Compilation

TP 4 : Organisation des données

A. Isoard & C. Alias

Dans les TP précédents nous avons vu comme analyser le texte du programme (TP 1 et 2), et comment vérifier les types (TP 3). Dans ce TP, nous allons maintenant nous intéresser à la production de code à proprement parler. La première étape est de déterminer l'emplacement et la structure des données.

Exercice 1. *Fonctions (à la main)*

On reprend les programmes vus en cours :

```
void g()
{
  printf('g');
  g();
}

void f()
{
  printf('f');
  g();
}

void main()
{
  printf('m');
  f();
  g();
}
```

```
void f(int n)
{
  int m;
  m = n;
}

void main()
{
  f(0);
}
```

Manip.

- Le fichier `simple0_mask.asm` contient le squelette du programme de *gauche*. **Complétez-le** et **testez sur** DigMIPS.
- Le fichier `simple1_mask.asm` contient le squelette du programme de *droite*. **Complétez-le** et **testez sur** DigMIPS.
  - **Modifiez** `simple1_mask.asm` pour que `f` retourne `m`, et pour que `main` affiche sa valeur.

Exercice 2. *Fonctions (avec* DigCC*)*

Le fichier `simple1_compiled.asm` contient le code assembleur généré par notre compilateur, DigCC.
Le fichier .asm généré par DigCC comporte trois parties :

Code d'initialisation (lignes 1 – 47). Initialisation de la pile (ligne 1), initialisation du tas (appel à la fonction ___init_heap() ligne 16), et enfin appel à la fonction main() (ligne 38).

Programme (lignes 48 – 114). On trouvera les fonctions f() (lignes 48 – 68) et main() (lignes 69 – 114).

Bibliothèque d'exécution (lignes 115 – 397). Il s'agit d'une couche système rudimentaire :

- `void __init_heap()` (lignes 115 – 158). Initialise le tas. Toujours appelé dans le code d'initialisation.
- `void* malloc(int size)` (lignes 159 – 222). Alloue `size` octets dans le tas, et retourne l'adresse du premier élément (un pointeur, donc).
- `void free(void* data)` (lignes 223 – 265). Libére la zone du tas qui commence à l'adresse `data`.
- `void print_char(char c)` (lignes 266 – 288). Affiche le caractère `c` à l'écran.
- `void print_int(int n)` (lignes 289 – 319). Affiche l'entier `n` à l'écran. On suppose $0 \leq n \leq 19$.
- `void print_string(char* s)` (lignes 320 – 350). Affiche la chaine de caractère `s` (p. ex. "bonjour"). Utiliser avec modération.
- `void print_newline()` (lignes 351 – 372). Passe à la ligne.
- `char input_char()` (lignes 373 – 397). Lit un caractère au clavier.

Il ne s'agit bien sur que du minimum vital. On pourrait ajouter des fonctions arithmétiques (*, /, $\sqrt{}$, sin, cos, tan, etc), des fonctions de dessin, etc.
La bibliothèque d'exécution a elle même été compilée (une fois pour toutes) à partir de `runtime.c`

Manip.

- **Inspectez le code** des fonctions de la bibliothèque d'exécution. En particulier les routines de gestion du tas (___init_heap, malloc, free). Comparez avec ce qui vous a été présenté en cours.
- Comment la fragmentation est-elle gérée par `free()` ?