TD1 – **Programmes : du code source à l'exécution**

{nollinge,Emmanuel.Godard,esposito}@cmi.univ-mrs.fr

6 octobre 2004

☞ *L'objet de ce TD est de s'initier au cycle de vie des programmes, depuis leur compilation à partir du code source, jusqu'à leur environnement d'exécution et leurs interactions avec le système d'exploitation. Ce TD mêle des aspects d'architecture des ordinateurs à des aspects de compilation. Ce TD est aussi une initiation à notre architecture de référence et aux systèmes POSIX.*

Introduction

Le but d'un système d'exploitation est avant tout de fournir des services qui permettent de faire fonctionner des programmes en leur proposant une vue abstraite des ressources.

Dans ce TD nous allons suivre le cheminement depuis le programme source C `toto.c` ci-dessous jusqu'à l'exécution du programme `toto` qu'il engendre :

```
1   #include <stdio.h>
2
3   long int factorielle(int n)
4   {
5      if (n<2)
6        return 1;
7      else
8        return n*factorielle(n-1);
9   }
10
11  int main(int argc, char *argv[])
12  {
13    int i;
14
15    printf("Hello World!\n\n");
16    for(i=0;i<10;i++)
17      printf("%i! = %li\n",i,factorielle(i));
18    return 0;
19  }
```

La compilation d'un programme aussi simple s'effectue habituellement en une seule commande :

```
% cc -o toto toto.c
```

En réalité, cette opération se découpe en trois phase distinctes :

compilation. le code source C est transformé en code assembleur (production de `toto.s`)

```
% cc -c -S toto.c
```

assemblage. le code assembleur est transformé en un fichier objet (production de `toto.o`)

```
% as -o toto.o toto.s
```

édition de liens. le fichier objet est lié aux fonctions de bibliothèque et à l'environnement d'exécution C (production de l'exécutable `toto`)

```
% ld -o toto crt0.o toto.o libc.a
```

1 Le processeur Ant-32

☞ *Lire dans le document sur l'architecture de référence uniquement les sections 1, 2.1 et regarder rapidement les familles d'instructions 2.5.1 à 2.5.5.*

Un certain nombre d'instructions usuelles semblent faire défaut à ce processeur. Nous allons montrer comment les mettre en oeuvre. On notera dans la suite `ze` le registre `r0`.

Q1. Proposer un codage pour les instructions :
- `mov des, src1` : recopie du contenu de `r[src1]` dans `r[des]` ;
- `j const32` : saut à l'adresse `const32` ;
- `j src1` : saut à l'adresse `r[src1]` ;
- `b const32` : branchement à l'adresse `const32` ;
- `b src1` : branchement à l'adresse `r[src1]`.

Q2. Gestion de pile. En supposant que le registre `sp` soit affecté à la pile, proposer un codage pour les instructions d'empilement et de dépilement :
- `push src1` : empile `r[src1]` ;
- `pop des` : dépile dans `r[des]`.

Q3. Gestion d'appel de fonction. En supposant que le registre `ra` soit affecté à l'adresse de retour dans un appel de fonction, le registre `fp` à l'adresse du bloc d'information, et le registre `g0` au stockage de la valeur de retour, proposer un codage pour les instructions :
- `call const32` : saute à l'adresse `const32` en plaçant l'adresse de retour dans le registre `ra` ;
- `entry const32` : empile `fp` et `ra`, met la valeur de `sp` dans `fp`, alloue `const32` octets dans la pile pour les variables locales ;
- `return src1` : met la valeur `r[src1]` dans `g0`, met la valeur de `fp` dans `sp`, dépile `ra` et incrémente-le de 4, dépile `fp`, saute en `ra` ;
- `return const32` : idem en utilisant `const32` comme valeur de retour au lieu de `r[src1]`.

On appelle une fonction avec `call` après avoir empilé les arguments. La fonction débute par `entry` et termine avec `return`. Dessiner l'état typique de la pile juste après l'exécution de `entry`.

