Du désassemblage de syscalls

lephe

(Première version)

1 Introduction

Les syscalls sont des procédures qui constituent l'OS comme le ferait une bibliothèque standard. Leur code nous renseigne sur le fonctionnement du système, les drivers des périphériques et plus particulièrement sur les modules du MPU, ce qui permet jusqu'à un certain point de reconstituer la documentation inexistante.

Ce tutoriel se propose de présenter le fonctionnement des syscalls en tant qu'outil, et de démontrer une manière de les désassembler pour en extraire des informations intéressantes, généralement pour le développement d'add-ins.

1.1 Prérequis

Pour ce tutoriel, je suppose que vous savez vous servir d'un éditeur hexadécimal pour accéder aux contenus d'un fichier, et que vous avez une sauvegarde d'un OS si vous voulez reproduire les manipulations. Voyez les liens suivants s'il vous manque quelque chose. Notez que la Bible de TeamFX (nombreux fichiers de documentation, sauvegardes d'OS, et bien plus) est un must-have ; je vous conseille vivement de la télécharger, et j'y ferai souvent référence dans la suite.

— HxD (éditeur hexadécimal pour Windows)
— Bless (éditeur hexadécimal pour Linux)
— Bible de TeamFX

Je suppose également que vous avez les connaissances en Assembleur pour comprendre le code que vous allez désassembler. C'est rarement une tâche aisée et ce sujet mériterait un bon tutoriel à lui seul. Si vous bloquez sur une instruction mal connue ou un doute sur l'expansion du bit de signe lors d'un déplacement d'un octet vers un longword, référez-vous à la documentation du processeur, qui contient toutes les informations sur les instructions et leur fonctionnement. Vous trouverez ces fichiers dans la Bible de TeamFX.

Vous aurez sans doute besoin également des informations concernant les MPU... vous pouvez également les trouver dans la Bible de TeamFX (si si !). Pour rappel, le MPU utilisé par les calculatrices SH-3 est très proche du SH7705 (la grande majorité des informations est réutilisable), tandis que les machines équipées d'un processeur SH-4A utilisent une variante du SH7724 (il y a beaucoup plus de différences). Gardez en tête que ce ne sont pas *exactement* les MPUs présents dans les machines, et certaines informations peuvent être inexploitables ou erronées.

Notez enfin, que les adresses logiques de la forme `8xxxxxxx` que vous trouverez dans les programmes, pointent vers la même mémoire physique que les adresses sous la forme `axxxxxxx`, et correspondent à une position dans le fichier qui est `0xxxxxxx`.

2 Un peu de théorie

La méthode que nous allons utiliser dans ce tutoriel consiste à reproduire manuellement la procédure qui est exécutée par le système lors de l'appel d'un syscall. Ça fonctionne très bien, mais vous allez voir que c'est assez compliqué [1]. Déblayons donc tout ça avec un peu de théorie, comme ça le code roulera tout seul.

2.1 La table de syscalls

Il y a deux manières communes de construire le support technique d'un OS : une librairie standard, ou un ensemble de syscalls. Les systèmes modernes installés sur les ordinateurs utilisent les deux, mais la calculatrice se contente de posséder une table de syscalls.

Chaque syscall est représenté par un numéro, et la table de syscalls associe à chaque numéro l'adresse du code (dans la plupart des cas ; parfois il s'agit d'une table de données). Notez que la table suivante suppose un processeur 32 bits (comme les SuperH-3 et SuperH-4A), de sorte que la longueur d'une adresse est de 4 octets.

Adresse	*Contenu*
table	Adresse du syscall `0x000`
table + 4	Adresse du syscall `0x001`
...	...
table + 4n	Adresse du syscall `n`
...	...

Pour utiliser un syscall, il convient de connaître son numéro pour lire son adresse dans la table des syscalls. Il suffit ensuite d'y sauter à l'aide d'un pointeur [2] pour exécuter la fonction. On récupère en sortie les valeurs de retour dans les registres ou sur la pile comme on le ferait avec n'importe quelle fonction.

Pour concrétiser un peu cette présentation, voici quelques syscalls classiques utilisés par la calculatrice :

Numéro	*Prototype*
`0x03b`	`int RTC_GetTicks(void)`
`0x135`	`char *GetVRAMAddress(void)`
`0x23e`	`void RTC_SetDateTime(unsigned char *time[7])`
`0x24a`	`int Keyboard_IsAnyKeyDown(short *matrixcode)`
`0x420`	`void Sleep(int milliseconds)`
`0xacd`	`void *malloc(size_t size)`

Dans le cas de la calculatrice, la procédure d'appel est essentiellement gérée par le système. Il suffit de demander l'exécution du syscall n en indiquant les arguments pour que ça roule. Et pourtant, en pratique, c'est un peu plus subtil. Puisqu'on est là pour parler d'Assembleur, je vais vous donner les détails techniques un peu gores.

