Logiciel de Base

Examen pratique
1A par alternance
Année 2015–2016
Session normale

Durée : 3h00
Ce document contient 7 pages.

Consignes générales :

Le barème donné est indicatif. Les exercices sont indépendants et peuvent être traités dans le désordre. **Il est recommandé d'essayer de traiter toutes les questions**, même si vous ne les terminez pas.

Les documents sont interdits, sauf une feuille A4 manuscrite recto-verso. Photocopies et documents imprimés interdits. Tout appareil électronique (*e.g.* PC, téléphones, clés USB, *etc.*) interdit.

Le sous-répertoire `docs` contient les documents PDF supports du cours, ainsi que les documentations de l'architecture Intel.

Le code C que vous écrirez devra être correctement présenté et commenté. La clarté du code et le respect des conventions d'écriture (*coding-style*) seront pris en compte dans la notation.

Dans tous les exercices portant sur l'assembleur Intel, on demande de traduire **systématiquement** du code C en assembleur, comme le ferait un compilateur. Vous ne devez donc pas chercher à optimiser le code écrit et **vous devez placer et lire systématiquement les variables locales dans la pile d'exécution comme vu en TP**, sauf indication contraire dans les questions.

Pour chaque ligne de C traduite en assembleur, vous recopierez en commentaire la ligne en question avant d'implanter les instructions assembleur correspondantes. Vous indiquerez aussi la position par rapport au registre `%rbp` (ou `%ebp` pour le code 32 bits) de chaque variable locale, ainsi que les registres ou les adresses dans la pile contenant les paramètres au début des fonctions implantées. Tous les commentaires additionnels sont bienvenus.

Tout le travail demandé est à rendre dans les fichiers fournis ou que l'on vous demande explicitement de créer : le correcteur ne regardera pas les autres fichiers que vous pourriez ajouter. On fournit des tests basiques pour vous aider à mettre au point vos programmes. On ne vous demande pas d'en ajouter, mais vous pouvez modifier les fichiers distribués pour ajouter vos propres tests si besoin.

Pour compiler vos programmes, vous utiliserez le `Makefile` fourni. Si vous voulez nettoyer le répertoire pour tout recompiler à partir de zéro, il suffit de taper la commande `make clean`. L'utilisation de `gdb` et `valgrind` est très fortement recommandée pour la mise au point de vos programmes.

A la fin de l'épreuve, vous devez fermer proprement votre session en cliquant sur l'icône « Sauvegarder et terminer l'examen ». Une fois déconnecté, vous ne pourrez plus vous reconnecter et votre code sera rendu automatiquement. Attention, vous ne devez surtout pas vous déconnecter via le menu CentOS, au risque de perdre votre travail. On recommande de sauvegarder régulièrement votre travail grâce à l'icône « Sauvegarder l'examen sans quitter » présente sur le bureau.

Ex. 1 : Exercice préliminaire *(1 pt)*

Dans cet exercice particulièrement difficile, on vous demande de compléter le fichier `identite.txt` avec votre nom, votre prénom et le nom de la machine sur laquelle vous composez cet examen.

Ex. 2 : Ecriture d'un programme principal *(4 pts)*

La fonction de cet exercice doit être implantée en **assembleur 64 bits** (`x86_64`) dans un fichier que vous appellerez `main.s`. Pour compiler le programme, vous utiliserez la commande `make main`.

On va écrire un programme principal directement en assembleur, ce qui ne change rien en pratique par rapport à une fonction classique.

Question 1 Implanter la fonction `main` : cette fonction est à écrire directement en assembleur dans le fichier `main.s` (il n'y a pas de fichier C rattaché pour cet exercice).

La fonction doit être la traduction du code C suivant :

```
int32_t main(int32_t argc, char *argv[])
{
    for (int32_t i = 0; i < argc; i++) {
        puts(argv[i]);
    }
    return 0;
}
```

On rappelle le sens des paramètres de la fonction `main` :

- `argc` est un entier signé (par convention) contenant le nombre d'arguments passés sur la ligne de commande ;
- `argv` est un tableau de chaînes de caractères contenant les paramètres passés sur la ligne de commande ;
- par convention, le premier paramètre est toujours le nom du programme lui-même.

