Logiciel de Base

Examen pratique
1A par alternance
Année 2017–2018
Session normale

Durée : 2h00
Ce document contient 5 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.* ordinateur portable, téléphones, clés USB, *etc.*) interdit. Toute tentative de fraude entrainera une convocation devant la Section Disciplinaire de Grenoble-INP.

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

Le code 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`.

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 système, 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 : Tri par recherche du maximum *(9 pts)*

Toutes les fonctions de cet exercice devront être implantées en assembleur **64 bits** pour la plate-forme `x86_64`. On demande à chaque fois **une traduction systématique, sans aucune optimisation** : les variables locales devront notamment être stockées et accédées depuis la pile. Les fonctions en assembleur seront écrites dans un fichier que vous devez créer et qui doit s'appeler `fct_trimax.s`.

On va travailler dans cet exercice sur un algorithme de tri de listes chainées.

Le type d'une cellule est défini dans le programme principal `trimax.c` :

```
struct cellule_t {
    uint16_t val;
    struct cellule_t *suiv;
};
```

Une liste sera simplement un pointeur sur la première cellule (pas de sentinelle en tête). On va donc travailler sur des listes d'entiers naturels sur 16 bits.

Le programme principal contient des tests ainsi que des fonctions de référence écrites en C, et il va falloir en traduire certaines en assembleur.

Question 1 Création de la liste : écrivez la fonction `creer_liste1` en assembleur.

La fonction de référence à traduire est la suivante :

```
struct cellule_t *creer_liste0(uint16_t tab[], uint32_t taille)
{
    struct cellule_t *liste = NULL;
    for (int32_t i = taille - 1; i >= 0; i--) {
        struct cellule_t *cell = malloc(sizeof(struct cellule_t));
        assert(NULL != cell);
        cell->val = tab[i];
        cell->suiv = liste;
        liste = cell;
    }
    return liste;
}
```

Cette fonction utilise la macro `assert` : cette macro n'est pas traduisible, vous l'ignorerez donc simplement dans votre code assembleur.

On rappelle que le compilateur C aligne les champs d'une structure sur des adresses multiples de la taille du type du champs, et que le premier champs d'une structure est toujours au début de la structure en mémoire (dit autrement, il n'y a jamais d'octets de *padding* avant le premier champs d'une structure, par contre il peut y en avoir entre le premier et le deuxième champs).

Question 2 Tri de la liste : écrivez la fonction `trier_liste1` en assembleur.

Cette fonction doit implanter l'algorithme du tri par recherche du maximum, dont on donne la fonction de référence en C ci-dessous :

```
void trier_liste0(struct cellule_t **liste)
{
    assert(NULL != liste);
    struct cellule_t fictif;
    fictif.suiv = *liste;
    *liste = NULL;
    while (NULL != fictif.suiv) {
        struct cellule_t *prec_max = &fictif;
        struct cellule_t *prec = fictif.suiv;
        while (NULL != prec->suiv) {
            if (prec->suiv->val > prec_max->suiv->val) {
                prec_max = prec;
            }
            prec = prec->suiv;
        }
        prec = prec_max->suiv->suiv;
        prec_max->suiv->suiv = *liste;
        *liste = prec_max->suiv;
        prec_max->suiv = prec;
    }
}
```

Là-encore, vous ignorerez la macro `assert` dans votre traduction.

Cette fonction utilise une sentinelle (*i.e.* un élément fictif en tête) : vous **devez** faire de même dans votre code assembleur.

Note : cette fonction est la plus longue de l'énoncé, ne restez pas bloqués dessus et gardez du temps pour l'exercice suivant.

Ex. 3 : Copie de zones mémoires *(10 pts)*

Toutes les fonctions de cet exercice sont à écrire en assembleur **32 bits** pour la plate-forme `x86_32`.

Attention : on rappelle que la version de Valgrind disponible sur les machines de l'école ne supporte pas le code 32 bits, vous ne pouvez donc pas utiliser ce logiciel pour cet exercice (où il ne serait pas très utile de toute façon).

On va implanter dans cet exercice des fonctions de copie de zones mémoire, sur le modèle de la fonction C standard `memcpy`.

On fourni un programme principal dans le fichier `copiemem.c`. Ce programme prend en paramètre sur la ligne de commande le nombre d'octet de la zone mémoire à copier, par exemple `./copiemem 15` crée et copie une zone de 15 octets.

La zone mémoire source est initialisée avec des octets aléatoires, et recopiée plusieurs fois dans une zone mémoire destination, en utilisant à chaque fois une fonction de copie différente :

1. la première fois en utilisant une fonction de copie naïve `copiemem0` écrite en C et fournie dans le fichier `copiemem.c` ;
2. la deuxième fois en utilisant une fonction assembleur `copiemem1` que vous devrez écrire et qui sera une simple traduction naïve de la fonction C `copiemem0` ;

3. la troisième fois en utilisant une fonction assembleur `copiemem2` que vous devrez écrire et qui sera une optimisation de la fonction `copiemem1` précédente ;
4. la quatrième fois en utilisant une fonction assembleur `copiemem3` que vous devrez écrire et qui utilisera une instruction de copie mémoire native au processeur x86 ;
5. la cinquième fois enfin en utilisant la fonction `memcpy` de la bibliothèque C standard.

Normalement, ces fonctions devraient être de plus en plus performantes, mais cela peut varier d'une machine à l'autre. Il est cependant possible que la fonction C naïve soit au final plus performante que certaines fonctions assembleur, car GCC est capable de faire des optimisations très sophistiquées.