2 Compilation

La compilation consiste à transformer le source C en un source assembleur. On supposera que l'assembleur suit nos conventions d'utilisation des registres et connaît `ze`, `ra`, `sp`, `fp`, ainsi que les registres généraux non utilisés `g0` à `g55` qui sont disponibles pour le calcul et quelques registres spéciaux `u0` à `u3` qui sont réservés pour la mise en oeuvre des macro-instructions `mov`, `j`, `b`,`push`, `pop`, `entry`, `call` et `return`.

Un fichier source assembleur est constitué d'une suite d'instructions dans laquelle s'insèrent :
- des définitions de labels qui désignent des endroits spécifiques du fichier (ex. `factorielle:`) ;
- des identificateurs de sections (`.text` et `.data`) qui séparent les parties du programme qui sont du code et des données. Une déclaration de section est valable jusqu'à la prochaine déclaration ;
- des déclarations de données brutes (ex. `.ascii "titi\000"`, `.word 0x12345678`, `.byte 65`) ;
- des déclarations d'alignement (ex. `.align 4`) qui demande l'alignement en mémoire de ce qui suit par rapport à un certain nombre d'octets (ex. alignement sur une adresse multiple de 4) ;
- des déclarations d'exportation de symboles (ex. `.global main`).

Q1. Proposer un fragment de code assembleur correspondant au corps de la fonction `factorielle`.

Q2. Proposer un fragment de code assembleur correspondant à l'en-tête de la fonction `factorielle`.

Q3. Proposer un fragment de code assembleur correspondant au corps de la fonction `main`.

Q4. Proposer un fragment de code assembleur correspondant à l'en-tête de la fonction `main`.

3 Assemblage

Le modèle mémoire des systèmes POSIX est simple : le code exécutable est situé en bas de la mémoire, en dessous des données qui vont croître pendant l'exécution au fur et à mesure de l'allocation dynamique (`malloc`), enfin tout en haut de la mémoire se trouve la pile qui elle grandit vers le bas de la mémoire.

L'assemblage consiste à transformer le code source assembleur en un fichier objet. Ce fichier (au format `a.out`, `ELF` ou encore `COFF` par exemple) contient généralement une entête qui identifie le type de fichier (fichier objet, fichier exécutable), et des informations sur les autres sections du fichier : le code sous forme de langage machine (une suite d'octets), les données sous la même forme, ainsi qu'une table des symboles.

Q1. Dessiner schématiquement la mémoire d'un programme en cours d'exécution ainsi que la structure d'un fichier objet. Expliquer à quoi peut bien servir la table des symboles.

Q2. Assembler les sections `.text` et `.data` de `toto.s`.

Q3. Donner la structure de `toto.o` et expliciter sa table des symboles.

4 Édition de liens

La phase d'édition de liens combine différents fichiers objet (les librairies dites statiques ne sont que des collections ordonnées de fichiers objet) en un fichier exécutable.

Q1. L'éditeur de liens lie aux fichiers objet C un fichier objet généralement appelé `crt0.o` dit environnement d'exécution C et qui contient le point d'entré du fichier exécutable, correspondant au symbole `__start`. Expliciter le rôle du code associé à ce symbole.

Q2. La fonction `printf` fait partie de la librairie C, stockée dans `libc.a`, qui fait appel à la fonction `write` sur la sortie standard `stdout`. La fonction `write` est mise en oeuvre par le biais d'un appel système car un programme utilisateur ne peut accéder directement au matériel. Expliquer comment réaliser un appel système grâce à l'instruction `trap`.

Q3. Les appels systèmes sont gérés par le processeur Ant-32 comme des exceptions, de même que les requêtes des périphériques ou les erreurs d'exécution (division par zéro, etc). Proposer un squelette de gestionnaire d'exception et expliciter le retour en mode utilisateur et la gestion des drapeaux.

5 Environnement d'exécution

L'exécution dans un shell du programme `toto` produit le comportement suivant :

```
% ./toto
Hello World!

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
```

```
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
```

Le shell exécute le programme en exécutant l'appel système `fork` puis en utilisant `execve`.

Q1. Définir les tâches que doit effectuer `execve` et proposer un squelette pour cette fonction.
Q2. Dans le cas d'un système multitâches, le système d'exploitation doit reprendre périodiquement la main, par exemple pour attribuer le processeur à un autre programme. Expliquer comment un tel comportement est possible sans la coopération des programmes utilisateur.