TP Raspberry Pi

Chapitre II

Cross-compiler: compilateur capable d’exécuter du code pour une autre plateforme sur la plateforme sur laquelle nous sommes.

Emulateur: logiciel qui abstrait l’hardware et permet d’exécuter du code assembleur d’un autre jeu d’instruction ou écrit dans un autre jeu d’instruction

Str : store dans la mémoire

Ldr: load dans la mémoire

Bl: saut à l’adresse indiquée

Chapitre III

Question 3.1

- Toutes les variables sont utilisées avec $.

- L’instruction de la ligne n’est pas encore executée lorsqu’on a un breakpoint.

- si : step i (step instruction en assembleur)

- print $sp : affiche la valeur de sp (stack pointer)

- p/x $r0 : affiche la valeur qu’il y a dans r0 (0x5 dans kmain)

- lr : link register

- print &nom d’une variable : donne l’adresse de la variable

- mov r3, #5 : (void \*) 0x47ff0

str r3, [sp, #4] : (void \*) 0x47ff0

ldr r0, [sp, #4] : (void \*) 0x47ff0

bl 0x812c <compute\_volume> : (void \*) 0x47ff0

* function compute\_volume  
   …  
  sub sp, sp, #20 : (void \*) 0x47fec  
  str : 0x47fd8

…

* function divide

…

str r0, [sp, #4] : (void\*) 0x47fc8

Remarque:

* cela change quand nous avons des push, pop, et quand on manipule directement sp (ex : sub)
* function Kmain:

Pour la première variable locale (radius), nous trouvons une adresse : 0x47ff4 et pour sp, nous trouvons 0x47ff0. Cela est expliqué par les crochets: on récupère l’adresse mémoire stockée dans sp(0x47ff0), on y ajoute 4 (0x47ff4) et on grâce aux crochets, on va chercher la valeur stockée dans cette adresse, et on l’attribue à r0 (ldr).

L’adresse de volume c’est la même que celle contenue dans sp.

Function Compute\_volume:

Pour rad3, nous obtenons 0x47fe4. Et pour sp, nous obtenons 0x47fd8. La difference des deux est de 12 (0xC) car nous avons fait un str de r3 à la valeur de sp +12 (adresse mémoire).

Function Divide:

Adresse de result : 0x47fd4

Adresse de remainder : 0x47fd0

Adresse contenue dans sp: 0x47fc8

Nous pouvons voir que la difference entre l’adresse de result et celle contenue dans sp est de 12 ce qui est expliqué par le fait qu’on ait fait un str de r3 à la valeur de sp +12.

De meme entre remainder et sp : la différence est de 8. Idem avec offset de 8.

* Comparer avec les paramètres:

Function divide:

Adresse de dividend: 0x47fcc

Adresse de divisor: 0x47fc8

* On remarque qu’à chaque fois nous faisons un offset pour les operations str et ldr ce qui explique la difference d’adresse entre sp et les adresses des paramètres des fonctions.
* voir schema annexe n°1
* lr:

function kmain:

Adresse de lr: 0x8080 avant l’appel de compute\_volume

Adresse de lr: 0x8198 après l’appel de compute\_volume

Function divide:

Adresse de lr: 0x8168 après l’appel de divide

On observe que c’est le bl qui va changer la valeur de lr parce que lr contient la valeur de l’adresse avant l’appel de la function, permet de garder l’adresse de la function de retour.

Bx : cela permet d’aller dans une autre function (très utile quand on a plusieurs processus)

Lr : permet de savoir où on est dans l’exécution, de restorer la valeur où on était.

* Entre 2 sp, on a un espace pour le processus.

**3.3 Un dispatcher pour coroutines**

- coroutine : procédures qui s’exécutent dans des contextes séparés

Question 3.2

Le programme permet de switcher entre deux processus en sauvegardant leur contexte grâce à ping et pong.

Question 3.3 et 3.4

Voir fichiers

Question 3.5

2x la taille du int + extra\_space (512)

* car nous avons 2 ints et extra\_space pour les différents registres (14), les variables, paramètres et toutes les instructions (y compris les iterations…) donc il faut qu’il soit assez élevé pour pouvoir tout stocker.

Question 3.6 & 3.7

Voir fichiers

Question 3.8

Avec Naked: pas de sauvegarde de sp …

Sans Naked (enlever dans le .h et dans le .c les primitives avec naked): modification de sp (soustrait sp à 8), sauvegarde de registres (sp, r0) et reinitialise la valeur de sp à sa valeur d’origine. Il fait des sauvegardes de registres => comportement normal

Question 3.9

Voir fichiers

Question 3.10

Après compilation, on remarque bien qu’il passe d’un contexte à un autre (ping puis pong en passant par la primitive de la question 3.9 puis ping et ainsi de suite). On reprend là où on s’était arrêté.

**3.4 Sauvegarde des variables locales**

Question 3.11

Il faut penser à mettre le push avant la sauvegarde du contexte courant car le compteur reviendra à 0 sinon (il reinitialise tout), ce n’est pas la même pile entre les deux contextes. Ce n’est pas le deuxième qui récupère les valeurs du premier.

Nous observons que le programme fonctionne bien.

Chapitre IV

**4.2 Passons à un nombre de processus supérieur à 2**

Question 4.1 & 4.2 & 4.3 & 4.4 & 4.5 & 4.6

Voir fichiers

Question 4.7

On remarque que nous bouclons à chaque fois car nous avons nos variables globales initialisées dans start\_sched() qui restent nulles. Lorsque nous appelons la fonction ctx\_switch() lors de la première invocation, il va restaurer le contexte du processus élu qui est nul, on ne fait donc rien.

Questin 4.8