R 2.04

2023 - 2024

Communication et fonctionnement bas niveau

TD N°3

« Pointeurs, paramètres, structures, E/S, alignement et gdb »

ANNE Jean-François
D'après le TD de F. BOURDON

Le but de ce TD est de se familiariser avec l'architecture bas niveau système et réseau.

« Pointeurs, paramètres, structures, E/S, alignement et gdb »

Notions vues dans ce TD :

Paramètres du programme main, structures et alignement en mémoire, accès disque, gdb.

Nombre de séance de 2h00 prévu pour faire ce TD : 2.

Prochain TD : Pointeurs et passage des paramètres par valeur.

PS : Les parties correspondant à du travail à faire sont toutes en italiques ; le restant étant du complément au cours.

1°). Affectation/assignation en « C » (rappels)

Un « objet » est défini par quatre éléments :

- son **nom** : identifiant externe
- son **type** : caractérise sa forme, ici la place mémoire nécessaire
- son **adresse** : sa localisation dans l'espace mémoire
- sa **valeur** : son contenu variable

L'affectation/assignation revient à résoudre deux expressions dont la première, située à gauche du signe égal (expression1), calcule l'adresse d'un objet « contenant », alors que la seconde, située à droite du signe égal (expression2), calcule un contenu.

expression1 = expression2

Dans le langage « C » la transmission des paramètres aux fonctions lors des appels à ces dernières, est faite par valeur, c'est-à-dire que l'on ne transmet à la fonction visée que la partie « valeur » de l'objet utilisé lors de cet appel. C'est pourquoi la fonction se charge de construire un objet « contenant » dans son propre espace d'adressage (pile), pour travailler sur la valeur transmise par le programme appelant.

On retrouve ici exactement le même mécanisme que celui de l'affectation/assignation tel que nous venons de le décrire. Dans l'appel à une fonction « *expression1* » (adresse de l'objet contenant) est défini par le paramètre formel de la fonction, alors que « *expression2* » est la valeur transmise par le programme appelant dans l'appel à la fonction.

Cette technique de passage des paramètres par valeur implique que l'information transmise par le programme appelant ne peut être modifiée par la fonction qui utilise ses propres objets. En effet la fonction ne connaît pas l'adresse (localisation dans la mémoire) de l'objet correspondant. Autrement dit, les paramètres d'une fonction sont des **paramètres d'entrée** pour la fonction (non modifiables par la fonction). Il existe une solution pour contourner ce problème lorsque l'on souhaite que la fonction agisse sur l'objet dont on ne lui transmet a priori que la valeur. Au lieu de transmettre la valeur de l'objet à la fonction visée, il suffit de lui transmettre l'adresse de cet objet. Dans ce cas on dira que nous avons à faire à un paramètre d'**entrée/sortie** dans la mesure où la fonction pourra accéder à l'objet et donc lire son contenu et le modifier.

Pour cela on va utiliser la notion de « pointeur » qui est un objet, au sens défini ci-dessus, c'est-à-dire qui

possède un nom, un type, une adresse et un contenu. Par exemple si l'on définit :

int * ptr_A;

On dispose d'un objet dont le nom est « ptr_A », le type « pointeur » (*) vers un objet de type entier (int), l'adresse (&ptr_A) et le contenu qui est l'adresse d'un autre objet de type entier (int).

Il faut noter que lorsque l'on utilise un pointeur comme paramètre d'une fonction, le mécanisme de transmission par valeur est respecté, dans la mesure où la fonction ne peut pas agir sur l'objet pointeur puisque seul son contenu (adresse d'un autre objet) lui est transmis.

En résumé, un paramètre d'une fonction peut être de trois natures différentes :

- **Paramètre d'entrée** : la valeur du paramètre est transmise à (entre dans) la fonction appelée, mais ne sera pas modifiée par la fonction (ex. « printf »), en retour de cette dernière (le paramètre « entre » dans la fonction) ;
- **Paramètre de sortie** : aucune valeur du paramètre n'est transmise à la fonction appelée (ex. « scanf »), par contre une valeur sera fournie au paramètre par la fonction, en retour de cette dernière (le paramètre « sort » de la fonction) ;
- **Paramètre d'entrée/sortie** : c'est la combinaison des deux premiers, une valeur du paramètre est transmise à (entre dans) la fonction appelée (ex. « fonction_qui_eleve_au_carre »), qui modifiera en sortie cette valeur (le paramètre « entre » et « sort » de la fonction).

