Assembly HOWTO

François-René Rideau rideau@ens.fr v0.4l, 16 Novembre 1997

(Version française réalisée par Eric Dumas dumas@freenix.fr dumas@Linux.EU.Org, et Faré Rideau rideau@ens.fr, 11 Novembre 1997). Ce document décrit comment programmer en assembleur x86 en n'utilisant que des outils de développement *libres*, et tout particulièrement avec le système d'exploitation Linux sur la plate-forme i386. Les informations contenues dans ce document peuvent être applicables ou non applicables à d'autres plates-formes matérielles ou logicielles. Les contributions à ce documents seront acceptées avec gratitude. *mots-clefs*: assembleur, libre, macroprocesseur, préprocesseur, asm, inline asm, 32 bits, x86, i386, gas, as86, nasm

Contents

1 Introduction

1.1 Copyright

1.2 Note importante

Ceci est censé être la dernière version que j'écrirai de ce document. Il y a un candidat pour reprendre en charge le document, mais jusqu'à ce qu'il le reprenne complètement en main, je serai heureux de m'occuper de tout courrier concernant ce document.

Vous êtes tout spécialement invités à poser des questions, à y répondre, à corriger les données, à ajouter de nouvelles informations, à compléter les références sur d'autres logiciels, à mettre en évidence les erreurs et lacunes du document. Si vous êtes motivés, vous pouvez même **prendre en charge ce document**. En un mot, apporter votre contribution!

Pour contribuer à ce document, contactez la personne qui apparaît actuellement en charge. Au moment où j'écris ces lignes, il s'agit de

François-René Rideau `mailto:rideau@clipper.ens.fr`) ainsi que de

Paul Anderson `mailto:paul@geeky1.ebtech.net` .

1.3 Avant-Propos

Ce document est destiné à répondre aux questions les plus fréquemment posées par les gens qui développent ou qui souhaitent développer des programmes en assembleurs x86 32 bits en utilisant des logiciels *libres*, et tout particulièrement sous Linux. Vous y trouverez également des liens sur d'autres documents traitant d'assembleur, fondés sur des outils logiciels qui ne sont pas libres, pas 32-bit, ou pas dédiés à l'architecture x86, bien que cela ne soit pas le but principal de ce document.

Etant donné que l'intéret principal de la programmation en assembleur est d'établir les fondations de systèmes d'exploitation, d'interpréteurs, de compilateurs, et de jeux, là où un compilateur C n'arrive plus à fournir le pouvoir d'expression nécessaire (les performances étant de plus en plus rarement un problème), nous insisteront sur le développement de tels logiciels.

1.3.1 Comment utiliser ce document

Ce document contient des réponses à un certain nombre de questions fréquemment posées. Des URL y sont donnés, qui pointent sur des sites contenant documents ou logiciels. Prenez conscience que les plus utiles de ces sites sont dupliqués sur des serveurs miroirs, et qu'en utilisant le site miroir le plus proche de chez vous,

vous évitez à un gâchis inutile aussi bien de précieuses ressources réseau communes à l'Internet que de votre propre temps. Ainsi, il existe un certain nombre de gros serveurs disséminés sur la planète, qui effectuent la duplication d'autres sites importants. Cherchez où se trouvent ces sites et identifiez les plus proches de chez vous (du point de vue du réseau). Parfois, la liste des miroirs est données dans un fichier ou dans le message de connexion. Suivez ces conseils. Si ces informations ne sont pas présentes, utilisez le programme archie.

La version la plus récente de ce document peut être trouvée sur

`http://www.eleves.ens.fr:8080/home/rideau/Assembly-HOWTO`

ou

`http://www.eleves.ens.fr:8080/home/rideau/Assembly-HOWTO.sgml`

mais les répertoires de HowTo Linux *devraient* normalement être à peu près à jour (je ne peux pas le garentir):

`ftp://sunsite.unc.edu/pub/Linux/docs/HOWTO/` (?)

La version française de ce document peut être trouvée sur le site

`ftp://ftp.ibp.fr/pub/linux/french/HOWTO/`

1.3.2 Autres documents de référence

- si vous ne savez ce qu'est le *libre* logiciel, lisez avec attention la GPL (GNU General Public License), qui est utilisée dans un grand nombre de logiciels libres, et est une source d'inspiration pour la plupart des autres licences d'utilisations de logiciels libres. Elle se trouve généralement dans un fichier nommé `COPYING`, avec une version pour les bibliothèques de routines dans un fichier nommé `COPYING.LIB`. Les écrits publiés par la FSF (free software foundation) peuvent également vous aider à comprendre le phénomène.
- plus précisément, les logiciels libres intéressants sont ceux desquels les sources sont disponibles, que l'on peut consulter, corriger, et desquels on peut emprunter une partie. Lisez les licences d'utilisation avec attention et conformez-vous y.
- il existe une FAQ dans le forum de discussion comp.lang.asm.x86 qui répond aux questions générales concernant la programmation en assembleur pour x86, et aux questions concernant certains assembleurs commerciaux dans un environnement DOS 16 bits. Certaines de ces réponses peuvent s'appliquer à la programmation 32 bits, aussi serez-vous sans-doute intéressés de lire cette FAQ...

  `http://www2.dgsys.com/~raymoon/faq/asmfaq.zip`
- Sont disponibles des FAQs, de la documentation, et des sources, concernant la programmation sur votre plate-forme préférée, quelqu'elle soit, et vous devriez les consulter pour les problèmes liés à votre plate-forme qui ne seraient pas spécifique à la programmation en assembleur.

1.4 Historique de document

Chaque version inclue quelques modifications et corrections mineures, qui ne sont pas indiquées à chaque fois.

Version 0.1 23 Avril 1996

Francois-Rene "Faré" Rideau <rideau@ens.fr> crée et diffuse initialement le document sous forme d'un mini-HOWTO car “Je suis un peu fatigué d'avoir à répondre encore et toujours aux mêmes questions dans le forum comp.lang.asm.x86”

Version 0.2 4 Mai 1996

*

Version 0.3c 15 Juin 1996

*

Version 0.3f 17 Octobre 1996

Tim Potter indique l'option -fasm pour activer l'assembleur en-ligne de GCC sans le reste des optimisations de -O.

Version 0.3g 2 Novembre 1996

Création de l'historique. Ajout de pointeurs dans la section sur la compilation croisée. Ajout d'une section concernant la programmation des entrées/sorties sous Linux (en particulier pour l'accès vidéo).