Pour chaque copie, le programme principal :

— alloue une nouvelle zone destination initialisée avec des 0 ;
— mesure le temps d'exécution de la copie ;
— vérifie que la zone destination contient bien les mêmes octets que la zone source et affiche un message d'erreur le cas échéant ;
— affiche le contenu de la zone destination si elle est de taille raisonnable (20 octets ou moins) ;
— détruit la zone destination (pour minimiser des effets de gestion mémoire qui pourraient perturber les performances).

Bien sûr, les temps d'exécution n'ont de sens que sur des grandes zones mémoires. On recommande donc :

— de mettre au point les différentes fonctions sur des petites zones (par exemple 15 octets) pour pouvoir vérifier visuellement que tout se passe bien et localiser d'éventuelles erreurs de copie ;
— de tester ensuite sur une très grande zone (par exemple 500 MiO) pour avoir une idée des performances relatives des différentes fonctions.

Note : ne dépassez pas un giga-octet pour la zone à copier, cela risque de saturer la mémoire de la machine et de provoquer un plantage du système.

Toutes les fonctions à écrire en assembleur le seront dans un fichier qui doit s'appeler `fct_copiemem.s` et que vous devez créer et compléter au fur et à mesure. Vous pouvez modifier le programme principal en C si cela vous est utile pour mettre au point vos fonctions (par exemple pour commenter les appels aux fonctions que vous n'avez pas encore implantées) mais on ne vous demande notamment pas de rajouter des tests à ceux déjà donnés dans le programme.

Question 1 Implanter la fonction `copiemem1` en assembleur 32 bits dans le fichier `fct_copiemem.s`.

Cette fonction sera une traduction **systématique** de la fonction C de référence :

```
void copiemem0(uint8_t *dst, uint8_t *src, uint32_t taille)
{
    for (uint32_t i = 0; i < taille; i++) {
        dst[i] = src[i];
    }
}
```

Vous ne devez effectuer **aucune** optimisation dans l'écriture de cette fonction : on attend une traduction ligne par ligne de la fonction C en assembleur 32 bits. Notamment, les accès aux paramètres et aux variables locales se feront systématiquement depuis la pile, même si cela implique d'effectuer des accès redondants.

Question 2 Implanter la fonction `copiemem2` en assembleur 32 bits dans le fichier `fct_copiemem.s`.

Cette fonction sera une optimisation de la fonction `copiemem1` précédente. Pour cela, vous devrez :

— recopier le code de la fonction `copiemem1` que vous avez écrite ;
— copier les valeurs initiales des paramètres et des variables locales dans des registres ;
— remplacer les accès à la pile par des accès à ces registres dans le corps de la boucle de votre fonction.

Attention : vous devez respecter les conventions d'utilisation des registres et sauvegarder les registres non-scratch que vous utilisez éventuellement dans l'implantation de la fonction, car le compilateur C a pu y laisser des valeurs importantes avant d'appeler la fonction `copiemem2`.

Question 3 Implanter la fonction `copiemem3` en assembleur 32 bits dans le fichier `fct_copiemem.s`.

Pour cette fonction, on va utiliser des instructions natives du processeur x86 : les instructions de la classe `movs` (*move string*) qui sont capables de copier des mots de données d'une zone mémoire à une autre.

Le principe de cette classe d'instruction est le suivant :
— on place l'adresse de la zone source dans le registre `%esi` ;
— on place l'adresse de la zone destination dans le registre `%edi` ;
— on affecte le bit `D` du registre `%eflags` pour préciser le sens de la copie ;
— on exécute l'instruction `movs` en précisant un suffixe de taille correspondant à la taille des données qu'on veut copier : **b** pour un octet, **w** pour un mot de 16 bits ou **l** pour un mot de 32 bits (le suffixe **q** n'est pas utilisable directement en 32 bits) ;
— en fonction du bit `D`, les registres `%esi` et `%edi` sont automatiquement incrémentés ou décrémentés du nombre d'octets correspondant à la taille des données qu'on a copié.

Pour affecter le bit `D`, on utilise les instructions suivantes :
— `cld` met le bit `D` à zéro, ce qui signifie qu'on veut incrémenter `%esi` et `%edi` ;
— `std` met le bit `D` à un, ce qui signifie qu'on veut décrementer `%esi` et `%edi`.

Bien sûr, il faudra répéter l'exécution de l'instruction `movs` pour copier toute la zone mémoire, et pas seulement un mot. On pourrait utiliser une boucle, mais l'architecture x86 fourni une pré-instruction particulière (que l'on appelle un préfixe) qui permet de répéter l'exécution de certaines instructions (dont la classe `movs`) de façon très efficace : l'instruction `rep`.

Le préfixe `rep` répète l'exécution de l'instruction `movs` autant de fois que la valeur stockée dans le registre `%ecx`, qui est automatiquement décrémenté de 1 à chaque exécution du `movs`. Il suffit d'écrire le préfixe `rep` avant l'instruction `movs`, sur la même ligne, pour que le `movs` soit répété.

Vous utiliserez donc ces instructions pour implanter la fonction `copiemem3` : le temps d'exécution de cette fonction sur une très grosse zone de donnée (au moins 500 MiO) devrait se rapprocher de celui de la fonction `memcpy` de la bibliothèque C standard (les temps peuvent être variables d'une exécution à l'autre).