La déclaration des pointeurs se fait en intercalant une étoile (*) entre le nom de l'objet et son type de base. La manipulation des pointeurs permet des indirections dans la mémoire. Nous disposons des caractères « * » et « & » pour manipuler les objets et leurs adresses à travers les pointeurs. En utilisant « l'* » avec un objet pointeur, cela nous permettra d'accéder au contenu de l'objet pointé, c'est-à-dire l'objet dont l'adresse est dans la partie contenu du pointeur. En utilisant « & » suivi du nom d'un objet quelconque (de type pointeur ou non), nous obtenons l'adresse de l'objet en question.

2°). ***Passage des paramètres à une fonction :***

- ***Exercice n°1 :***

Soit la fonction *fonc1* suivante :

```
int fonc1 ( int x ) {
int y = 0 ;
x = x + 1 ;
y = y + 1 ;
printf ( " fonction fonc1 , x = %d,\t y = %d\n", x, y ) ;
return y ;
}
```

✍ *Ecrivez un programme principal qui fait appel 5 fois (utiliser l'instruction for) à la fonction « fonc1 », avec comme paramètre l'indice de boucle, et qui affiche à l'écran à chaque itération les informations suivantes :*

- *Le résultat renvoyé par cette fonction ;*
- *La valeur du compteur d'itérations ;*

- *Ainsi que la valeur du paramètre d'appel (x).*

*Que se passe-t-il si l'on met « **static int y = 0 ;** » à la place de l'instruction « **int y = 0 ;** » ?*

Exemple de résultat attendu de l'exécution du programme sans static :

```
Le résultat de l'exécution est le suivant :
prompt> fonc1
fonction fonc1, x=2, y=1
resultat de fonc1 : 1 compteur : 0 valeur argument : 1
fonction fonc1, x=3, y=1
resultat de fonc1 : 1 compteur : 1 valeur argument : 2
fonction fonc1, x=4, y=1
resultat de fonc1 : 1 compteur : 2 valeur argument : 3
fonction fonc1, x=5, y=1
resultat de fonc1 : 1 compteur : 3 valeur argument : 4
fonction fonc1, x=6, y=1
resultat de fonc1 : 1 compteur : 4 valeur argument : 5
prompt>
```

3°). Rappels sur la fonction « main ( ) »

Lorsqu'on lance une commande externe (binaire) dans un Shell, on peut en plus du nom de la commande, ajouter des paramètres/options sous la forme de mots séparés par des blancs (séparateur classique des Shells). Certains de ces mots sont destinés au Shell lui-même avant le lancement de la commande, les autres doivent être récupérés par la commande, sur le même principe que les paramètres pour un appel de fonction classique (dans le programme principal ou dans une autre fonction).

La différence avec l'appel de fonction classique réside principalement dans le fait que les paramètres formels (de la fonction « main ») sont fixés (type et nom) par le langage : « int argc » et « char* argv [ ] » (ou encore « char ** argv », ce qui est identique).

Prenons un exemple : *prompt*> ls -l main.c

Cette commande affiche le détail des informations concernant le fichier « main.c ». Afin que le binaire « /bin/ls » prenne en compte les deux arguments « -l » et « main.c », il suffira de construire le programme principal « main » du programme « ls » en déclarant les paramètres formels « argc » et « argv » de la façon imposée suivante :

int main (int argc, char* argv[ ]) { ... }

Ou encore (c'est équivalent) :

int main (int argc, char ** argv) { ... }

Il suffit de comprendre comment les paramètres sont passés dans ces deux variables, pour pouvoir en disposer dans le programme correspondant.

- « argv » est un tableau de pointeurs vers des chaînes de caractères ;
- « argc » est le nombre d'éléments passés dans le tableau « argv ».

Dans notre exemple, « argc » vaut 3, un pour « ls », un pour « -l » et un pour « main.c ».

« argv[0] » pointe vers la chaîne « ls », « argv[1] » vers la chaîne « -l » et « argv[2] » vers la chaîne

« main.c ».