Version 0.3h 6 Novembre 1996

plus sur la compilation croisée - voir sur sunsite: devel/msdos/

Version 0.3i 16 Novembre 1996

NASM commence à être particulièrement intéressant

Version 0.3j 24 Novembre 1996

Référence sur la version française

Version 0.3k 19 Décembre 1996

Quoi? J'avais oublié de parler de Terse?

Version 0.3l 11 Janvier 1997

*

Version 0.4pre1 13 Janvier 1997

Le mini-HOWTO au format texte est transformé en un authentique HOWTO au format linuxdoc-sgml, pour explorer les possibilités dudit format.

Version 0.4 20 Janvier 1997

Première diffusion de ce HOWTO.

Version 0.4a 20 Janvier 1997

Ajout de la section CREDITS

Version 0.4b 3 Février 1997

NASM mis avant AS86

Version 0.4c 9 Février 1997

Ajout de la partie "Avez-vous besoin d'utilisateur l'assembleur?"

Version 0.4d 28 Février 1997

Annonce fantôme d'un nouveau responsable de ce HowTo.

Version 0.4e 13 Mar 1997

Version diffusée pour DrLinux

Version 0.4f 20 Mars 1997

*

Version 0.4g 30 Mars 1997

*

Version 0.4h 19 Juin 1997

Ajouts à propos de "Comment ne pas utiliser l'assembleur"; mises à jour concernant NASM et GAS.

Version 0.4i 17 Juillet 1997

Informations sur l'accès au mode 16 bits à partir de Linux.

Version 0.4j 7 September 1997

*

Version 0.4k 19 Octobre 1997

je (Faré) reprends en main la traduction française du HowTo

Version 0.4l 16 Novembre 1997

version pour LSL 6ème édition.

Il s'agit encore d'une nouvelle "toute dernière version réalisée par Faré avant qu'un nouveau responsable ne prenne la main".

1.5 Crédits

Je souhaiterais remercier les personnes suivantes:

- *Linus Torvalds* `mailto:buried.alive@in.mail`

  pour Linux

- *Bruce Evans* `mailto:bde@zeta.org.au`

  pour bcc d'où as86 est extrait

- *Simon Tatham* `mailto:anakin@poboxes.com` et

  Julian Hall `mailto:jules@earthcorp.com`

  pour NASM.

- *Jim Neil* `mailto:jim-neil@digital.net`

  pour Terse

- *Greg Hankins* `mailto:gregh@sunsite.unc.edu`

  pour la coordination des HOWTOs

- *Raymond Moon* `mailto:raymoon@moonware.dgsys.com`

  pour sa FAQ

- *Eric Dumas* `mailto:dumas@Linux.EU.Org`

  pour la traduction initiale en français... (l'auteur, français, est le premier attristé de devoir écrire l'original en anglais)

- *Paul Anderson* `mailto:paul@geeky1.ebtech.net`

  et *Rahim Azizarab* `mailto:rahim@megsinet.net`

  pour m'avoir aidé, à défaut de reprendre le HowTo en main.

- toutes les personnes qui ont contribué à l'écriture de ce document, par leurs idées, remarques ou leur soutient moral.

2 Avez-vous besoin de l'assembleur?

Je ne veux en aucun cas jouer les empêcheurs-de-tourner-en-rond, mais voici quelques conseils issus d'une expérience gagnée à la dure.

2.1 Le Pour et le Contre

2.1.1 Les avantages de l'assembleur

L'assembleur peut vous permettre de réaliser des opérations très bas niveau:

- vous pouvez accéder aux registres et aux ports d'entrées/sorties spécifiques à votre machine;
- vous pouvez parfaitement contrôler le comportemant du code dans des sections critiques où pourraient sinon advenir un blocage du processeur ou des périphériques;
- vous pouvez sortir des conventions de production de code de votre compilateur habituel; ce qui peut vous permettre d'effectuer certaines optimisations (par exemple contourner les règles d'allocation mémoire, gérer manuellement le cours de l'éxécution, etc.);
- accéder à des modes de programmation non courants de votre processeur (par exemple du code 16 bits pour l'amorçage ou l'interfaçage avec le BIOS, sur les pécés Intel);
- vous pouvez construire des interfaces entre des fragments de codes utilisant des conventions incompatibles (c'est-à-dire produit par des compilateurs différents ou séparés par une interface bas-niveau);
- vous pouvez générer un code assez rapide pour les boucles importantes pour pallier aux défauts d'un compilateur qui ne sait les optimiser (mais bon, il existe des compilateurs optimisateurs librement disponibles!);
- vous pouvez générer du code optimisé "à la main" qui est plus parfaitement règlé pour votre configuration matérielle précise, même s'il ne l'est pour aucune autre configuration;
- vous pouvez écrire du code pour le compilateur optimisateur de votre nouveau langage. (c'est là une activité à laquelle peu se livrent, et encore, rarement.)

2.1.2 Les inconvénients de l'assembleur

L'assembleur est un langage très bas niveau (le langage du plus bas niveau qui soit au dessus du codage à la main de motifs d'instructions en binaire). En conséquence:

- l'écriture de code en est longue et ennuyeuse;
- les bogues apparaissent aisément;
- les bogues sont difficiles à repérer et supprimer;
- il est difficile de comprendre et de modifier du code (la maintenance est très compliquée);
- le résultat est extrêmement peu portable vers une autre architecture, existante ou future;
- votre code ne sera optimisé que une certaine implémentation d'une même architecture: ainsi, parmi les plates-formes compatibles Intel, chaque réalisation d'un processeur et de ses variantes (largeur du bus, vitesse et taille relatives des CPU/caches/RAM/Bus/disques, présence ou non d'un coprocesseur arithmétique, et d'extensions MMX ou autres) implique des techniques d'optimisations parfois radicalement

différentes. Ainsi diffèrent grandement les processeurs déjà existant et leurs variations: Intel 386, 486, Pentium, PPro, Pentium II; Cyrix 5x86, 6x86; AMD K5, K6. Et ce n'est sûrement pas terminé: de nouveaux modèles apparaissent continuellement, et cette liste même sera rapidement dépassée, sans parler du code "optimisé" qui aura été écrit pour l'un quelconque des processeurs ci-dessus.

