la bibliothèque de microcontrôleur (avrlib, microbitlib, etc) puis génére le fichier exécutable **.hex**.

OMicroB produit deux fichiers exécutables après de compiler un programme **OCaml**.
- **.elf** est exécutable en mode simulation, il permet de démarrer le simulateur et montrer le changement des états de pin et les effets de programme sur une interface graphique.
- **.hex** est exécutable sur un microcontrôleur.

3 Composition du simulateur

Le simulatuer est composé du coté de client et du coté de serveur, ainsi que un langage de description pour le montage des périphériques sur un microcontrôleur.

3.1 Montage

Pour démarrer le processus client et implémenter l'interface graphique, le client a besoin de savoir quels composants du microcontrôleur il doit visualiser sur l'interface graphique. Donc nous devons proposer un fichier **circuit.txt** qui décrit les composants du microcontrôleur et les associations entre chaque composant et les pins correspondants. Avant de démarrer le processus client, le processus analyse est exécuté pour évaluer ce fichier. Une structure qui contient ces informations est stockée dans un mémoire partagé après l'évaluation. Le client peut recevoir les informations des composants du microcontrôleur depuis ce mémoire partagé.
Cette fonctionnalité est réalisée dans le dossier **src/byterun/montage**, la structure est définie dans **src/byterun/simul/share.h**.

3.2 Client

La partie client est programmée par C dans **src/byterun/client**, se compose de deux fichiers.

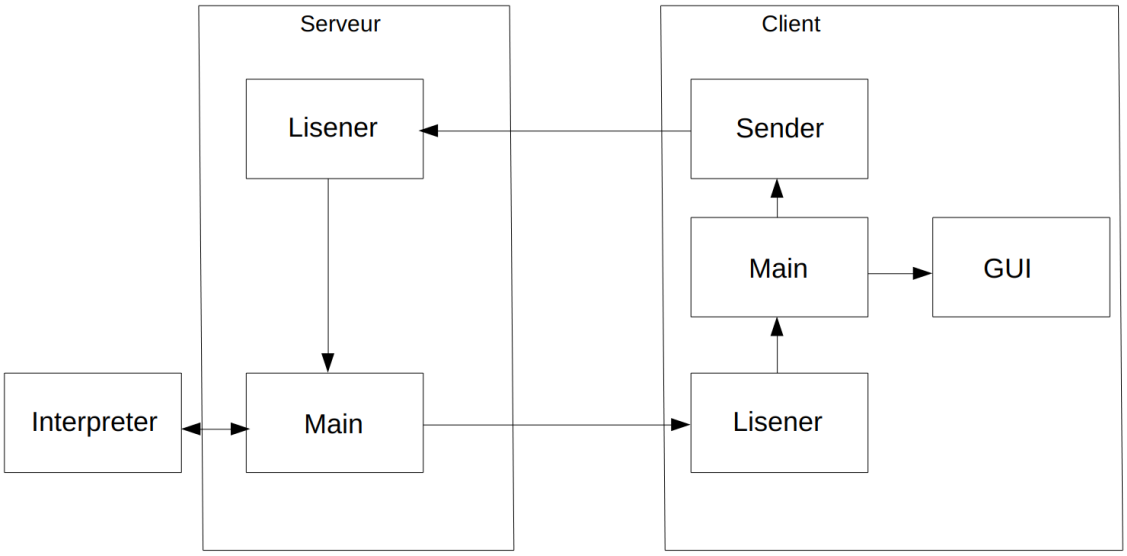
1. **client.c** définit les fonctions du travail par exemple traiter le message qui vient du côté server.
2. **gui.c** implémente l'interface graphique par **gtk3.0**. il nous montre un UI de simulateur permet de visualiser le résultat de programme et les états des pins pendant la simulation, ainsi que l'interaction par les boutons.

3.3 Serveur

La partie serveur est codée par C dans "/src/byterun/", il contient des fichiers :

1. **/src/byterun/vm**. Ce dossier contient les fichiers de l'interprèteur d'OMicroB, par exemple le format des type basiques de données, le gc de machine virtuelle ainsi que les définition des instructions de code binaire. Il exécute le programme OCaml et envoie les instructions des codes binaires au serveur.
2. **/src/byterun/simul/sf-regs.c/.h**. C'est la bibliothèque des primitives de simulateur, lorsque le programme utilise un primitive par exemple **set_pin**, l'interprèteur va appeler le primitive correspondant dans la bibliothèque, le serveur envoie l'instruction au côté client après le calcul.
3. **src/byterun/shared.c/.h**. Ce fichier définit des structures de données partagées entre le processus client et serveur, elles sont stockées dans un mémoire partagé au fur et mesure de l'exécution de programme.

4 Architecture du simulateur



Cet image décrit globalement la structure du simulateur. On l'a réalisé par **l'architecture serveur-client** et il exécute deux processus en concurrence. Chaque processus exécute plusieurs threads et travaille asynchroniquement.

Processus Serveur exécute deux threads, **lisener** et **Main**.

1. **Main** traite simplement l'instruction qui vient de l'interprèteur et envoyer au thread **lisener** de client.
2. **lisener** reçoit l'instruction venant de **Sender** et informer **Main**.

Processus Client tourne simultanément quatre threads, **Main**, **GUI**, **Sender** et **Lisener**.

1. **Main** est le premier thread principal, il traite les arguments venant de processus serveur et connecte aux mémoires partagés, par exemple le mémoire partagé pour la communication et pour le montage. Il est responsable d'exécuter les autres threads.
2. **GUI** crée les composants de l'interface graphique en fonction de l'information du montage. Il actualise l'interface dans un boucle infini afin de visualiser le changement des états de chaque composant simultanément.
3. **Sender** sert à traiter input de l'utilisateur et envoyer l'instruction au serveur, par exemple appuyer les boutons.
4. **Lisener** reçoit l'instruction venant de serveur, en plus manipuler les composants de l'interface graphique en fonction de l'instruction.