- ***Exercice n°2 :***

  ✍ *Dessinez la représentation en mémoire du passage des paramètres dans cet exemple.*

  ✍ *Ecrivez un programme qui prend un paramètre (le nom de la personne qui a écrit le programme) et dont le comportement consiste à afficher « Mon nom est ... », en utilisant le paramètre passé à l'appel. Ce programme doit vérifier que l'appel a été fait avec le bon nombre de paramètre (ici 1). Testez ce programme.*

  ✍ *Ecrivez un programme qui peut être lancé avec un nombre variable d'arguments. Pour chacun d'eux le programme affichera le numéro de l'argument et l'argument lui-même. Utiliser ce programme avec « *.c » dans un répertoire où il y a plusieurs fichiers « c ». Qu'en conclure sur la limitation théorique du nombre d'arguments passés à une commande.*

  ✍ *Lancez votre programme en passant les arguments suivants, qu'en concluez-vous ?*

  prompt> **mon_prog a bc "def ghi" \"jkl mno\"**

4°). La fonction « main ( ) » (suite)

Comme nous l'avons vu au TD sur la chaîne de compilation, les variables d'environnement sont définies sous la forme de chaînes de caractères contenant des affectations du type NOM=VALEUR.

Ces variables accessibles en « C » ou en shell peuvent donc être modifiées par programme. En lançant un binaire écrit en « C », l'ensemble des variables de l'environnement du processus résultant est copié automatiquement dans un tableau de chaînes de caractères accessible par le processus. Ce tableau est disponible dans la variable globale « environ », qu'il faudra déclarer ainsi en début de fichier source « **extern char** environ ; ».** Ce tableau contient des chaînes de caractères terminées par un caractère nul, et se finit lui-même par un pointeur nul.

Ce tableau de variables d'environnement avec leur valeurs associées est également fourni comme troisième argument à la fonction « main » :

int main (int argc, char argv, char** envp); { … }**

Bien que cette méthode fonctionne, elle n'est pas normalisée ; il est donc préférable d'utiliser directement la variable globale « environ ».

- ***Exercice n°3 :***

  ✍ *Ecrivez un programme qui affiche les variables d'environnement du processus courant en utilisant ce troisième argument envp du programme principal. Pour cela vous utiliserez la structure itérative « while », en utilisant l'arithmétique sur les pointeurs. Vous afficherez le numéro de l'itération en cours (depuis 0) en début de chaque ligne, suivi de « NOM=valeur » de chaque variable d'environnement (une ligne par variable). Expliquez pourquoi il est nécessaire d'utiliser un pointeur intermédiaire et non directement « envp ».*

5°). Manipulation des informations contenues dans un fichier sur

disque

- ***Exercice n°4 :***

  ✍ *Complétez le programme « gest_fich.c » suivant afin qu'il ait le comportement proposé. Vous utiliserez les fonctions d'accès au disque dont les interfaces sont décrites ci-dessous. Compilez le fichier et exécutez-le afin de vous assurer qu'il fonctionne conformément aux spécifications.*

```
#include <stdio.h>
#define MAXPERS 3
#define TAILLE_NOM 15
#define TAILLE_ADRESSE 25

typedef struct personne {
char nom[TAILLE_NOM];
int age;
char adresse[TAILLE_ADRESSE];
} Personne;
```

- Saisie des informations pour les MAXPERS personnes */

```
void saisiePersonnes(Personne *p) { ... }
```

- Sauvegarde des informations saisies dans le fichier « personne.data » */

```
void sauvegardePersonnes(Personne *p){ ... }
```

- Lecture du fichier « personne.data » entier */

```
void lecturePersonnes(Personne *p){ ... }
```

- Lecture d'un enregistrement particulier (num) du fichier « personne.data » */

```
void lireUnePersonne(Personne *p,int num){ ... }
```