- le code peut également ne pas être portable entre différents systèmes d'exploitation sur la même architecture, par manque d'outils adaptés (GAS semble fonctionner sur toutes les plates-formes; NASM semble fonctionner ou être facilement adaptable sur toutes les plates-formes compatibles Intel);
- un temps incroyable de programmation sera perdu sur de menus détails, plutôt que d'être efficacement utilisé pour la conception et le choix des algorithmes utilisés, alors que ces derniers sont connus pour être la source de la majeure partie des gains en vitesse d'un programme. Par exemple, un grand temps peut être passé à grapiller quelques cycles en écrivant des routines rapides de manipulation de chaînes ou de listes, alors qu'un remplacement de la structure de données à un haut niveau, par des arbres équilibrés et/ou des tables de hachage permettraient immédiatement un grand gain en vitesse, et une parallélisation aisée, de façon portable permettant un entretien facile.
- une petite modification dans la conception algorithmique d'un programme anéantit la validité du code assembleur si patiemment élaboré, réduisant les développeurs au dilemne de sacrifier le fruit de leur labeur, ou de s'enchaîner à une conception algorithmique obsolète.
- pour des programmes qui fait des choses non point trop éloignées de ce que font les benchmarks standards, les compilateurs/optimiseurs commerciaux produisent du code plus rapide que le code assembleur écrit à la main (c'est moins vrai sur les architectures x86 que sur les architectures RISC, et sans doute moins vrai encore pour les compilateurs librement disponible. Toujours est-il que pour du code C typique, GCC est plus qu'honorable).
- Quoi qu'il en soit, ains le dit le saige John Levine, modérateur de comp.compilers, "les compilateurs rendent aisée l'utilisation de structures de données complexes; ils ne s'arrêtent pas, morts d'ennui, à mi-chemin du travail, et produisent du code de qualité tout à fait satisfaisante". Ils permettent également de propager *correctement* les transformations du code à travers l'ensemble du programme, aussi hénaurme soit-il, et peuvent optimiser le code par-delà les frontières entre procédures ou entre modules.

2.1.3 Affirmation

En pesant le pour et le contre, on peut conclure que si l'assembleur est parfois nécessaire, et peut même être utile dans certains cas où il ne l'est pas, il vaut mieux:

- minimiser l'utilisation de code écrit en assembleur;
- encapsuler ce code dans des interfaces bien définies;
- engendrer automatiquement le code assembleur à partir de motifs écrits dans un langage plus de haut niveau que l'assembleur (par exemple, des macros contenant de l'assembleur en-ligne, avec GCC);
- utiliser des outils automatiques pour transformer ces programmes en code assembleur;
- faire en sorte que le code soit optimisé, si possible;
- utiliser toutes les techniques précédentes à la fois, c'est-à-dire écrire ou étendre la passe d'optimisation d'un compilateur.

Même dans les cas où l'assembleur est nécessaire (par exemple lors de développement d'un système d'exploitation), ce n'est qu'à petite dose, et sans infirmer les principes ci-dessus.

Consultez à ce sujet les sources du noyau de Linux: vous verrez qu'il s'y trouve juste le peu qu'il faut d'assembleur, ce qui permet d'avoir un système d'exploitation rapide, fiable, portable et d'entretien facile. Même un jeu très célèbre comme DOOM a été en sa plus grande partie écrit en C, avec une toute petite routine d'affichage en assembleur pour accélérer un peu.

2.2 Comment ne pas utiliser l'assembleur

2.2.1 Méthode générale pour obtenir du code efficace

Comme le dit Charles Fiterman dans comp.compilers à propos de la différence entre code écrit par l'homme ou la machine,

“L'homme devrait toujours gagner, et voici pourquoi:

- Premièrement, l'homme écrit tout dans un langage de haut nivrau.
- Deuxièmement, il mesure les temps d'éxécution (profiling) pour déterminer les endroits où le programme passe la majeure partie du temps.
- Troisièmement, il demande au compilateur d'engendrer le code assembleur produit pour ces petites sections de code.
- Enfin, il effectue à la main modifications et réglages, à la recherche des petites améliorations possibles par rapport au code engendré par la machine.

L'homme gagne parce qu'il peut utiliser la machine.”

2.2.2 Langages avec des compilateurs optimisateurs

Des langages comme ObjectiveCAML, SML, CommonLISP, Scheme, ADA, Pascal, C, C++, parmi tant d'autres, ont tous des compilateurs optimiseurs librement disponibles, qui optimiseront le gros de vos programmes, et produiront souvent du code meilleur que de l'assembleur fait-main, même pour des boucles serrées, tout en vous permettant de vous concentrer sur des détails haut niveau, et sans vous interdire de gagner par la méthode précédente quelques pourcents de performance supplémentaire, une fois la phase de conception générale terminée. Bien sûr, il existe également des compilateurs optimiseurs commerciaux pour la plupart de ces langages.

Certains langages ont des compilateurs qui produisent du code C qui peut ensuite être optimisé par un compilateur C. C'est le cas des langages LISP, Scheme, Perl, ainsi que de nombreux autres. La vitesse des programmes obtenus est toute à fait satisfaisante.

2.2.3 Procédure générale à suivre pour accélerer votre code

Pour accélérer votre code, vous ne devriez traiter que les portions d'un programme qu'un outil de mesure de temps d'éxécution (profiler) aura identifié comme étant un goulot d'étranglement pour la performance de votre programme.

Ainsi, si vous identifiez une partie du code comme étant trop lente, vous devriez

- d'abord essayer d'utiliser un meilleur algorithme;

- essayer de la compiler au lieu de l'interpréter;
- essayer d'activer les bonnes options d'optimisation de votre compilateur;
- donner au compilateur des indices d'optimisation (déclarations de typage en LISP; utilisation des extensions GNU avec GCC; la plupart des compilos fourmillent d'options);
- enfin de compte seulement, se mettre à l'assembleur si nécessaire.

Enfin, avant d'en venir à cette dernière option, vous devriez inspecter le code généré pour vérifier que le problème vient effectivement d'une mauvaise génération de code, car il se peut fort bien que ce ne soit pas le cas: le code produit par le compilateur pourrait être meilleur que celui que vous auriez écrit, en particulier sur les architectures modernes à pipelines multiples! Il se peut que les portions les plus lentes de votre programme le soit pour des raisons intrinsèques. Les plus gros problèmes sur les architectures modernes à processeur rapide sont dues aux délais introduits par les accès mémoires, manqués des caches et TLB, fautes de page; l'optimisation des registres devient vaine, et il vaut mieux repenser les structures de données et l'enchaînement des routines pour obtenir une meilleur localité des accès mémoire. Il est possible qu'une approche complètement différente du problème soit alors utile.

