Assembleur

A. Appel de fonctions

b)

Après avoir chargé le programme dans le débogueur x96dbg, nous avons placé un point d'arrêt sur la ligne `call crt_printf` pour observer le déroulement juste avant l'appel à la fonction `printf`. Une fois le point d'arrêt atteint, nous avons examiné la pile pour voir comment les arguments étaient passés à la fonction.

Voici ce que l'on obsere dans la fenêtre de la pile :

- L'adresse de retour était en haut de la pile. C'est l'adresse à laquelle le contrôle du programme reviendra après l'exécution de `printf`.
- Juste en dessous, il y avait l'adresse de la chaîne `Phrase` (qui contient le texte "Hello World : %d", ainsi que le caractère de nouvelle ligne et le caractère nul). Cette adresse est le deuxième argument de `printf`, indiquant où la chaîne format se trouve en mémoire.
- Encore en dessous, le nombre `42` était présent, qui est le premier argument de `printf` et correspond à l'entier à afficher dans la chaîne formatée (l'ordre est logique car stack LIFO)

On remarque que l'on push les paramètres avant de call la fonction.

Note : `offset textString` donne l'adresse mémoire où commence la chaîne `"Hello, world!"`. Cette adresse est poussée sur la pile. `printf` lit cette adresse, navigue à cette adresse mémoire, et lit la chaîne jusqu'à rencontrer le caractère nul, puis l'affiche.

Si on en met pas le offset et seulement le `push textString`, on push la valeur du premier caractère de `textString` sur la pile. En ASCII, "H" est 72, donc on push 72 sur la pile, et `printf` nous a causé un crash dans ce cas sans offset.

c)

Voici un résumé de l'évolution pas à pas de la pile pendant l'exécution du programme HelloWorld :

1. **Débogueur :**

Tout d'abord, on se déplace à l'Entry point car il y a beaucoup d'exécution au début (surement dû au "include" des différentes librairies).
Sur le débogueur, on ouvre le .exe du programme et on fait "débogueur>run to user code"

2. **Le push des éléments :**

Le programme commence par push les arguments :

- `push 42` : L'entier 42 est poussé sur la pile. Cette valeur est l'argument qui sera utilisé par `printf` pour le formatage. 42 apparaît au sommet de la pile.
- `push offset Phrase` : L'adresse de la chaîne `Phrase` est poussée sur la pile. Cette adresse pointe vers la chaîne formatée "Hello World : %d", suivie d'une nouvelle ligne et d'un caractère nul. Cette valeur se place au-dessus de 42 dans la pile.

3. **Appel à `crt_printf` :**

- À ce moment, la pile contient, du sommet vers le bas, l'adresse de `Phrase` puis le nombre 42, et en dessous, l'adresse de retour qui pointe vers l'instruction suivante après `call crt_printf`. Lorsque `printf` est appelé, il utilise ces valeurs pour le formatage et l'affichage, puis, une fois l'exécution de `printf` terminée, ces valeurs sont retirées de la pile.

4. **Exécution de `crt_system` :**

- `invoke crt_system, offset strCommand` : La commande `invoke` est utilisée pour appeler `crt_system` avec l'adresse de `strCommand`. `invoke` semble gère automatiquement le placement de l'adresse de `strCommand` sur la pile. Cette adresse pointe vers la chaîne "Pause", permettant au système d'exécuter la commande pause dans le terminal.

5. **Terminaison du programme :**

- `invoke ExitProcess, eax` : Avant cet appel, `eax` est mis à 0, qui est ensuite utilisé comme argument pour `ExitProcess`, ce qui met fin au programme.

C. Variables locales :

b)

Cette fonction calcule le nième terme de la séquence de la suite de Fibonacci.

```
C:\Users\pierr\OneDrive\Desktop\ENSIBS_2023\Assembleur\TP\TP1\Exercice\c-a-Myst>Variableslocales.exe
K = 610
```

On obtient bien le résultat attendu.

C. Un peu de lecture :

a)

Dans l'article "Plongeon dans les appels systèmes Windows", on explore en détail comment un appel système est exécuté sous Windows, du programme utilisateur au noyau. L'auteur commence par clarifier les différents niveaux de privilège sur les processeurs x86, en particulier le ring 0 pour le noyau et le ring 3 pour les applications utilisateurs.

On suit ensuite un exemple concret avec un programme en C qui utilise l'API `FindFirstFile` pour montrer comment les appels aux API Windows sont transformés en appels systèmes. Grâce à l'usage d'un débogueur, on visualise les étapes qui mènent à l'appel système, notamment le mécanisme de trampoline via l'Import Address Table (IAT).

L'article explique également le rôle crucial de l'instruction `SYSENTER` pour passer du mode utilisateur au mode noyau et comment les registres MSR spéciaux facilitent cette transition. Une fois en mode noyau, la System Service Dispatch Table (SSDT) est utilisée pour acheminer les appels systèmes vers les fonctions adéquates.