HEIG VD | REDS Institut Reconfigurable and Embedded Digital Systems

Programmation assembleur (ASM)

Prof. Daniel Rossier
Assistants : Bruno Da Rocha Carvalho, Thomas Rieder

Pile et Fonctions, ARM ABI

lab06 (10.05.2023)

Objectifs de laboratoire

L'objectif de ce laboratoire est d'apprendre à utiliser les instructions de branchement, de se familiariser avec la pile ainsi que de se familiariser avec l'ABI ARM.

Validation du laboratoire

Un script de rendu vous est fourni afin de packager les fichiers sources modifiés sans inclure tous les fichiers générés (code objet, exécutables, etc.). Ce script s'appelle ***rendu.sh*** et sera fourni pour tous les laboratoires.

```
reds@reds2022:~/asm/asm_lab0X$ ./rendu.sh
```

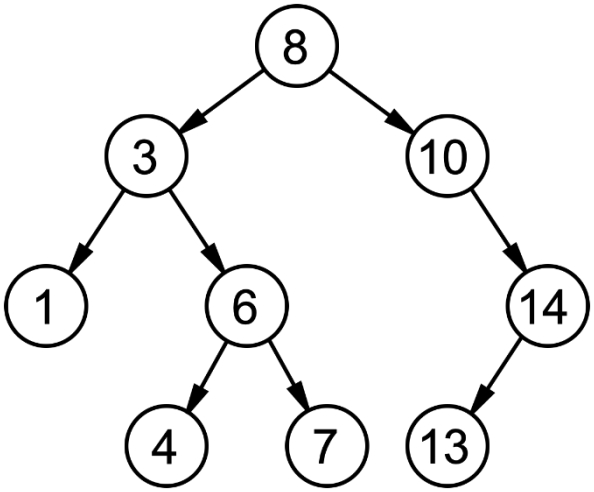
Cette commande va générer un fichier *asm_lab0X_rendu.tar.gz* (où X est le numéro du labo) et c'est cette archive qui sera à remettre sur Cyberlearn. Il est demandé de toujours préparer l'archive du laboratoire grâce au script de rendu fourni.

Récupération des sources & mise en place de l'environnement

```
reds@reds2021:~$ retrieve_lab asm lab06
```

Étape 1 – Parcours d'arbre binaire (x86)

Dans cette étape il s'agira d'écrire un algorithme pour le parcours d'un arbre binaire de recherche[1] (arbre binaire trié) à l'aide de la pile.



Un arbre binaire de recherche a deux propriétés principales :

1) Tous les nœuds ont 0,1, ou 2 fils
2) Tous les sous-nœuds (fils et petits-fils) de gauche sont inférieurs au nœud courant, tous les sous-nœuds de droite sont supérieurs au nœud courant.

Dans cette étape il s'agira d'écrire un algorithme qui permet de parcourir les nœuds d'un tel arbre dans l'ordre croissant, donc dans l'exemple ci-dessus, le parcours sera : 1, 3, 4, 6, 7, 8, 10, 13, 14.
En mémoire l'arbre utilise la représentation suivante :

```
Node {
        int  value;
        Data *data;
        Node *gauche;
        Node *droite;
};
```

Mémoire (32-bits)	Offset
Nœud * droite	+ 0xc
Nœud * gauche	+ 0x8
Data * ptr	+ 0x4
Valeur	+ 0x0

Cette structure permet non seulement de représenter l'arbre mais aussi des données associées (data). Dans notre cas le pointeur sur les données « *Data ** » sera un pointeur sur une chaîne de caractères C (char * avec '\0' à la fin) qui représente un mot. Le parcours de l'arbre représentera l'ordre des mots dans une phrase. Ceci permettra de valider l'ordre du parcours d'une seconde manière. (Si l'ordre est correct, ordre croissant, la phrase affichée sera dans le bon ordre).

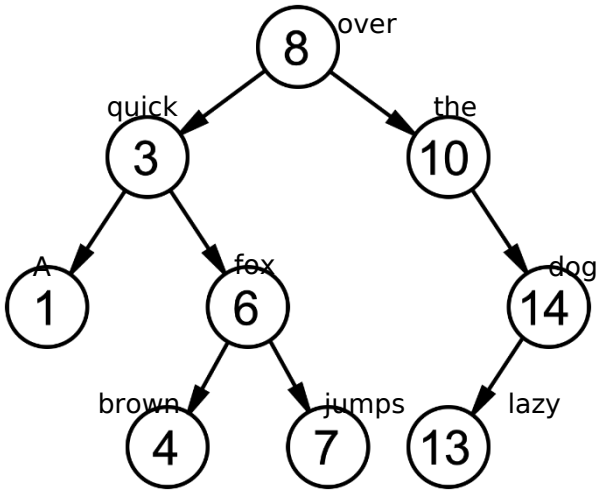
a) Écrire l'algorithme qui permet de parcourir l'arbre binaire de recherche dans l'ordre et afficher les données (*data*) correspondantes dans le fichier « ***x86/binary_tree/binary_tree_asm.S*** ».

⇒ Un pointeur sur le nœud racine de l'arbre est passé en paramètre.
⇒ Utiliser la pile pour stocker l'adresse des nœuds qui ne doivent pas encore être traités (affichés).