2.2.4 Inspection du code produit par le compilateur

Il existe de nombreuses raisons pour vouloir regarder le code assembleur produit par le compilateur. Voici ce que vous pourrez faire avec ce code:

- vérifier si le code produit peut ou non être améliorer avec du code assembleur écrit à la main (ou par un réglage différent des options du compilateur);
- quand c'est le cas, commencer à partir de code automatiquement engendré et le modifier plutôt que de repartir de zéro;
- plus généralement, utilisez le code produit comme des scions à greffer, ce qui à tout le moins vous laisse permet d'avoir gratuitement tout le code d'interfaçage avec le monde extérieur.
- repérer des bogues éventuels dus au compilateur lui-même (espérons-le très rare, quitte à se restreindre à des versions "stables" du compilo).

La manière standard d'obtenir le code assembleur généré est d'appeller le compilateur avec l'option `-S`. Cela fonctionne avec la plupart des compilateur Unix y compris le compilateur GNU C (GCC); mais à vous de voir dans votre cas. Pour ce qui est de GCC, il produira un code un peu plus compréhensible avec l'option `-fverbose-asm`. Bien sur, si vous souhaitez obtenir du code assembleur optimisé, n'oubliez pas d'ajouter les options et indices d'optimisation appropriées!

3 Assembleurs

3.1 Assembleur en-ligne de GCC

Le célèbre GNU C/C++ Compiler (GCC), est un compilateur 32 bits optimisant situé au coeur du projet GNU. Il gère assez bien les architectures x86 et permet d'insérer du code assembleur à l'intérieur de programmes C de telle manière que les registres puissent être soit spécifiés soit laissé aux bons soins de GCC. GCC fonctionne sur la plupart des plates-formes dont Linux, *BSD, VSTa, OS/2, *DOS, Win*, etc.

3.1.1 Où trouver GCC

Le site principal de GCC est le site FTP du projet GNU:

`ftp://prep.ai.mit.edu/pub/gnu/`

On y trouve également toutes les applications provenant du projet GNU. Des versions configurées ou pré-compilées pour Linux sont disponibles sur

`ftp://sunsite.unc.edu/pub/Linux/GCC/` . Il existe un grand nombre de miroirs FTP des deux sites partout de par le monde, aussi bien que des copies sur CD-ROM.

Le groupe de développement de GCC s'est récemment scindé en deux; pour plus d'informations sur la version expérimentale, egcs, voir

`http://www.cygnus.com/egcs/`

Les sources adaptés à votre système d'exploitation préféré ainsi que les binaires précompilés peuvent être trouvés sur les sites FTP courants.

Le portage le plus célèbre de GCC pour DOS est DJGPP et il peut être trouvé dans le répertoire du même nom sur les sites ftp. Voir:

`http://www.delorie.com/djgpp/`

Il existe également un portage de GCC pour OS/2 appelé EMX qui fonctionne également sous DOS et inclut un grand nombre de routines d'émulation Unix. Voir les sites

`http://www.leo.org/pub/comp/os/os2/gnu/emx+gcc/`

`http://warp.eecs.berkeley.edu/os2/software/shareware/emx.html`

`ftp://ftp-os2.cdrom.com/pub/os2/emx09c/`

3.1.2 Où trouver de la documentation sur l'assembleur en ligne avec GCC?

La document de GCC inclus les fichiers de documentation au format texinfo. Vous pouvez les compiler avec TeX et les imprimer, ou les convertir au format .info et les parcourir interactivement avec emacs, ou encore les convertir au format HTML, ou en à peu près n'importe quel format (avec les outils adéquats). Les fichiers .info sont généralement installés en même temps que GCC.

La section à consulter est `C Extensions::Extended Asm::`

La section `Invoking GCC::Submodel Options::i386 Options::` peut également vous aider. En particulier, elle donne les noms de contraintes pour les registres du i386: abcdSDB correspondent respectivement à `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp` (aucune lettre pour `%esp`).

Le site "DJGPP Games resource" (qui n'est pas réservé aux seuls développeurs de jeux) possède une page particulière sur l'assembleur:

`http://www.rt66.com/~brennan/djgpp/djgpp_asm.html`

Enfin, il existe une page de la Toile appelée "DJGPP Quick ASM Programming Guide", contenant des URL sur des FAQ, la syntaxe assembleur AT&T x86, des informations sur l'assembleur en ligne, et la conversion des fichiers .obj/.lib:

`http://remus.rutgers.edu/~avly/djasm.html`

GCC soutraite l'assemblage proprement dit à GAS et suit donc sa syntaxe (voir plus bas), cela implique que l'assembleur en ligne doit utiliser des caractères pourcents entre apostrophes pour qu'ils soient passés à GAS. Voir la section dédiée à GAS.

Vous trouverez un *grand* nombre d'exemples instructifs dans le répertoire `linux/include/asm-i386/` des sources de Linux.

3.1.3 Appeller GCC pour obtenir du code assembleur en ligne correcte?

Assurez-vous d'appeller gcc avec l'option `-O` (ou `-O2`, `-O3`, etc) pour activer les optimisations et l'assembleur en ligne. Si vous ne le faîtes pas, votre code pourra compiler mais ne pas s'exécuter correctement!! En fait (merci à Tim Potter, timbo@moshpit.air.net.au), il suffit d'utiliser l'option `-fasm`, faisant partie de toutes les fonctionnalités activées par l'option `-O`. Donc si vous avez des problèmes en raison d'optimisations boguées dans votre implémentation de gcc, vous pouvez toujours utiliser l'assembleur en ligne. De même, utilisez l'option `-fno-asm` pour désactiver l'assembleur en ligne (on peut se demander pourquoi?).

Plus généralement, les bonnes options de compilation à utiliser avec gcc sur les plates-formes x86 sont

```
        gcc -O2 -fomit-frame-pointer -m386 -Wall
```

`-O2` est le bon niveau d'optimisation. Les optimisations supérieures génèrent un code un peu plus important, mais très légèrement plus rapide. De telles sur-optimisations peuvent être utiles que dans le cas d'optimisations de boucles que vous pouvez toujours réaliser en assembleur. Si vous avez besoin de faire ce genre de choses, ne le faîtes que pour les routines qui en ont besoin.

`-fomit-frame-pointer` permet au code généré de se passer de la gestion inutile des pointeurs de fenêtre, ce qui rend le code plus petit plus rapide et libère un registre pour de plus amples optimisations. Cette option exclue l'utilisation des outils de déboggage (`gdb`), mais lorsque vous les utilisez, la taille et la vitesse importent peu.

`-m386` génère un code plus compacte sans ralentissement notable, (moins de code signifie également mois d'entrées/sorties sur disque et donc une exécution plus rapide). Vous pouvez également utiliser l'option -mpentium sur la version GCC gérant l'optimisation pour ce processeur.