/* *Fonctionnement du programme :*

- *S'il n'y a pas d'arguments, le programme saisit les personnes et les sauvegarde dans le fichier avant de se terminer ;*
- *Si l'argument est 'a', le programme lit le fichier des personnes, l'affiche et se termine ; si le fichier « personne.data » n'existe pas, affichez un message d'erreur ;*
- *Si l'argument est un chiffre entre 0 et MAXPERS, lire sur disque l'enregistrement correspondant et l'afficher, sinon erreur.* */

```
int main(int argc, char *argv[])
{
Personne pers[MAXPERS];
int i;
if(argc == 1) { /* pas d'arguments */
```

```
/* saisie et enregistrement des personnes */
... }
else {
if(strcmp(argv[1],"a") == 0) { /* lire tous les éléments */
... }
else { /* lire l'élément « n » indique en argument */
… }}}
```

Pour écrire le programme « gest_fich.c » de la question N°7, vous utiliserez les primitives de manipulation des fichiers *fopen*, *fclose*, *fread*, *fwrite* et *fseek* dont les signatures sont les suivantes :

FILE *fopen (const char *filename, const char *type)

Où *type* est une chaîne de caractères correspondant à l'une des valeurs suivantes :

- *r* ouverture pour lecture
- w troncature ou création pour écriture
- a concaténation ; ouverture pour écriture en fin de fichier, ou création pour écriture
- r+ ouverture pour mise à jour (lecture/écriture)
- w+ troncature ou création pour mise à jour
- a+ concaténation ; ouverture ou création pour mise à jour en fin de fichier

```
int fread (void *ptr, sizee_t size, int nitems, FILE *stream)
int fwrite (void *ptr, sizee_t size, int nitems, FILE *stream)
int fclose (FILE *stream)
```

Où *ptr* est l'adresse de la zone mémoire où vont être placées les données lues sur le disque (*read*), ou bien la zone mémoire dans laquelle ont été préparées les données à écrire sur le disque (*write*) ; *size* correspond à la taille des données en octet à lire/écrire sur le disque ; *nitems* permet de minimiser le nombre d'appels à ces fonctions (read/write) ; *stream* correspond au flux depuis ou vers lequel vont se faire les opérations de lecture/écriture avec le disque.

```
int fseek (FILE *stream, long offset, int ptrname)
```

Où *ptrname* est un entier avec l'une des valeurs suivantes (définie dans « unistd.h ») :

- SEEK_CUR déplacement à partir de la position courante augmenté de l'offset
- SEEK_END déplacement à partir de la fin de fichier augmenté de l'offset
- SEEK_SET déplacement à partir du début du fichier augmenté de l'offset

---

6°). L'alignement en mémoire :

A partir de l'exemple suivant, nous allons regarder en quoi consiste le problème de l'alignement des variables (en particulier les structures) en mémoire opéré par le compilateur « C ».

/* on peut former les expressions suivantes :

- &enr : adresse de la structure « enr » ;
- &enr.nom[0] : adresse du premier caractère du champ « nom » ;
- &enr.nom[TNOM-1] : adresse du dernier caractère du champ « nom » ;
- &enr.reference : adresse du champ « reference ».

On peut également utiliser la macro-fonction « offsetof » définie dans l'entête <stddef.h>, pour connaître le décalage entre un champ d'une structure et le début de cette structure. Par exemple le décalage du champ « reference » est donné par l'expression :

```
offsetof (struct enregistrement, reference)
```

- ***Exercice n°5***

✍ *Complétez les « ... » dans le programme suivant, puis exécutez-le :*

```
# include <stdlib.h>
# include <stdio.h>
# include <stddef.h>
# define TNOM 10
typedef struct enregistrement { char nom[TNOM]; long reference; }
Enregistrement;
int main (void) {
Enregistrement enr;
printf (" Adresse de \n ");
printf (" - enr : %p \n ", ... );
printf (" - enr.nom[0] : %p (décalage : %d) \n ", ... , offsetof(... , ...)
);
printf (" - enr.nom[TNOM-1] : %p \n ", ... );
printf (" - enr.reference : %p (décalage : %d) \n ", ... , offsetof(... ,
... ) );
return EXIT_SUCCESS;
}
```

✍ *Observez le réalignement effectué par le compilateur. Comment cela se manifeste-t-il ?*

✍ *Modifiez le programme « gest_fich.c » pour faire apparaître ce problème en utilisant les mêmes instructions que dans cet exemple.*