[1] https://fr.wikipedia.org/wiki/Arbre_binaire_de_recherche

⇒ La macro « ***PUT_S*** » permet de traiter (afficher) les data et prends en argument le pointeur sur data.
⇒ Attention ! À la fin de l'algorithme, la pile devrait être dans le même état (contenu et position du pointeur) qu'avant l'appel de l'algorithme (bien entendu, le contenu mémoire au-delà du pointeur peut avoir changé).
⇒ Lorsqu'un nœud n'a pas de fils à droite et/ou à gauche. La valeur du pointeur gauche/droite vaut 0.

À titre d'exemple, l'arbre ci-dessous produirait le message suivant : « A quick brown fox jumps over the lazy dog », si le parcours 1, 3, 4, 6, 7, 8 ,10, 13, 14 est suivi.



Si un noeud ("2" – "**super**") était inséré, le message « A **super** quick brown fox jumps over the lazy dog » serait alors affiché.

Étape 2 – ARM ABI AAPCS – Backtrace

Dans cette étape nous allons écrire une fonction qui permet d'afficher une « backtrace »[2] aussi appelée « stack trace ». C'est la trace hiérarchique des appels de fonctions effectués dans un programme à un endroit donné. En général la « backtrace » est utilisée lors de débug, avec GDB par exemple et permet de savoir quelle est la séquence d'appels de fonction jusqu'à l'endroit donné.

Le but de cet exercice sera d'afficher les adresses de retour vers les fonctions appelantes. Par exemple si « main() » appelle « foo() » qui appelle « bar() » et qu'on aimerait afficher la « backtrace » dans « bar() » on aurait des adresses dans « bar() – foo() – main() ». Afin de réaliser ceci, nous allons parcourir la pile « stack ».

`[fp, #12]`		- 3ème argument supplémentaire de la fonction
`[fp, #8]`		- 2ème argument supplémentaire de la fonction
`[fp, #4]`		- 1er argument supplémentaire de la fonction
`fp`	**`lr`**	- adresse de retour
`[fp, #-4]`	**`fp`**	- valeur de ***fp*** préservée (ancienne)
`[fp, #-8]`		- 1ère variable locale
`[fp, #-12]`		- 2ème variable locale
`[fp, #-16]`		- 3ème variable locale

[2] https://sourceware.org/gdb/current/onlinedocs/gdb/Backtrace.html#Backtrace

Selon l'ABI ARM AAPCS chaque fonction appelée va enregistrer l'adresse de retour (vers la fonction appelante) ainsi que le « frame pointer » (de l'appelant) sur la pile.

Pour l'exercice nous avons instrumenté la pile afin qu'avant que « main() » soit appelé, les valeurs -1, -1 (0xFFFF'FFFF, 0xFFFF'FFFF) soient écrits sur la pile avant d'appeler « main() ». Ceci nous permettra de trouver la base de la pile pour notre application[3]. Ceci est réalisé dans le fichier « crt0.S » juste avant d'appeler « main() ».

Il s'agira ici d'écrire le code assembleur requis pour afficher les adresses de retour (« lr ») enregistrés sur la pile. Les valeurs « fp » sauvegardées nous permettront de localiser les piles des appelants. Jusqu'à ce que les valeurs de -1, -1 (pour « fp » et « lr ») soient trouvées (c'est le sommet de la chaîne d'appels).

a) Compléter la fonction « show_backtrace() » dans le fichier « arm/backtrace/backtrace_asm.S » qui va afficher l'un après l'autre la valeur du « lr » précédent jusqu'au sommet de la pile.

b) Tester dans QEMU avec l'application « backtrace.bin »[4]. Cette application va appeler des fonctions imbriquées dans un ordre aléatoire, jusqu'à appeler « show_backtrace() ».

c) Le log de la sortie de QEMU (« ./st ») est enregistré dans le fichier « st.log ». Grâce au script « ./addr2fun.sh » la sortie de ce fichier est filtrée et les adresses sont traduites vers les noms de fonctions correspondantes[5] (voir exemple ci-dessous). Ceci permet de valider que les adresses correspondent bien aux fonctions appelées (l'ordre sera en miroir des appels puisqu'on affiche la valeur de lr vers l'appelant depuis la fonction la plus profonde).

```
~/asm22_labo6$ less st.log
…
----
Function main called
Function function_D called

Backtrace :
----
Previous link register : 0x820003a0
Previous link register : 0x820004cc
Previous link register : 0x82000020
----
Hit any key to exit ...
…

~/asm22_labo6$ ./addr2fun.sh
----
Function main called
Function function_D called
----
function_D at backtrace.c:87 (discriminator 1) <= 0x820003a0
main at backtrace.c:100 <= 0x820004cc
$a at crt0.S:23 <= 0x82000020
----
```

⇒ Attention à ne pas corrompre (modifier) la pile.
⇒ Attention à ne pas écraser des registres non « scratch », les sauvegarder sur la pile et les restaurer si vous en faites usage.
⇒ Utiliser la fonction « printf » afin d'afficher les adresses avec la « format string » fournie dans le fichier assembleur.
⇒ Attention la fonction « printf » peut modifier les registres « scratch ».

[3] Ceci n'est pas standard et normalement c'est au système d'exploitation de définir où débute la pile. Ici nous sommes dans un environnement « bare-metal » avec juste le bootloader « u-boot » alors la pile continue juste là où elle se trouvait au lancement (« go ») de notre application.
[4] Ce fichier est copié sous le nom « bt » à la racine du labo afin de pouvoir faire « tftp bt ».
[5] Cette traduction est réalisée grâce à l'utilitaire « addr2line » https://linux.die.net/man/1/addr2line et le fichier ELF contenant les symboles de débug.