`-Wall` active toutes les mises-en-garde (warning) et vous évite de nombreuses erreurs stupides et évidentes.

Pour optimiser encore plus, vous pouvez utiliser l'option `-mregparm=2` et/ou les attributs de fonctions qui peuvent être utilisés mais ils peuvent dans certains cas poser de nombreux problèmes lors de l'édition de liens avec du code externe (notamment les bibliothèques partagées)...

Notez que vous pouvez ajoutez ces options aux options utilisées par défaut sur votre système en éditant le fichier `/usr/lib/gcc-lib/i486-linux/2.7.2.3/specs` (cependant, ne rajoutez pas `-Wall` à ces options).

3.2 GAS

GAS est l'assembleur GNU, utilisé par gcc.

3.2.1 Où le trouver?

Au même endroit où vous avez trouvé gcc, dans le paquetage binutils.

3.2.2 Qu'est-ce que la syntaxe AT&T

Comme GAS a été inventé pour supporter un compilateur 32 bits sous unix, il utilise la syntaxe standard "AT&T", qui ressemblent assez à l'assembleur m68k. La syntaxe n'est ni pire, ni meilleur que la syntaxe "Intel". Elle est juste différente. Lorsque vous aurez l'habitude de vous en servir, vous la trouverez plus régulière que la syntaxe Intel, quoique que légèrement plus ennuyeuse aussi.

Voici les points les plus importants à propos de la syntaxe de GAS:

- Les noms de registres sont préfixés avec `%`, de façon que les registres sont `%eax`, `%dl` et consorts au lieu de juste `eax`, `dl`, etc. Ceci rend possible l'inclusion directe de noms de symboles externes C sans risque de confusion, ou de nécessité de préfixes _.
- L'ordre des opérandes est source(s) d'abord, destination en dernier, à l'opposé de la convention d'intel consistant à mettre la destination en premier, les source(s) ensuite. Ainsi, ce qui en syntaxe intel s'écrit `mov ax,dx` (affecter au registre `ax` le contentu du registre `dx`) s'écrira en syntaxe att `mov %dx, %ax`.
- La longueur des opérandes est spécifiée comme suffixe du nom d'instruction. Le suffixe est `b` pour un octet (8 bit), `w` pour un mot (16 bit), et `l` pour un mot long (32 bit). Par exemple, la syntaxe correcte pour l'instruction ci-dessus aurait dû être `movw %dx,%ax`. Toutefois, gas n'est pas trop aussi strict que la syntaxe att l'exige, et le suffixe est optionel quand la longueur peut être devinée grâce aux opérandes qui sont des registres, la taille par défaut étant 32 bit (avec une mise en garde quand on y fait appel).
- Les opérandes immediates sont marqués d'un préfixe `$`, comme dans `addl $5,%eax` (ajouter la valeur longue immédiate 5 au registre `%eax`).
- L'absence de préfixe à une opérande indique une adresse mémoire; ainsi `movl $foo,%eax` met l'*adresse* de la variable `foo` dans le registre `%eax`, tandis que `movl foo,%eax` met le `contenu` de la variable `foo` dans le registre `%eax`.
- L'indexation ou l'indirection se fait en mettant entre parenthèses le registre d'index ou la case mémoire contenant l'indirection, comme dans `testb $0x80,17(%ebp)` (tester le bit de poids fort de l'octet au déplacement 17 après la case pointée par `%ebp`).

Un programme existe pour vous aider à convertir des programmes écrits avec la syntaxe TASM en syntaxe AT&T. Voir

`ftp://x2ftp.oulu.fi/pub/msdos/programming/convert/ta2asv08.zip`

GAS possède une documentation complète au format TeXinfo, qui est distribuée entre autre avec les sources. Vous pouvez parcourir les pages .info qui en sont extraites avec Emacs. Il y avait aussi un fichier nommé gas.doc ou as.doc disponible autour des sources de GAS, mais il a été fusionné avec la documentation TeXinfo. Bien sûr, en cas de doute, l'ultime documentation est constituée par les sources eux-mêmes! Une section qui vous intéressera particulièrement est `Machine Dependencies::i386-Dependent::`

Les sources de Linux dont un bon exemple: regardez dans le répertoire linux/arch/i386 les fichiers suivants: `kernel/*.S, boot/compressed/*.S, mathemu/*.S`

Si vous codez ce genre de chose, un paquetage de thread, etc vous devriez regarder d'autres langages (OCaml, gforth, etc), ou des paquetages sur les thread (QuickThreads, pthreads MIT, LinuxThreads, etc).

Enfin générer à partir d'un programme C du code assembleur peut vous montrer le genre d'instructions que vous voulez. Consultez la section 2 (Avez-vous besoin de l'assembleur?) au début de ce document.

3.2.3 mode 16 bits limité

GAS est un assembleur 32 bits, conçu pour assembler le code produit par un compilateur 32 bits. Il ne reconnaît que d'une manière limité le mode 16 bits du i386, en ajoutant des préfixes 32 bits aux instructions; vous écrivez donc en réalité du code 32 bits, qui s'exécute en mode 16 bits sur un processeur 32 bits. Dans les deux modes, il gère les registres 16 bits, mais pas l'adressage 16 bits. Utilisez les instructions `.code16` et `.code32` pour basculer d'un mode à l'autre. Notez que l'instruction assembleur en ligne `asm(".code16\n")` autorisera gcc à générer du code 32 bits qui fonctionnera en mode réél!

Le code nécessaire pour que GAS gère le mode 16 bits aurait été ajouté par Bryan Ford (à confirmer?). Toutefois, ce code n'est présent dans aucune distribution de GAS que j'ai essayée (jusqu'à binutils-2.8.1.x) ... plus d'informations à ce sujet seraient les bienvenues dans ce HowTo.

Une solution bon marché pour insérer quelques instructions 16-bit non reconnues pas GAS consiste à définir des macros (voir plus bas) qui produisent directement du code binaire (avec `.byte`), et ce uniquement pour les rares instructions 16 bits dont vous avez besoin (quasiment aucunes, si vous utilisez le `.code16` précédement décrit, et pouvez vous permettre de supposer que le code fonctionnera sur un processeur 32 bits). Pour obtenir le système de codage correct, vous pouvez vous inspirer des assembleurs 16 bits.

3.3 GASP

GASP est un préprocesseur pour GAS. Il ajoute des macros et une syntaxe plus souple à GAS.

3.3.1 Où trouver gasp?

gasp est livré avec gas dans le paquetage binutils GNU.