7°). Exercices complémentaires et facultatifs sur les pointeurs

- ***Exercice n°6***

Soit le programme « swapNaif.c » suivant. Compilez ce programme et lancez-le. Expliquez pourquoi la

```
fonction « swap » de fonctionne pas vraiment.
prompt> cat swapNaif.c
#include <stdio.h>
void swap(int x, int y) {
int z;
z=x;
x=y;
y=z;
printf("valeur après permutation dans la fonction swap :\n");
printf("x = %d\t y = %d\n",x,y);
}
void main(){
int x,y;
printf("Entrez deux valeurs pour la permutation\nx = ");
scanf("%d",&x);
printf("y = ");
scanf("%d",&y);
swap(x,y);
printf("valeur après l'appel à la fonction swap :\n");
printf("x = %d\t y = %d\n",x,y);
}
prompt>
```

Voici un exemple de programme qui utilise plusieurs paramètres de sortie. Après avoir corrigé le code de la fonction « swap » ci-dessus, complétez (??) le code « milieu.c », avant de le compiler et de le lancer.

```
prompt> cat milieu.c
#include <stdio.h>
void saisieXY(float* x,float* y) {
printf("Entrez le premier reel : ");
scanf("%f", ??);
printf("Entrez le second reel : ");
scanf("%f", ??);
}
void milieu(float xp1,float yp1,float xp2,float yp2,float* xpM,float* ypM)
{
if(xp1>xp2) swap( ??, ??);
??=xp1+(xp2-xp1)/2;
```

```
if(yp1>yp2) swap( ??, ??);
??=yp1+(yp2-yp1)/2;
}
int main() {
float xP1,yP1,xP2,yP2,xM,yM;
printf("Entrez les coordonnees x,y du point P1 :\n");
saisieXY( ??, ??);
printf("Entrez les coordonnees x,y du point P2 :\n");
saisieXY( ??, ??);
milieu(xP1,yP1,xP2,yP2, ??, ??);
printf("Les coordonnees du point milieu sont :\n");
printf("xM = %f\t yM = %f\n",xM,yM);
}
prompt>
```

- ***Exercice n°7***

Dans l'exemple ci-dessous (pointer_types.c) nous allons parcourir les deux tableaux (caractères et entiers) en utilisant des pointeurs, respectivement d'entier pour le tableau d'entiers et de caractère pour le tableau de caractères, et l'arithmétique sur les pointeurs.

```
prompt> cat pointer_types.c
#include <stdio.h>
int main() {
int i;
char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
int int_array[5] = {1, 2, 3, 4, 5};
char *char_pointer; // pointeur de caractère
int *int_pointer; // pointeur d'entier
char_pointer = char_array; // initialisation des pointeurs
int_pointer = int_array;
for(i=0; i < 5; i++) { // parcours du tableau d'entiers
printf("[integer pointer] points to %p, which contains the integer %d\n",
int_pointer, *int_pointer);
int_pointer = int_pointer + 1; // arithmétique sur les pointeurs, ici +4
}
for(i=0; i < 5; i++) { // parcours du tableau de caractères
printf("[char pointer] points to %p, which contains the char '%c'\n",
char_pointer, *char_pointer);
char_pointer = char_pointer + 1; // arithmétique sur les pointeurs, ici +1
}
}
prompt>
```

Compilez ce programme et lancez-le.

Dans l'exemple ci-dessous (pointer_types2.c) nous allons reprendre l'exemple précédent (pointer_types.c) mais en utilisant un pointeur de caractères pour parcourir le tableau d'entier et inversement un pointeur d'entier pour parcourir le tableau de caractères.

```
prompt> cat pointer_types2.c
#include <stdio.h>
int main() {
int i;
char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
int int_array[5] = {1, 2, 3, 4, 5};
char *char_pointer;
int *int_pointer;
char_pointer = int_array; // les pointeurs char_pointer et int_pointer
int_pointer = char_array; // pointent vers des types de données
incompatibles
for(i=0; i < 5; i++) { // parcours du tableau de caractères
printf("[integer pointer] points to %p, which contains the char '%c'\n",
int_pointer, *int_pointer);
int_pointer = int_pointer + 1; // arithmétique sur les pointeurs, ici +4
}
for(i=0; i < 5; i++) { // parcours du tableau d'entiers
printf("[char pointer] points to %p, which contains the integer %d\n",
char_pointer, *char_pointer);
char_pointer = char_pointer + 1; // arithmétique sur les pointeurs, ici +1
}
}
prompt>
```

Compilez ce programme avec l'option « -Wall ». Qu'observez-vous ? Lancez le programme et commentez les résultats affichés. Vous pouvez utiliser « gdb » pour manipuler les données lors de l'exécution du programme.