2.2 Rappel sur les ABI

Un petit rappel sur les ABI pourrait être utile. Une ABI (*Application Binary Interface*) est un ensemble de conventions qui décrit entre autres, au niveau du code Assembleur, de quelle manière on passe les paramètres et les valeurs de retour d'un appel de fonction à l'autre. Deux programmes compilés en suivant la même ABI sont basiquement compatibles, et deux programmes utilisant différentes ABI ne le sont définitivement pas.

Dans le cas des processeur SuperH, les conventions incluent l'usage du registre *r15* pour l'adresse de la pile, l'obligation de restaurer les registres *r8* à *r15* si on les utilise dans une fonction (alors que les valeurs de *r0* à *r7* peuvent être modifiées sans préavis), le passage des arguments dans les registres *r4* à *r7* (puis sur la pile, s'il y en a plus), l'utilisation de *r0* comme valeur de retour, et bien d'autres.

1. En tous cas, assez pour que vous n'ayez pas envie de le refaire à chaque fois.
2. Seulement si vous avez le luxe de le programmer en C.

2.3 Technique d'appel générique de syscall

Tout le problème de l'appel d'un syscall se trouve dans l'implémentation d'une fonction théorique `void syscall(int no, ...)` capable d'appeler un syscall tout en transférant les paramètres qu'elle a reçu. Les langages modernes de haut niveau en sont capables en utilisant des techniques de haut niveau, mais on s'intéresse ici à une méthode de plus bas niveau que le langage C. On ignorera par ailleurs le fait qu'une telle fonction ne peut correspondre, en termes de prototype, à tous les appels de syscalls, parce que sa valeur de retour est d'un type fixé, alors que les syscalls renvoient des données de types très variés.

Le problème de la fonction `syscall()` se trouve dans la transmission des paramètres. Il y a une méthode simple assez intuitive : étant donné que le compilateur placera, pour réaliser l'appel à la fonction `syscall()`, les arguments à leur place dans les registres, il suffirait de sauter à l'adresse du syscall pour lui transférer les arguments, qu'ils soient dans les registres ou sur la pile. Au fond, il n'y a rien à changer.

Seulement, la fonction `syscall()` ne reçoit pas tout à fait les mêmes arguments que le syscall : elle prend en plus le numéro du syscall, ce qui induit un décalage dans l'ordre des registres. Le registre *r4* est utilisé pour placer l'id du syscall, donc le premier paramètre se retrouve placé dans *r5*, et un argument qui aurait dû occuper *r7* se retrouve sur la pile. Il n'est pas possible d'effectuer le décalage inverse car le nombre d'arguments est inconnu.

La première solution consiste à passer en plus le nombre d'arguments, ce qui permettrait d'annuler le décalage, mais cela présente deux inconvénients : d'une part c'est lourd, d'autre part c'est dépendant de l'ABI — c'est donc non-portable au possible. La seconde solution, adoptée par le système, consiste à passer le numéro du syscall ailleurs que dans les paramètres, en l'occurrence dans le registre *r0*. Cela rend l'appel incompatible avec l'ABI du compilateur et explique que les procédures d'appel soient écrites en Assembleur. Voyons tout cela plus en détail.

3 Une méthode manuelle

Pour comprendre le fonctionnement réel des syscalls, il n'y a rien de mieux que d'en exécuter un soi-même de bout en bout. Je vous propose ici de reproduire le fonctionnement complet des appels de syscalls, quitte à en désassembler quelques-uns au passage.

3.1 Interface avec la bibliothèque

Au fond, à quoi ressemble un appel de syscall ? Le meilleur moyen d'en observer est définitivement la bibliothèque *fxlib*. Elle passe son temps à en faire ; par exemple, si vous tentez de regarder le code de la fonction *malloc()*, vous tombez sur ceci :

```
0:  d201   mov.l   <8>(#0x80010070), r2
2:  d002   mov.l   <c>(#0x00000acd), r0
4:  422b   jmp     @r2
6:  0009   nop
8:  8001
a:  0070
c:  0000
e:  0acd
```

De nombreuses fonctions de fxlib ressemblent à ça. Elles se contentent de sauter à l'adresse `80010070` après avoir placé une valeur cryptique dans le registre *r0*. Vous l'aurez compris, il s'agit du numéro de syscall et ces quelques lignes de code sont la partie de l'appel qu'on ne peut pas écrire en C parce qu'elle utilise un registre non conventionnel pour passer un paramètre.