3.3.2 Comment il fonctionne?

Cela fonctionne comme un filtre, tout comme cpp et ses variantes. Je ne connais pas les détails, mais il est livré avec sa propre documentation texinfo, donc consultez-la, imprimez-la, assimilez-la. La combinaison GAS/GASP me semble être un macro-assembleur standard.

3.4 NASM

Du projet Netwide Assembler est issu encore un autre assembleur, écrit en C, qui devrait être assez modulaire pour supporter toutes les syntaxes connues et tous les formats objets existants.

3.4.1 Où trouver NASM?

`http://www.cryogen.com/Nasm`

Les versions binaires se trouvent sur votre miroir sunsite habituel dans le répertoire `devel/lang/asm/`. Il devrait également être disponible sous forme d'archive .rpm ou .deb parmi les contributions à votre distribution préférée RedHat ou Debian.

3.4.2 Son rôle

Au moment de l'écriture de ce HOWTO, NASM en est à la version 0.96.

La syntaxe est à la Intel. Une gestion de macros est intégrée.

Les formats objets reconnus sont `bin`, `aout`, `coff`, `elf`, `as86`, (DOS) `obj`, `win32`, et `rdf` (leur propre format).

NASM peut être utilisée comme assembleur pour le compilateur libre LCC.

Comme NASM évolue rapidement, ce HowTo peut ne pas être à jour à son sujet. A moins que vous n'utilisiez BCC comme compilateur 16 bit (ce qui dépasse le cadre de ce document), vous devriez utiliser NASM plutôt que AS86 ou MASM, car c'est un logiciel libre avec un excellent service après-don, qui tourne sur toutes plateformes logicielles et matérielles.

Note: NASM est également livré avec un désassembleur, NDISASM.

Son analyseur “grammatical”, écrit à la main, le rend beaucoup plus rapide que GAS; en contrepartie, il ne reconnaît qu’une architecture, en comparaison de la pléthore d’architectures reconnues par GAS. Pour les plates-formes x86, NASM semble être un choix judicieux.

3.5 AS86

AS86 est un assembleur 80x86, à la fois 16 et 32 bits, faisant partie du compilateur C de Bruce Evans (BCC). Il possède une syntaxe à la Intel.

3.5.1 Where to get AS86

Une version complètement dépassée de AS86 est diffusée par HJLu juste pour compiler le noyau Linux, dans un paquetage du nom de bin86 (actuellement version 0.4) disponible dans le répertoire GCC des sites FTP Linux. Je déconseille son utilisation pour toute autre chose que compiler Linux. Cette version ne reconnaît qu’un format de fichiers minix modifié, que ne reconnaissent ni les binutils GNU ni aucun autre produit. Il possède de plus certains bogues en mode 32 bits. Ne vous en servez donc vraiment que pour compiler Linux.

Les versions les plus récentes de Bruce Evans (bde@zeta.org.au) est diffusée avec la distribution FreeBSD. Enfin, elles l’étaient! Je n’ai pas pu trouver les sources dans la distribution 2.1. Toutefois, vous pouvez trouver les sources dans

`http:///www.eleves.ens.fr:8080/home/rideau/files/bcc-95.3.12.src.tgz`

Le projet Linux/8086 (également appelé ELKS) s’est d’une certaine manière chargée de maintenir bcc (mais je ne crois pas qu’ils aient inclus les patches 32 bits). Voir les sites `http://www.linux.org.uk/Linux8086.html`

et `ftp://linux.mit.edu/` .

Entre autres choses, ces versions plus récentes, à la différence de celle de HJLu, gèrent le format a.out de Linux; vous pouvez donc effectuer des éditions de liens avec des programmes Linux, et/ou utiliser les outils habituels provenant du paquetage binutils pour manipuler vos données. Cette version peut co-exister sans problème avec les versions précédentes (voir la question à ce sujet un peu plus loin).

La version du 12 mars 1995 de BCC ainsi que les précédentes a un problème qui provoque la génération de toutes les opérations d’empilement/dépilement de segments en 16 bits, ce qui est particulièrement ennuyant lorsque vous développez en mode 32 bits. Un patch est diffusé par le projet Tunes

`http://www.eleves.ens.fr:8080/home/rideau/Tunes/`

à partir du lien suivant: `files/tgz/tunes.0.0.0.25.src.tgz` ou dans le répertoire `LLL/i386/`.

Le patch peut également être directement récupéré sur

`http://www.eleves.ens.fr:8080/home/rideau/files/as86.bcc.patch.gz`

Bruce Evans a accepté ce patch, donc si une version plus récente de BCC existe, le patch devrait avoir été intégré...

3.5.2 Comme appeller l’assembleur?

Voici l’entrée d’un Makefile GNU pour utiliser bcc pour transformer un fichier assembleur `.s` à la fois en un objet a.out GNU `.o` et un listing `.l`:

```
%.o %.l:        %.s
        bcc -3 -G -c -A-d -A-l -A$*.l -o $*.o $<
```

Supprimez `%.l`, `-A-l`, et `-A$*.l`, si vous ne voulez pas avoir de listing. Si vous souhaitez obtenir autre chose que du a.out GNU, consultez la documentation de bcc concernant les autres formats reconnus et/ou utilisez le programme objcopy du paquetage binutils.

3.5.3 Où trouver de la documentation

Les documentations se trouvent dans le paquetage bcc. Des pages de manuel sont également disponibles quelque part sur le site de FreeBSD. Dans le doute, les sources sont assez souvent une bonne documentation: ce n'est pas très commenté mais le style de programmation est très simple. Vous pouvez essayer de voir comment as86 est utilisé dans Tunes 0.0.0.25...

3.5.4 Que faire si je ne peux plus compiler Linux avec cette nouvelle version

Linus est submergé par le courrier électronique et mon patch pour compiler Linux avec un as86 a.out n'a pas dû lui parvenir (!). Peu importe: conservez le as86 provenant du paquetage bin86 dans le répertoire /usr/bin, et laissez bcc installer le bon as86 en tant que /usr/local/libexec/i386/bcc/as comme que de droit. Vous n'aurez jamais besoin d'appeler explicitement ce dernier, car bcc se charge très bien de tout, y compris la conversion en a.out Linux, lorsqu'il est appelé avec les bonnes options. Assemblez les fichiers uniquement en passant par bcc, et non pas en appelant as86 directement.