4.1 Mécanisem de communication

La mémoire partagée sert à communiquer entre le serveur et client, puisque c'est la plus rapide manière du transport de données. Mais son inconvénient est que l'on doit implémenter un mécanisme de synchronization afin de protéger les données partagées pour la lecture et l'écriture. Donc pour deux directions du transport (du serveur au client et du client au serveur), on a besoin de deux mémoires partagées, **shm1** est écrit par le serveur, le client le lit. **shm2** est inversé.

Afin de synchroniser la lecture et l'écriture, on met un mutex et une variable conditionnelle pour chaque mémoire partagée. Il garant que chaque instruction de serveur peut être bien traiter.
Sénario de l'envoie une instruction

1. **client lisener** se bloque pour attendre l'instruction de serveur.
2. **serveur main** demande le mutex
3. **serveur main** vérifie la variable conditionnelle.
   - Si la condition se satisfait, il passe à l'étape suivant. - Sinon, cela veut dire que l'instruction dernière n'est pas encore traitée par client, **serveur main** se bloque.
4. **serveur main** envoie l'instruction au client.
5. **serveur main** débloque le thread **client lisener**.
6. **serveur main** rend le mutex.
7. **client lisener** est débloqué, demande le mutex.
8. **client lisener** vérifie si la condition se satisfait. - Si oui, cela veut dire que la nouvelle instruction arrive, il passe à l'étape suivant. - Sinon, c'est un déblocage fausse, il se bloque.
9. **client lisener** informe au **GUI** afin de visualiser le nouvel état de composant.
10. **client lisener** débloque le thread **serveur main**.
11. **client lisener** rend le mutex.
12. retour à l'étape 1.

4.2 Protocode

Protocode est un rôle de transfer les informations entre coté client et coté serveur.
0 ::25bits

(S -> C) Afficher un image représenté par 25bits sur l'écran.

1 ::x ::y ::val

(S -> C) Modifier l'état du pixel des coordonnées x et y à val.

2

(S -> C) Mettre les états de tous les pixels à 0

3 ::p ::n

(S -> C) Modifier le pin p au niveau n. (n = 0 ou 1)

4 ::p ::v

(S -> C) Modifier le pin p à la valeur v (0 <= v <= 1024).

0

(C -> S) Inverser l'état du bouton A.

1

(C -> S) Inverser l'état du bouton B.

Afin de représenter le protocol en un entier de 32 bits. on a utilisé 7bits pour le nom de fonction, alors on peut étendre jusqu'à 128 fonctions.

0 ::25bits

On juste lit 25 fois l'état de bit à partir de 8ieme bit. Pour modifier les états de pixel 0 au pixel 25. Mais cette commande n'est pas très extensible. Par exemple s'il y a plus pixel comme 10x10, alors on ne peut pas les représenter. Donc je veux plutôt réaliser cette fonction par setpixel. Cela veut dire que l'on appel le nombre de pixel fois setpixel pour afficher un image.

1 ::x ::y ::val. Cela veut dire écran[x][y]=val. x est représenté par 12bits, y aussi. val est l'état de pixel, soit 0 soit 1 donc 1 bit est suffisant. Donc la structure est 7 ::12 ::12 ::1=32bits

3 ::pin ::niveau. Pin est représenté en 8bits, niveau est 1 bit. La structure est 7 ::8 ::1=16bits

4 ::pin ::valeur. Pin est aussi représenter en 8bits. Valeur est 17bits. Donc sa structure est 7 ::8 ::17=32

La différence entre 3 et 4 est que
la commande 3 est envoyé par la fonction microbit_digital_write qui permet de mettre le niveau de pin soit 0 soit 1. la commande 4 est envoyé par microbit_analog_write qui peut modifier la valeur de pin. (0<=valeur<=1024)

5 Fonctionalités Implémentées

5.1 Afficher les phrases à l'écran du simulateur

Nous convertissons les informations de caractère dans la table Ascii en code hexadécimal 5x8 bits et les stockons dans un tableau, puis calculons le code ascii de chaque caractère un par un en recherchant. Utilisez-le pour calculer le décalage de ce caractère dans le tableau. Sortir enfin les 5 * 8 bits d'information dont il dispose. Prendre 5 * 5 bits d'entre eux et les imprimer pixel par pixel

sur l'écran.

'A'
->
Ascii code =65
->
{30,28,38,28,28}
->
00110000
00101000
00111000
00101000
00101000
->
**
* *
***
* *
* *

6 Sénarios et Tests

7 Difficulités Rencontrées

Dans le processus de développement du programme, nous avons essayé diverses méthodes pour réaliser le transfert d'informations entre le client et le serveur. Au début, nous utilisions le pipe mais il était limité par la conception du programme. Le pipe n'avait aucun moyen de synchroniser le processus en fonctionnement. Plus tard, nous avons utilisé la méthode du signal pour transmettre des informations. Mais nous avons également constaté que parce qu'une action nécessite un signal défini par l'utilisateur pour être implémentée. Cependant, sur le système mac OS, il n'y a que deux signaux personnalisés que les utilisateurs peuvent utiliser. Nous avons donc également abandonné ce plan. Après de nombreuses tentatives, nous avons utilisé la méthode de la mémoire partagée et utilisé la variable mutex et de condiction pour synchroniser les processus.

8 Conclusion