Faites une version « pointer_types3.c » en corrigeant la version « pointer_types2.c » de telle façon que le programme s'exécute correctement. Pour cela utilisez uniquement l'opérateur de « cast ».

Faites une version « pointer_types4.c » en corrigeant la version « pointer_types2.c » mais en ayant recours au type « void* » (type indéfini) permettant de n'utiliser qu'un seul pointeur et l'opérateur de « cast ».

Modifiez le code de « pointer_types4.c », mais cette fois en utilisant un entier à la place du pointeur de type « void ». Dans la mesure où un entier est codé sur quatre octets, comme le sont les pointeurs, le code va bien fonctionner. Il faudra juste faire artificiellement (appel à la fonction « sizeof ») l'équivalent de l'arithmétique sur les pointeurs à chaque tour de boucle.

8°). ***Utilisation du débogueur symbolique « gdb » pour désassembler du code C***

Cette partie consiste à utiliser l'outil gdb afin de voir comment le code écrit en langage C est

transformé en assembleur (désassemblage), où chaque instruction possède son équivalent en langage machine (binaire).

Rappels : Avant de pouvoir utiliser gdb il faut compiler votre programme avec l'option -g. Cela permet au compilateur de générer les informations nécessaires au débogage symbolique. Ensuite il suffit de lancer gdb avec le nom de votre programme en paramètre et l'option -q pour éviter des traces au lancement :

```
prompt> gcc firstprog.c -o prog_test -Wall -g
prompt> gdb firstprog -q
(gdb)
```

Le débogueur possède un interprète de commandes qui attend que vous lui donniez l'une de ses commandes à exécuter. Pour connaître ces commandes, vous pouvez utiliser le manuel (man). Le principe de fonctionnement de gdb est d'explorer l'exécution d'un programme en l'arrêtant (commande b) sur des points d'arrêt (break point), afin de pouvoir consulter le contenu des variables, modifier le contenu de ces variables ou encore voir l'état de la pile (stack), correspondant aux différents appels successifs à des fonctions. On peut faire avancer cette exécution ligne par ligne (step/next). Mettre des points d'arrêt (break) fait partie des commandes disponibles dans gdb.

Par défaut gdb désassemble les fichiers écrits en langage C dans la syntaxe att. Il existe la possibilité d'utiliser la syntaxe intel (plus lisible). Pour cela on peut soit utiliser au coup par coup la commande set disassembly-flavor de gdb, soit une fois pour toute en créant un fichier de configuration ~/.gdbinit.

- ***Exercice n°8***

Faites les manipulations suivantes sur votre compte afin d'utiliser le mode « intel » pour l'affichage de l'assembleur.

```
prompt> gdb -q // lancement du débogueur
(gdb) show disassembly-flavor // affichage de la syntaxe utilisée par défaut (att)
The disassembly flavor is "att".
(gdb) set disassembly-flavor intel // modification de cette syntaxe (intel)
(gdb) show disassembly-flavor
The disassembly flavor is "intel".
(gdb) quit
prompt> echo "set disassembly-flavor intel" > ~/.gdbinit
prompt> cat ~/.gdbinit // affichage du contenu du fichier ~/.gdbinit.
set disassembly-flavor intel
prompt> gdb -q
(gdb) show disassembly-flavor // vérification de la persistance du choix de la syntaxe
The disassembly flavor is "intel".
(gdb) q
prompt>
```

La commande x examine le contenu de la mémoire des registres (rip par exemple, pour l'instruction en cours d'exécution) où le contenu de la mémoire pointée par le contenu des registres (équivalent