L'adresse `80010070` désigne donc la localisation de la fonction `syscall()` décrite précédemment, qui reçoit un numéro de syscall dans le registre *r0* et divers paramètres dans les registres *r4* à *r7* et sur la pile, qu'elle transmet sans même savoir qu'ils sont là.

3.2 La fonction d'appel générale

Allons donc désassembler cette localisation. On y trouve le code suivant :

```
10070:  d202   mov.l   <1007c>(#0x801cdd84), r2
10072:  4008   shll2   r0
10074:  002e   mov.l   @(r0, r2), r0
10076:  402b   jmp     @r0
10078:  0009   nop
1007a:  0000   --
1007c:  801c
1007e:  dd84
```

Cette fonction n'est pas très compliquée, mais il faut faire attention aux différents niveaux de pointeurs. Ici, elle charge une certaine adresse dans *r2*, vue comme l'adresse d'un tableau dont chaque élément fait 4 octets, et va y lire la *r0*-ème entrée. Puis, elle saute directement à l'adresse obtenue, comme pour transférer l'exécution à une fonction. Cette procédure correspond à l'utilisation attendue de la table de syscalls, que l'on localise donc à l'adresse `801cdd84` (notez que cette adresse peut changer si vous utilisez un autre OS que le mien).

Pour cet exemple, je vais désassembler le syscall `0x420` d'un OS SH-4A, qui est supposé endormir la machine pendant un certain temps passé en paramètre sous la forme d'un délai en millisecondes. Il s'agit donc d'aller lire la `0x420`-ème entrée de la table de syscalls. Un calcul rapide indique que l'adresse recherchée se trouve à `801cdd84 + 4 * 420 = 801cee04`. Voici ce que je trouve dans le voisinage de cette adresse :

```
1cee00:  8006
1cee02:  62c8
```

```
1cee04:  8006
1cee06:  5b6c
1cee08:  8006
1cee0a:  5bf2
```

La table nous indique gentiment que le syscall que nous recherchons réside à l'adresse `80065b6c`. On touche enfin au but !

3.3 Le Saint Graal

Je ne vous fais pas plus attendre, voici le code que l'on trouve à l'adresse indiquée. Notez le problème qui se pose pour localiser la fin du syscall en termes d'adresse. Ici, la fonction est simple, la première occurrence de *rts* marque la fin de l'exécution dans tous les cas de figure. Des syscalls plus complexes auront un flux d'exécution autrement plus compliqué comportant plusieurs apparitions de *rts* pas toujours exécutées au même moment, sans compter les sous-routines qui peuvent provenir d'autres syscalls, ou pas. Sans outil informatique (type un debugger), difficile de détecter avec précision jusqu'où il faut désassembler. En général, vous pourrez vous contenter de désassembler un bloc large et d'élaguer au fur et à mesure de votre analyse.

```
65b6c:  4f12   sts.l   macl, @-r15
65b6e:  4f02   sts.l   mach, @-r15
65b70:  4400   shll    r4
65b72:  2448   tst     r4, r4
65b74:  8918   bt      <65ba8>
65b76:  9297   mov.w   <65ca8>(#0x0fa0), r2
65b78:  3426   cmp/hi  r2, r4
65b7a:  8915   bt      <65ba8>
65b7c:  d195   mov.l   <65dd4>(#0x000f4240), r1
65b7e:  e5f2   mov     #-14, r5
65b80:  e7ff   mov     #-1, r7
65b82:  0147   mul.l   r4, r1
65b84:  011a   sts     macl, r1
65b86:  d494   mov.l   <65dd8>(#0x78c06387), r4
65b88:  3415   dmulu.l r1, r4
65b8a:  060a   sts     mach, r6
65b8c:  465d   shld    r5, r6
65b8e:  d593   mov.l   <65ddc>(#0xa4490004), r5
65b90:  1577   mov.l   r7, @(28, r5)
65b92:  1578   mov.l   r7, @(32, r5)
65b94:  6050   mov.b   @r5, r0
65b96:  cb04   or      #4, r0
65b98:  2500   mov.b   r0, @r5
65b9a:  5258   mov.l   @(32, r5), r2
65b9c:  6227   not     r2, r2
65b9e:  3262   cmp/hs  r6, r2
65ba0:  8bfb   bf      <65b9a>
65ba2:  6050   mov.b   @r5, r0
65ba4:  c9fb   and     #-5, r0
65ba6:  2500   mov.b   r0, @r5
65ba8:  4f06   lds.l   @r15+, mach
65baa:  000b   rts
65bac:  4f16   lds.l   @r15+, macl
```

Notez que cette implémentation de `Sleep()` comporte quelques surprises et informations intéressantes. Sans être exhaustif, voici un extrait de ce que l'on peut comprendre avec cette analyse.