3.6 Autres assembleurs

Il s'agit d'autres possibilités, qui sortent de la voie ordinaire, pour le cas où les solutions précédentes ne vous conviennent pas (mais je voudrais bien savoir pourquoi?), que je ne recommande pas dans les cas habituels, mais qui peuvent se montrer fort utiles si l'assembleur doit faire partie intégrante du logiciel que vous concevez (par exemple un système d'exploitation ou un environnement de développement).

3.6.1 L'assembleur de Win32Forth

Win32Forth est un système ANS FORTH 32 bit *libre* qui fonctionne sous Win32s, Win95, Win/NT. Il comprend un assembleur 32 bit libre (sous forme préfixe ou postfixe) intégrée au langage FORTH. Le traitement des macro est effectué en utilisant toute la puissance du langage réflexif FORTH. Toutefois, le seul contexte d'entrée et sortie reconnu actuellement est Win32For lui-même (aucune possibilité d'obtenir un fichier objet, mais vous pouvez toujours l'ajouter par vous-même, bien sûr). Vous pouvez trouver Win32For à l'adresse suivante:

`ftp://ftp.forth.org/pub/Forth/win32for/`

3.6.2 Terse

Terse est un outil de programmation qui fournit *LA* syntaxe assembleur la plus compacte pour la famille des processeur x86! Voir le site `http://www.terse.com` . Ce n'est cependant pas un logiciel libre. Il y aurait eu un clone libre quelque part, abandonné à la suite de mensongères allégations de droits sur la syntaxe, que je vous invite à ressusciter si la syntaxe vous intéresse.

3.6.3 Assembleurs non libres et/ou non 32 bits

Vous trouverez un peu plus d'informations sur eux, ainsi que sur les bases de la programmation assembleur sur x86, dans la FAQ de Raymond Moon pour le forum comp.lang.asm.x86. Voir http://www2.dgsys.com/~raymoon/faq/asmfaq.zip

Remarquez que tous les assembleurs DOS devraient fonctionner avec l'émulateur DOS de Linux ainsi qu'avec d'autres émulateurs du même genre. Aussi, si vous en possédez un, vous pouvez toujours l'utiliser à l'intérieur d'un vrai système d'exploitation. Les assembleurs sous DOS assez récents gèrent également les formats de fichiers objets COFF et/ou des formats gérés par la bibliothèque GNU BFD de telle manière que vous pouvez les utiliser en conjonction avec les outils 32 bits libres, en utilisant le programme GNU objcopy (du paquetage binutils) comme un filtre de conversion.

4 Méta-programmation/macro-traitement

La programmation en assembleur est particulièrement pénible si ce n'est pour certaines parties critiques des programmes.

Pour travail donné, il faut l'outil approprié; ne choisissez donc pas l'assembleur lorsqu'il ne correspond pas au problème à résoudre: C, OCAML, perl, Scheme peuvent être un meilleur choix dans la plupart des cas.

Toutefois, il y a certains cas où ces outils n'ont pas un contrôle suffisamment fin sur la machine, et où l'assembleur est utile ou nécessaire. Dans ces cas, vous apprécierez un système de programmation par macros, ou un système de méta-programmation, qui permet aux motifs répétitifs d'être factorisés chacun en une seule définition indéfiniment réutilisable. Cela permet une programmation plus sûre, une propagation automatique des modifications desdits motifs, etc. Un assembleur de base souvent ne suffit pas, même pour n'écrire que de petites routines à lier à du code C.

4.1 Description

Oui, je sais que cette partie peut manquer d'informations utiles à jour. Vous êtes libres de me faire part des découvertes que vous auriez dû faire à la dure...

4.1.1 GCC

GCC vous permet (et vous oblige) de spécifier les contraintes entre registres assembleurs et objets C, pour que le compilateur puisse interfacer le code assembleur avec le code produit par l'optimiseur. Le code assembleur en ligne est donc constitué de motifs, et pas forcément de code exact.

Et puis, vous pouvez mettre du code assembleur dans des macro-définitions de CPP ou des fonctions "en-ligne" (inline), de telle manière que tout le monde puisse les utiliser comme n'importe quelle fonction ou macro C. Les fonctions en ligne ressemblent énormément aux macros mais sont parfois plus propres à utiliser. Méfiez-vous car dans tous ces cas, le code sera dupliqué, et donc seules les étiquettes locales (comme `1:`) devraient être définies dans ce code assembleur. Toutefois, une macro devrait permettre de passer en paramètre le nom éventuellement nécessaire d'une étiquette définie non localement (ou sinon, utilisez des méthodes supplémentaires de méta-programmation). Notez également que propager du code assembleur en-ligne répandra les bogues potentiels qu'il contiendrait, aussi, faites doublement attention à donner à GCC des contraintes correctes.

Enfin, le langage C lui-même peut être considéré comme étant une bonne abstraction de la programmation assembleur, qui devrait vous éviter la plupart des difficultés de la programmation assembleur.

Méfiez-vous des optimisations consistant à passer les arguments en utilisant les registres: cela interdit aux fonctions concernées d'être appelées par des routines exterieurs (en particulier celles écrites à la main en assembleur) d'une manière standard; l'attribut asmlinkage devrait empêcher des routines données d'être concernées par de telles options d'optimisation. Voir les sources du noyau Linux pour avoir des exemples.

4.1.2 GAS

GAS a quelques menues fonctionnalité pour les macro, détaillées dans la documentation TeXinfo. De plus

J'ai entendu dire que les versions récentes en seront dotées... voir les fichiers TeXinfo). De plus, tandis que GCC reconnaît les fichiers en .s comme de l'assembleur à envoyer dans GAS, il reconnaît aussi les fichiers en .S comme devant être filtrer à travers CPP avant d'être envoyer à GAS. Au risque de me répéter, je vous convie à consulter les sources du noyau Linux.

4.1.3 GASP

Il ajoute toutes les fonctionnalités habituelles de macro à GAS. Voir sa documentation sous forme texinfo.

4.1.4 NASM

NASM possède aussi son système de macros. Consultez sa documentation. Si vous avez quelqu'idée lumineuse, contactez les auteurs, étant donné qu'ils sont en train de développer NASM activement. Pendant ce même temps, lisez la partie sur les filtres externes un peu plus loin.

4.1.5 AS86

Il possède un système simple de macros, mais je n'ai pas pu trouver de documentation. Cependant, les sources sont d'une approche particulièrement aisée, donc si vous êtes intéressé pour en savoir plus, vous devriez pouvoir les comprendre sans problème. Si vous avez besoin d'un peu plus que des bases, vous devriez utiliser un filtre externe (voir un peu plus loin).

