ARCHITECTURE DES ORDINATEURS

TP : **07**

Étude de quelques optimisations effectuées par le compilateur et débuggage

Copiez le répertoire `/net/cremi/sathibau/Archi/optims` dans votre *home*. Le fichier `Makefile` contient des règles pour compiler des programmes C avec différentes options d'optimisation, de `0` (pas d'optimisations) à `3` (optimisations très agressives), et `s` (optimisé en taille). Pour chaque partie, vous pouvez donc écrire un simple fichier « `.c` » contenant juste une fonction, sans aucun `#include` ni `main()`, et taper simplement `make` pour obtenir les différentes versions optimisées.

1 Multiplication/division par une constante

Multiplication et division sont des opérations *a priori* coûteuses. gcc est cependant capable d'optimiser grandement dans le cas où le facteur (resp. le quotient) est connu.

Écrivez une fonction `unsigned f(unsigned x)` retournant `4*x` . Remarquez l'utilisation d'une instruction de décalage, ici de 2 bits, vers la gauche (`sal` ou `shl`) pour effectuer la multiplication par 4. Il est donc inutile, lorsque vous écrivez des programmes C, de remplacer une multiplication par 4 par un décalage à gauche de 2 bits, il vaut mieux garder une version la plus lisible possible, le compilateur s'occupe d'optimiser.

Remplacez 4 par 5. Remarquez que selon le niveau d'optimisation utilisé, gcc va utiliser l'instruction `leal` (`leal (a,b,i),c` effectue `c=a+b*i`), ou bien simplement une multiplication. Essayez 7, observez le code produit.

Écrivez une fonction `unsigned g(unsigned x)` retournant `x/2` , observez le code produit (`sar` ou `shr` décalent vers la droite). Remplacez 2 par 3, observez le code produit avec `-O1`. Il est à noter que -1431655765 s'écrit `0xaaaaaaab` en hexadécimal, et qu'en le multipliant par 3, on obtient `0x200000001` (qui "déborde" donc de 32bits). Sachant que `mull` effectue une multiplication entre `%eax` et l'opérande fourni, et range la partie "basse" dans `%eax` et la partie "haute" dans `%edx`, pourquoi est-ce correct ?

De manière générale donc, gcc sait très bien optimiser des calculs, il vaut donc mieux écrire ses programmes de la manière la plus lisible plutôt que chercher à optimiser "à la main".

2 `switch`

Observez le résultat de la compilation du fichier `switch.c`, notamment l'instruction `jmp`. Note : `ja` est presque la même chose que `jg`, de même pour `jb` et `jl`. Quelle est la méthode utilisée ?

Note : vous pouvez voir dans `man ascii` l'encodage des caractères.

Observez le résultat de la compilation du fichier `printf.c`, notamment la transformation de `printf` en `puts`, et la factorisation de l'appel.

3 Pour aller plus loin : `strlen`

Observez le résultat de la compilation avec `-O1` et `-Os` du fichier `strlen.c`, remarquez l'astuce utilisée dans le deuxième cas.

4 Pour aller plus loin : Débuggage en assembleur avec gdb

Dans le répertoire `crash/` vous trouverez un programme qui, lorsqu'on le lance, imprime quelque chose, et crashe. Lancez-le dans gdb : `gdb ./crash` et tapez `run`. `backtrace` n'est guère utile... le programme `strace` montre que la dernière chose que le programme a réussi à faire c'est `write`. Dans `gdb`, tapez donc `break write` pour poser un breakpoint sur cet appel. Relancez le programme de zéro avec `run` . Quand `gdb` s'arrête sur `write`, utilisez `finish` pour terminer cet appel et revenir à `f` . Utilisez `disassemble` pour voir la version assembleur de `f`. Ensuite vous pouvez continuer le programme instruction par instruction à l'aide de la commande `stepi`, et appeler `info registers` pour examiner la valeur des registres (la valeur de `eip` contient l'adresse de l'instruction en cours, c'est équivalent à `pc`). Vous pouvez examiner la mémoire en utilisant par exemple `p/x *(unsigned*)0xffffd834`, observez notamment l'adresse de retour qui est très étrange (en temps normal elle devrait être de la forme `0x8012345`). L'adresse de retour a donc été écrasée. Faites un dessin de la pile. Pour savoir qui a écrasé l'adresse de retour, vous pouvez par exemple utiliser `stepi` pour progresser jusqu'à l'instruction `ret`, et là utiliser `info registers` pour récupérer la valeur de `esp`, qui est alors l'adresse de l'adresse de retour. Vous pouvez alors utiliser `watch` pour savoir qui l'écrase : commencer par redémarrer au début de `f`, en utilisant `break f` et `run` de nouveau, puis `watch *(unsigned*)0xffff1234` pour dire à gdb de tracer un emplacement mémoire précis , celui qui se fait corrompre dans notre cas (donc remplacez `0xffff1234` par l'adresse de l'adresse de retour, obtenue ci-dessus), et un simple `continue` va continuer le programme jusqu'au fautif, et `backtrace` indique d'où cela vient (le buffer est simplement trop petit, et donc `snprintf` en déborde !).