D'abord, la conversion du paramètre temporel fourni dans le registre *r4* en compteur pour le timer. On voit des références à plusieurs constantes (une multiplication par 2 à `65b70`, la comparaison à 4000 effectuée à `65b78`, le produit par 1000000 à `65b82`, entre autres). Le problème étant qu'il n'y a que des constantes. Aucun accès aux paramètres de la machine n'est effectué, alors que la vitesse de décompte du timer est dépendante de la fréquence P_ϕ de l'horloge périphérique. Ceci explique pourquoi `Sleep()` ne respecte plus le délai temporel passé en paramètre lorsqu'on overclocke la machine.

Autre mauvaise surprise, aucune invocation de l'instruction *sleep* n'est effectuée. Pour ceux qui ne la conaissent pas, cette instruction met le processeur en veille jusqu'à ce qu'il soit réveillé par un événement (selon les critères définis dans la configuration du MPU). En général une interruption suffit pour reprendre l'exécution du programme, ce qui rend facile l'utilisation d'un timer. Ici, la fonction se contente de réaliser une attente active en testant à `65b9e` la condition `r6 < ~r5[32]`, ce qui revient à tester si le compteur (initialisé à `ffffffff`) a parcouru `r6` valeurs, seuil issu des différents produits utilisés lors de la conversion du paramètre temporel en compteur pour le timer. En d'autres termes, cette fonction boucle rapidement en testant la valeur du compteur tous les quelques cycles processeur. Autant dire que vos piles ne vont pas apprécider...

Reste les effets positifs. La configuration du timer effectuée à partir de `65b8e` nous révèle la localisation dans l'espace logique d'adressage, du module périphérique gérant les timers. Cette information, qui aurait dû se trouver dans la documentation du SH7305 (rappelons que cette documentation n'existe pas car elle est protégée par des accords de non-publication), est d'autant plus importante que cette adresse ne correspond à aucune adresse issue de modèles proches (ceux dont la documentation existe). Par chance, la structure du module est elle, partagée. Un peu d'observation sur l'ordre des opérations (et la désactivation du timer juste avant la fin de la fonction) indique alors que le registre *TSTR* se trouve à l'adresse `a4490004` et que la troisième structure de timer est localisée à `a4490020`. On obtient finalement les adresses suivantes :

Adresse	Rôle dans le module gérant les timers
`a4490004`	Registre *TSTR*
`a4490008`	Structure du premier timer
`a4490014`	Structure du second timer
`a4490020`	Structure du troisième timer

Cette information reste la plus importante du point de vue de l'exploitation qu'on peut en faire dans les programmes.

4 Une méthode automatique

Comme promis, vous n'avez pas envie de refaire ce travail à chaque fois [3]. Pour automatiser cette tâche, le fxSDK propose un outil nommé *fxos* dont le travail est (entre autres) de désassembler les OS et en particulier les syscalls.

Pour ceux qui auraient pensé à utiliser *sh3eb-elf-objdump*, c'est tout à fait possible, mais il ne reconnaît pas toutes les instructions (les multiplications, notamment, lui échappent), et cela ne vous épargne pas la longue procédure de recherche décrite dans la section précédente.

fxos se contente de reproduire la technique manuelle à partir du numéro de syscall :

```
$ fxos disasm os_sh4.fls -s 0x420
Syscall table:    0x801cdd84
Syscall entry:    0x001cee04
Syscall id:       0x420
Syscall address:  0x80065b6c

 65b6c:  4f12   sts.l   macl, @-r15
 65b6e:  4f02   sts.l   mach, @-r15
 65b70:  4400   shll    r4
 65b72:  2448   tst     r4, r4
 65b74:  8918   bt      <65ba8>
 65b76:  9297   mov.w   <65ca8>(#0x0fa0), r2
 65b78:  3426   cmp/hi  r2, r4
 65b7a:  8915   bt      <65ba8>
 65b7c:  d195   mov.l   <65dd4>(#0x000f4240), r1
 65b7e:  e5f2   mov     #-14, r5

note: defaulted to 20 bytes, use -l for more.
```

La documentation de l'outil pourra fournir de plus amples informations sur les options de désassemblage et les autres commandes.

3. Ou alors, ça viendra après quize ou vingt répétitions.

5 Conclusion

Merci d'avoir lu ce tutoriel jusqu'au bout ! Le désassemblage de syscalls ne devrait plus avoir de secrets pour vous maintenant.

Tout ceci n'est que le début ; la véritable difficulté reste dans l'interprétation du code obtenu. Si vous voulez perfectionner votre compréhension de l'Assembleur, la documentation du processeur vous sera d'une grande aide. Sinon, tournez-vous vers la documentation du MPU, qui recèle les explications aux nombreux accès apparemment aléatoires à des adresses logiques.