Par exemple, si on lance le programme par la commande `./main toto titi tutu`, le programme affichera :

```
./main
toto
titi
tutu
```

Ex. 3 : Manipulation de matrices *(8 pts)*

Les fonctions de cet exercice doivent être implantées en **assembleur 32 bits** (`x86_32`) dans un fichier que vous appellerez `fct_matrice.s`. Pour compiler le programme, vous utiliserez la commande `make matrice`.

On rappelle que `valgrind` ne connait pas l'instruction `enter` en mode 32 bits, il faudra donc utiliser à sa place la suite d'instructions suivante, équivalente à l'instruction `enter $N, $0` :

```
pushl %ebp
movl %esp, %ebp
subl $N, %esp
```

On va travailler dans cet exercice sur des matrices (*i.e.* des tableaux à deux dimensions) d'entiers signés sur 16 bits. Une matrice sera donc définie par un `int16_t tab[nbr_lig][nbr_col]`, où `nbr_lig` et `nbr_col` sont des entiers 32 bits non-signés représentant respectivement le nombre de lignes et de colonnes de la matrice. Ces entiers sont définis comme des variables globales dans le fichier `matrice.c`.

On fourni dans le fichier `matrice.c` une fonction d'affichage d'une matrice ainsi qu'un programme de test minimal appelant les fonctions à implanter.

Question 1 Implanter la fonction `ecrire` : cette fonction doit écrire une valeur dans une case du tableau identifiée par ses coordonnées.

La fonction doit être la traduction du code C suivant :

```
void ecrire(int16_t mat[nbr_lig][nbr_col], uint32_t lig, uint32_t col, int16_t val)
{
    mat[lig][col] = val;
}
```

La fonction `ecrire` sera une fonction privée du fichier `fct_matrice.s`.

Notez bien que `nbr_lig` et `nbr_col` ne sont pas des paramètres de cette fonction : il s'agit de variables globales définies dans le fichier `matrice.c`.

Vous aurez besoin de l'instruction de multiplication pour implanter cette fonction. On fourni la documentation officielle de cette instruction dans le répertoire `docs`, et on rappelle brièvement son fonctionnement pour des entiers 32 bits :

- le premier opérande de la multiplication doit être placé dans le registre `%eax` ;
- le deuxième opérande peut-être soit un registre soit une adresse mémoire (mais pas une constante), et doit être précisé comme argument de l'instruction `mul` ;
- le résultat de la multiplication (sur 64 bits) sera stocké dans les registres `%edx:%eax` (partie haute dans `%edx` et partie basse dans `%eax`).

Question 2 Implanter la fonction `init` : cette fonction doit remplir la matrice avec des valeurs aléatoires.

La fonction doit être la traduction du code C suivant :

```
void init(int16_t mat[nbr_lig][nbr_col])
{
    srandom(time(NULL));
    for (uint32_t lig = 0; lig < nbr_lig; lig++) {
        for (uint32_t col = 0; col < nbr_col; col++) {
            ecrire(mat, lig, col, random() % 19 - 9);
        }
    }
}
```

On rappelle que :

- la fonction `time` renvoie un entier 32 bits correspondant au nombre de secondes écoulées depuis le 1er janvier 1970 : cette valeur nous sert de base pour initialiser le générateur de nombres aléatoires ;
- la fonction `srandom` sert à initialiser le générateur de nombres aléatoires : elle prend en argument un entier 32 bits qui sert de base, et ne renvoie rien ;
- la fonction `random` renvoie un entier 32 bits aléatoire compris entre 0 et $2^{31} - 1$.

Le bout de code `random() % 19 - 9` permet de garantir que les valeurs affectées dans la matrice seront en pratique comprises entre -9 et 9 inclus (juste pour faciliter la lecture).

On rappelle aussi que la constante `NULL` vaut en fait simplement 0.

