Petite API de la bibliothèque bfd nécessaire au projet

23 septembre 2009

Résumé

En guise de correction, je vais donner la description des principales fonctions de la bibliothèque bfd utilisé. Je tente de présenter cette bibliothèque de manière progressive afin de servir de guide à la réalisation du projet.

Dieu reconnaîtra les siens... mais ils seront dans un sale état en arrivant.

Alexandre 20:29

L'objet de base de la bibliothèque bfd est un type structure du même nom qu'il est hors de question de décrire dans son ensemble

Que doit-on en retenir alors ?

À chaque fichier objet sera attaché une variable de type bfd qui donne accès à l'ensemble des informations nécessaires. Ces dernières sont extractibles par le biais de fonctions déjà écritent et constituant une API. Le projet consiste donc à bien déterminer ce que l'on cherche et comment le trouver.

1 Placer l'information associé à un fichier dans une structure bfd

Cuisine interne. Pour commencer, il faut initialiser des données internes et pour ce faire, on dispose de :

```
 void bfd_init(void);
   *Description*
This routine must be called before any other BFD function to initialize
magical internal data structures.
```

De plus, on a tout intérêt à utiliser le fichier d'entête `#include <stdbool.h>`.

La compilation du code nécessite la libraire bfd et on peut invoquer le compilateur comme suit :

```
gcc foo.c -lbfd
```

Entrée-sortie. Un fichier étant donné, il est possible d'assigner une variable de type bfd trés simplement par :

```
     bfd *bfd_openr(const char *filename, const char *target);

Open the file FILENAME (using ‘fopen’) with the target TARGET.  Return
a pointer to the created BFD.
```

Cette instruction a son pendant :

```
     bfd_boolean bfd_close (bfd *abfd);

Close a BFD. If the BFD was open for writing, then pending operations
are completed and the file written out and closed.  If the created file
is executable, then ‘chmod’ is called to mark it as such.

   All memory attached to the BFD is released.

   The file descriptor associated with the BFD is closed (even if it
```

```
was passed in to BFD by ‘bfd_fdopenr’).

   *Returns*  ‘TRUE’ is returned if all is ok, otherwise ‘FALSE’.
```

qui ne nous sera pas utile puisque l'on ne fait que lire la table des symboles et que l'on ne cherche pas à écrire (mais autant être propre).

Quel est le type du fichier. En nous restreignant à ce qui nous concerne, le fichier que nous venons de considérer peut être :

- un fichier objet ;
- ou une archive.

La différence entre ses 2 types est donnée par :

```
   * ‘bfd_object’
   The BFD may contain data, symbols, relocations and debug info.

   * ‘bfd_archive’
   The BFD contains other BFDs and an optional index.
```

Comment les distinguer me direz vous ? Et bien, on utilise :

```
     bfd_boolean bfd_check_format(bfd *abfd, bfd_format format);

Verify if the file attached to the BFD ABFD is compatible with the
format FORMAT (i.e., one of ‘bfd_object’, ‘bfd_archive’ or ‘bfd_core’).

   The function returns ‘TRUE’ on success, otherwise ‘FALSE’ with one
of the following error codes:
```

Dans le cas d'une archive. Une archive est constituée de plusieurs fichiers objets et l'on dispose d'une fonction permettant de passer de fichier objet en fichier objet :

```
     bfd *bfd_openr_next_archived_file(bfd *archive, bfd *previous);

Provided a BFD, ARCHIVE, containing an archive and NULL, open an input
BFD on the first contained element and returns that.  Subsequent calls
should pass the archive and the previous return value to return a
created BFD to the next contained element. NULL is returned when there
are no more.
```

Pour chaque fichier objet constituant notre archive, on peut donc obtenir une structure bfd le décrivant.

2 Manipuler la table des symboles d'un fichier objet

Le type symbol_info. Pour commencer précisons que symbol_info est un type prédéfini dans bfd.h constitué par une structure :

```
typedef struct _symbol_info{
  symvalue value;
  char type;
  const char *name;            /* Symbol name.  */
  unsigned char stab_type;     /* Stab type.  */
  char stab_other;             /* Stab other.  */
  short stab_desc;             /* Stab desc.  */
  const char *stab_name;       /* String for stab type.  */
} symbol_info ;
```

À chaque symbole nous allons devoir associer une telle structure. En effet, les informations que l'on cherche sont incluse dans celle-ci. Par exemple, le type est un caractère qui est U si le symbol n'est pas défini et une majuscule si le symbol est global.

Comment obtenir un objet symbol_info ? Pour effectuer cette opération, nous allons devoir passer par deux objets intermédiaires :

- les “mini symboles” ;
- et “les asymbol”.

D’un objet bfd aux “mini symboles”. La fonction dont voici un exemple d’utilisation :

```
bfd *abfd ;
long symcount;
void *minisyms;
unsigned int size;
symcount = bfd_read_minisymbols (abfd, false, &minisyms, &size);
```

permet d’obtenir en retour un pointeur minisyms sur de l’espace mémoire contenant des informations désirées, le nombre symcount de symboles définis et la taille size de chaque espace mémoire représentant 1 symbol.

Le codage “mini symboles” n’est pas une structure mais de l’espace mémoire brut qu’il va falloir exploiter.

D’une “mini symboles” aux asymbols. À partir d’une structure bfd, on peut construire une structure du type asymbol :

```
  asymbol * bfd_make_empty_symbol (bfd *);

Create a new ‘asymbol’ structure for the BFD ABFD and return a pointer
to it.  Used by core file routines, binary back-end and anywhere else
where no private info is needed.
```

Cette structure va nous permettre d’extraire des informations de l’espace mémoire associé aux “mini symboles”. Pour ce faire, on utilise la fonction bfd_minisymbol_to_symbol :

```
bfd *abfd
asymbol *store, *sym;
bfd_byte *from;

    /* initialisation de store */
    store = bfd_make_empty_symbol (abfd);

    /* pour indiquer le point de d\’epart */
    from = (bfd_byte *) minisyms;

    sym = bfd_minisymbol_to_symbol (abfd, false, from, store);
```

La structure asymbol est associée à un unique symbole alors que les “mini symboles” contiennent l’information associé à l’ensemble des symboles.

Le pointeur store nous permet de stocker l’information associée au premier symbole dans la variable sym. Si on voulait accéder au second symbole, on utiliserait from += size ; avec size la taille déterminée dans le paragraph précédent.

D’une structure asymbol à une structure symbol_info. Pour compléter notre parcours, il nous reste à définir la fonction :

```
 bfd_get_symbol_info (bfd *, asymbol *, symbol_info *);
```

Cette dernière permet à partir d’une structure asymbol d’obtenir la structure symbol_info correspondante que nous avons vue être facile à utiliser.