*pointeur). On peut également examiner les registres pour y lire directement les instructions vers où ils pointent. Pour cela nous allons utiliser la commande x/i. x/i $rip donnera la première instruction (à exécuter) assembleur à l'adresse pointée par rip, x/3i $rip fournira les trois instructions consécutives placées à partir de l'adresse contenue dans rip. La commande x est équivalente à l'usage de l'opérateur « * » de dé référenciation d'un pointeur en C. Il faut préciser qu'un mot mémoire par défaut est codé sur 4 octets (32 bits). En hexadécimal il faut deux caractères pour coder un octet. L'affichage d'un mot en hexadécimal prendra donc 8 caractères :

```
x/x $rip → 0x00fc45c7 // contenu en hexadécimal
x/u $rip → 16532935 // contenu en réel non signé
x/4xb $rip → 0xc7 0x45 0xfc 0x00 // b comme mot simple sur un octet
// (2 caractères) en hexa
x/4ub $rip → 199 69 252 0 // idem en décimal
prompt> cat firstprog.c
#include <stdio.h>
int main() {
int i;
for(i=0;i<10;i++) printf("Hello World!\n");
}
prompt>
```

Compilez le programme « firstprog.c » avec l'option « -g ». Lancez « gdb » sur « firstprog », listez le contenu du source C et désassemblez la fonction « main » en utilisant la commande « disassemble main ». Retrouvez la première instruction effective du code. Que fait-elle ? Positionnez un point d'arrêt sur cette première instruction et lancez l'exécution.

Le registre « rip » contient l'adresse de l'instruction courante à exécuter. En faisant « info register rip » vous pouvez visualiser le contenu de ce registre « rip ». Vous pouvez visualisez le contenu de ce registre, qui est une adresse, au format suivant : « x/x $rip ». Ce format correspond à un mot de quatre octets (format par défaut des adresses), représenté sous la forme de huit caractères (deux caractères par octet).

Vous pouvez visualiser le contenu de ce registre « rip » au format suivant : « x/4xb $rip ». A quoi correspond ce format ? Dessinez la mémoire avec le registre « rip », ce qu'il contient et vers où il pointe.

On peut également visualiser les instructions en assembleur (équivalentes du binaire) pointées par le registre « rip ». Faites le en utilisant la commande « x/i $rip ». Lancez l'exécution de votre programme en utilisant la commande « nexti » qui vous permet d'avancer à l'instruction suivante. Vous allez faire plusieurs « nexti » de suite jusqu'à ce que le résultat de la commande suivante « x/4xb $rbp - 4 » valle 1 (0x01 0x00 0x00 0x00). Ecrivez la suite des instructions réalisées par GDB pour arriver à cet état. Pour cela, à chaque pas d'exécution vous désassemblerez la fonction « main » et vous lancerez la commande « x/4xb $rbp - 4 ». « $rbp » affiche le contenu du registre « rbp », qui est une adresse du début de frame sur la pile. « $rbp - 4 » correspond à cette adresse moins quatre octets. Vous pouvez faire un dessin de la mémoire avec le registre « rbp » et la zone de mémoire pointée avec son contenu dans l'état courant du processus.

- ***Exercice n°9***

Nous allons regarder le fonctionnement de la pile lors de l'appel de fonctions dans un programme

principal (main).

Le traitement d'un appel de fonction inclut les étapes suivantes :

1. l'empilement des paramètres de la fonction sur la pile d'exécution (avec l'instruction push), ou directement dans les registres (edi/rdi, esi/rsi, edx/rdx, r8, r9) ;

2. l'appel de la fonction (avec l'instruction call) ; cette étape déclenche la sauvegarde de l'adresse de retour de la fonction sur la pile d'exécution ;

3. le début de la fonction qui inclut :

    a. la sauvegarde de l'adresse de la pile qui marque le début de l'enregistrement de l'état actuel du programme,

    b. l'allocation des variables locales dans la pile d'exécution ;

4. l'exécution de la fonction ;

5. la sortie de la fonction qui inclut la restauration du pointeur qui marquait le début de l'enregistrement de l'état du programme au moment de l'appel de la fonction,

6. l'exécution de l'instruction ret qui indique la fin de la fonction et déclenche la récupération de l'adresse de retour et le branchement à cette adresse.