Pour implanter cette fonction, vous aurez besoin de l'instruction de division. On fourni la documentation officielle de cette instruction dans le répertoire `docs`, et on rappelle brièvement son fonctionnement pour des entiers 32 bits :

- le dividende (*i.e.* ce qu'on veut diviser, sur 64 bits) doit être placé dans la combinaison de registres `%edx:%eax` (partie haute dans `%edx` et partie basse dans `%eax`) ;
- le diviseur (*i.e.* ce qui divise, sur 32 bits ici) peut-être soit un registre soit une adresse mémoire (mais pas une constante), et doit être précisé comme argument de l'instruction `div` ;
- le résultat de la division sera placé dans les registres `%eax` (quotient de la division) et `%edx` (reste de la division).

Question 3 Implanter la fonction `somme` : cette fonction doit renvoyer la somme des valeurs contenues dans la matrice.

La fonction doit être la traduction du code C suivant :

```
int16_t somme(int16_t *mat, uint32_t nbr_cases)
{
    int16_t som = 0;
    for (uint32_t ix = 0; ix < nbr_cases; ix++) {
        som += *(mat + ix);
    }
    return som;
}
```

Cette fonction prend en argument un pointeur vers la première case de la matrice et le nombre de cases (*i.e.* `nbr_lig` × `nbr_col`).

On note qu'on ne passe pas la matrice sous sa forme naturelle d'un tableau à deux dimensions, mais d'un pointeur vers le premier entier 16 bits de la matrice.

En effet, on rappelle qu'un tableau à deux dimensions est « aplati » en mémoire. Par exemple, la matrice suivante : $\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{pmatrix}$ sera stockée en mémoire sous la forme d'une zone à une dimension contenant : `11 12 13 14 21 22 23 24 31 32 33 34`.

On peut donc parcourir simplement la matrice en faisant avancer un déplacement (`ix` dans le code C) par rapport à l'adresse de la première case de la matrice.

Ex. 4 : Parcours d'arbres binaires de recherche *(7 pts)*

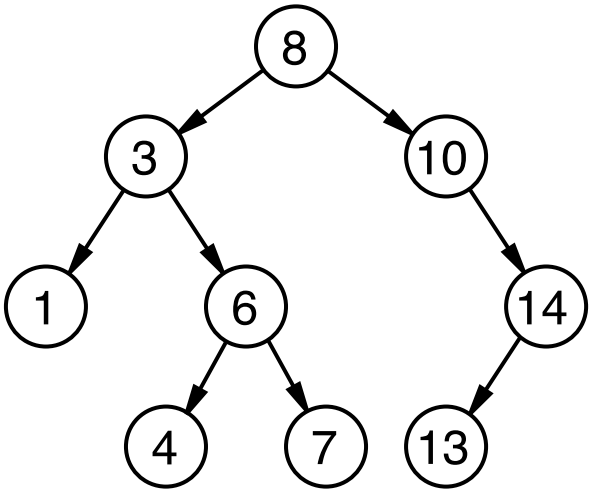
Les fonctions de cet exercice doivent être implantées en **assembleur 64 bits** (`x86_64`) dans un fichier que vous appellerez `fct_abr.s`. Pour compiler le programme, vous utiliserez la commande `make abr`.

On va travailler dans cet exercice sur des arbres binaires de recherche (ABR) dont les nœuds contiennent une simple valeur entière naturelle. On rappelle les principales propriétés de cette structure de données :

- chaque nœud de l'arbre a au plus 2 fils ;
- le sous-arbre gauche d'un nœud N donné ne contient que des nœuds dont la valeur est strictement inférieure à la valeur de N ;
- le sous-arbre droit d'un nœud N donné ne contient que des nœuds dont la valeur est strictement supérieure à la valeur de N ;
- les sous-arbres gauche et droit d'un nœud donné sont aussi des arbres binaires de recherche.

Il vient naturellement de ses propriétés que chaque clé est unique (*i.e.* il n'existe pas plusieurs nœuds de même valeur).

Le schéma ci-dessous est un exemple d'ABR qu'on va utiliser dans cet exercice :



On représente le type des nœuds d'un ABR par la structure suivante :

```
struct noeud_t {
    uint64_t val;       // valeur d'un noeud
    struct noeud_t *fg; // fils gauche du noeud
    struct noeud_t *fd; // fils droit du noeud
};
```

Un ABR est représenté simplement par un pointeur vers un nœud : `struct noeud_t *abr`. Si `abr == NULL` alors l'ABR est vide.

Le fichier `abr.c` contient le programme principal et quelques fonctions servant à tester le code écrit en assembleur.

Question 1 Implanter la fonction `est_present` : cette fonction renvoie vrai ssi la valeur donnée en paramètre est présente dans l'ABR passé en argument.

La fonction doit être la traduction du code C suivant :

```
bool est_present(uint64_t val, struct noeud_t *abr)
{
   if (NULL == abr) {
       return false;
   } else if (val == abr->val) {
       return true:
   } else if (val < abr->val) {
       return est_present(val, abr->fg);
   } else {
       return est_present(val, abr->fd);
   }
}
```

Le principe de la fonction à écrire est simple :

- si l'arbre est vide, alors la valeur n'est sûrement pas présente : on renvoie faux ;
- sinon si le nœud courant contient la valeur recherchée : on renvoie vrai ;
- sinon si la valeur recherchée est plus petite que la valeur du nœud courant, on poursuit la recherche dans le fils gauche ;
- sinon (c'est à dire si la valeur recherchée est plus grande que la valeur du nœud courant), on poursuit la recherche dans le fils droit.

On rappelle qu'un booléen est représenté par un octet contenant 1 pour `true` ou 0 pour `false`.

Question 2 Implanter la fonction `abr_vers_tab` : cette fonction copie les valeurs de l'ABR passé en argument dans un tableau désigné par une variable globale du programme.

La fonction doit être la traduction du code C suivant :

```
void abr_vers_tab(struct noeud_t *abr)
{
    if (abr != NULL) {
        abr_vers_tab(abr->fg);
        *ptr = abr->val;
        ptr++;
        struct noeud_t *fd = abr->fd;
        free(abr);
        abr_vers_tab(fd);
    }
}
```

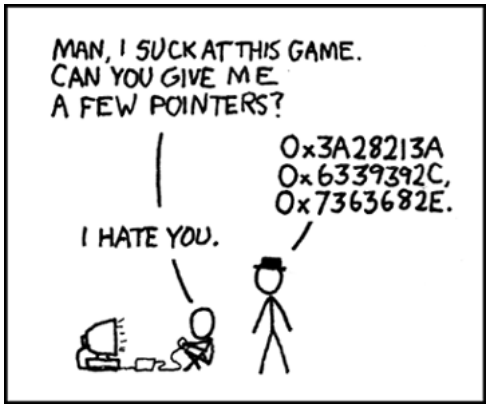
La fonction parcours l'ABR et copie les valeurs des nœuds de l'arbre dans un tableau. Comme le parcours se fait en profondeur d'abord dans le sous-arbre gauche, puis dans celui de droite, le tableau final sera donc trié par ordre strictement croissant. Au passage, la fonction détruit les nœuds pour récupérer l'espace mémoire.

Le tableau est alloué dans le programme principal et on copie ensuite son adresse dans une variable globale `uint64_t *ptr` de type « pointeur vers un entier ». Cette variable est physiquement localisée dans la zone `.data` du fichier `fct_abr.s`, et elle est rendue visible dans le fichier `abr.c` grâce au mot clé `extern`. **Vous devez donc déclarer la variable `ptr` dans le fichier `fct_abr.s`.**

Le principe de la fonction de copie est simple, pour un arbre non-vide :

— on copie récursivement tous les éléments du fils gauche du nœud courant dans le tableau ;
— on copie la valeur du nœud courant dans le tableau ;
— on incrémente la variable `ptr` de façon à ce qu'elle pointe sur la case suivante du tableau ;
— on détruit le nœud courant (avec la fonction `free`) : pour pouvoir continuer à parcourir le reste de l'arbre, on doit donc d'abord sauvegarder un pointeur vers le fils droit du nœud ;
— on copie récursivement tous les éléments du fils droit dans le tableau.

La variable `ptr` sert donc à désigner la prochaine case libre dans le tableau, et on l'avance d'une case à chaque fois qu'on copie une valeur dans le tableau.



`http://xkcd.com/`