Voici un exemple simple de programme en C qui fait appel à une fonction :

```
prompt> cat foo.c
void foo(int i, int j){
int a = 1;
int b = 2;
return;
}
int main(){
foo(5,6);
}
prompt>
```

A la fin d'un appel à une fonction, l'exécution doit se poursuivre avec l'instruction qui suit cet appel. Il faut mémoriser l'adresse de retour en utilisant l'adressage indirect. Pour cela nous disposons des instructions suivantes :

call op // saut inconditionnel à op, et empile l'adresse de l'instruction suivante

ret // dépile une adresse et saute à cette adresse

Le passage des paramètres à une fonction peut se faire directement sur la pile ou via des registres (dans le cas de peu de paramètres). Quand il y a peu de paramètres (les six premiers paramètres entiers sont passés dans rdi, rsi, rdx, rcx, r8 et r9, les autres par la pile), on les stocke dans les registres avant le call. Si la fonction doit modifier le paramètre (passage par référence), il faut passer l'adresse de la donnée à modifier. Dans ce cas les paramètres ne sont pas dépilés par la fonction, mais on y accède directement par la pile. L'adresse de retour empilée par call doit être dépilée en premier.

Les paramètres sont empilés dans l'ordre inverse de celui dans lequel ils sont écrits dans la fonction C.

Pour accéder aux paramètres passés par la pile on utilise le registre rbp.

La fonction doit empiler rbp (son contenu qui est le paramètre à mettre sur la pile) avec push rbp puis copier rsp (son contenu qui pointe sur le haut de la pile mis à jour par push) dans rbp avec l'instruction move rbp,rsp. Ces deux instructions ont permis de mettre l'ancien contenu de rbp sur la pile et de faire pointer rbp sur ce contenu, mais cette fois sur la pile. Dès que l'on manipule la pile (add, sub, push, pop, call), rsp (haut de pile) est mis à jour. La fonction add diminue la stackframe (vers le haut), sub agrandit la stackframe vers le bas, push empile et augmente la pile (vers le bas), pop dépile et diminue la pile vers le haut, call empile une adresse de retour après l'appel à la fonction courante (stackframe).

Pour accéder aux paramètres stockés sur la pile en utilisant rsp on ajoute le nombre d'octets nécessaires ([rsp+16], [rsp+24]...). C'est l'inverse en utilisant rbp ([rbp-16], [rbp-24]...)

A la fin d'une fonction on libère l'espace mémoire correspondant avec mov rsp,rbp. A la fin de l'appel, la fonction doit restaurer l'ancienne valeur de rbp dans rbp, qui était stockée sur la pile, puis diminuer la pile, le tout avec pop rbp.

Lancez « foo » dans « gdb » et désassembler le « main ». Les registres « esi » et « edi » permettent de stocker des valeurs entières sans les passer par la pile. Regardez ce que fait chaque instruction du « main ».

Désassembler la fonction « foo ». En vous aidant du code assembleur, dessinez la pile avec ses variables et registres. Le registre « rbp » pointe sur l'avant-dernière adresse du haut de la pile, correspondant au « frame » précédent.

Modifiez « foo.c » en créant un fichier « fooBis.c » dans lequel vous ajouterez les instructions « i=i+a ; » et « j=j+b ; » à la fonction « foo ». Compilez le fichier avec l'option « -g » de « gdb », puis lancez « gdb » sur ce binaire exécutable. Enfin, dans « gdb » désassemblez « main » et « foo ». Déduisez du résultat où se trouvent les variables « a », « b », « i » et « j » sur la pile.

Modifiez à nouveau « fooBis.c » en « fooTer.c » de telle façon que la fonction « foo » appelle une fonction « foo2 », qui ne fait rien d'autre qu'un « return 0 ; », à laquelle elle aura passé « i » et « j » en paramètre. Examiner le code assembleur du « main », de « foo » et de « foo2 ». Observez la gestion des paramètres et du sommet de pile dans les codes générés. Regardez la différence entre le code de « foo » sans appel à « foo2 » (« fooBis.c »), d'avec celui de « foo » qui appelle « foo2 » (« fooTer.c »).

Le registre eax permet de gérer la valeur renvoyée par un return.

I. Webographie :

- https://bourdon.users.info.unicaen.fr/cours/IUT-1A/index.html#exercices
-