4.1.6 Autres assembleurs

- Win32FORTH: CODE et END-CODE sont des macros qui ne basculent pas du mode interprétation au mode compilation; vous aurez donc accès à toute la puissance du FORTH lors de l'assemblage.
- Tunes: cela ne fonctionne pas encore, mais le langage Scheme est un langage de très haut niveau qui permet une méta-programmation arbitraire.

4.2 Filtres externes

Quelque soit la gestion des macros de votre assembleur, ou quelque soit le langage que vous utilisez (même le C), si le langage n'est pas assez expressif pour vous, vous pouvez faire passer vos fichier à travers un filtre externe grâce à une règle comme suit dans votre Makefile:

```
%.s:    %.S autres_dépendances
        $(FILTER) $(FILTER_OPTIONS) < $< > $@
```

4.2.1 CPP

CPP n'est vraiment pas très expressif, mais il suffit pour les choses faciles, et il est appelé d'une manière transparente par GCC.

Comme exemple de limitation, vous ne pouvez pas déclarer d'objet de façon à ce qu'un destructeur soit automatiquement appelé à la fin du bloc ayant déclaré l'objet. Vous n'avez pas de diversions ou de gestion de portée des variables, etc.

CPP est livré avec tout compilateur C. Si vous pouvez faire sans, n'allez pas chercher CPP (bien que je me demande comment vous pouvez faire).

4.2.2 M4

M4 vous donne la pleine puissance du macro-traitement, avec un langage Turing-équivalent, récursivité, expressions régulières, etc. Vous pouvez faire avec tout ce que cpp ne peut faire.

Voir macro4th/This4th que l'on trouve sur

`ftp://ftp.forth.org/pub/Forth/` dans Reviewed/ ANS/ (?), ou les sources de Tunes 0.0.0.25 comme exemple de programmation avancée en utilisant m4.

Toutefois, le système de citation est très pénible à utiliser et vous oblige à utiliser un style de programmation par fonctions récursives avec passage explicite de continuation (CPS) pour toute programmation *avancée* (ce qui n'est pas sans rappeler à TeX – au fait quelqu'un a-t-il déjà essayé d'utiliser TeX comme macro-processeur pour autre chose que de la mise-en-page?). Toutefois, ce n'est pas pire que cpp qui ne permet ni citation ni récursivité.

La bonne version de m4 à récupérer est GNU m4 1.4 (ou ultérieure si elle existe). C'est celle qui contient le plus de fonctionnalité et le moins de bogues ou de limitations. m4 est conçu pour être intrinsèquement lent pour toute utilisation sauf la plus simple; cela suffit sans aucun doute pour la plupart des programmes en assembleur (vous n'allez quand même pas écrire des millions de lignes en assembleur, si?).

4.2.3 Macro-traitement avec votre propre filtre

Vous pouvez écrire votre propre programme d'expansion de macro avec les outils courants comme perl, awk, sed, etc. C'est assez rapide à faire et vous pouvez tout contrôler. Mais bien toute puissance dans le macro-traitement doit se gagner à la dure.

4.2.4 Méta-programmation

Plutôt que d'utiliser un filtre externe qui effectue l'expansion des macros, une manière de réaliser cela est d'écrire des programmes qui écrivent d'autres programmes, en partie ou en totalité.

Par exemple, vous pourriez utiliser un programme générant du code source

- pour créer des tables de sinus/cosinus (ou autre),
- pour décompiler un fichier binaire en source annoté annoté,
- pour compiler vos bitmaps en des routines d'affichage rapides,
- pour extraire de la documentation, du code d'initilisation ou finalisation, des tables de descriptions, aussi bien que du code normal depuis les mêmes fichiers sources;
- pour utiliser une technique spécifique de production de code, produite avec un script perl/shell/scheme
- pour propager des données définies en une seule fois dans de nombreux morceaux de code ou tables avec références croisées.
- etc.

Pensez-y!

Backends provenant de compilateur existants Des compilateurs comme SML/NJ, Objective CAML, MIT-Scheme, etc, ont leur propre générateur de code assembleur, que vous pouvez ou non utiliser, si vous souhaitez générer du code semi-automatiquement depuis les langages correspondants.

Le New-Jersey Machine-Code Toolkit Il s'agit projet utilisant le langage de programmation Icon pour bâtir une base de code de manipulation d'assembleur. Voir `http://www.cs.virginia.edu/~nr/toolkit/`

Tunes Le projet de système d'exploitation OS développe son propre assembleur comme étant une extension du langage Scheme. Il ne fonctionne pas encore totalement, de l'aide est bienvenue.

L'assembleur manipule des arbres de syntaxes symboliques, de telle manière qu'il puisse servir comme base d'un traducteur de syntaxe assembleur, un désassembleur, l'assembleur d'un compilateur, etc. Le fait qu'il utile un vrai langage de programmation puissant comme Scheme le rend imbatable pour le macro-traitement et pour la méta-programmation.

`http://www.eleves.ens.fr:8080/home/rideau/Tunes/`

5 Conventions d'appel

5.1 Linux

5.1.1 Edition de liens avec GCC

C'est la solution la plus pratique. Consultez la documentation de gcc et prenez exemple sur les sources du noyau Linux (fichiers `.S` qui sont utilisés avec gas, non pas as86).

Les arguments 32 bits sont empilés dans la pile vers le bas dans l'ordre inverse de l'ordre syntaxique (c'est-à-dire qu'on accède aux arguments ou les dépile dans l'ordre syntaxique), au-dessus de l'adresse de retour 32 bits. `%ebp`, `%esi`, `%edi`, `%ebx` doivent être conservés par l'appelé, les autres registres peuvent être détruits; `%eax` doit contenir le résultat, ou `%edx:%eax` pour des résultats sur 64 bits.

Pile virgule flottante: je ne suis pas sûr, mais je pense que le résultat se trouve dans `st(0)`, la pile étant à la discrétion de l'appelé.

Notez que GCC possède certaines options pour modifier les conventions d'appel en réservant certains registres, en mettant les arguments dans des registres, en supposant que l'on ne possède pas de FPU, etc. Consultez les pages .info concernant le i386.

Il faut prendre garde à déclarer l'attribut `cdecl` pour une fonction qui suit la convention standard GCC (je ne sais pas exactement ce que cela produit avec des conventions modifiées). Consultez la documentation GCC dans la section: `C Extensions::Extended Asm::`

5.1.2 Problèmes ELF et a.out

Certains compilateurs C ajoutent un underscore avant tout symbole, alors que d'autres ne le font pas.

En particulier, la version GCC a.out effectue ce genre d'ajouts, alors que la version ELF ne le fait pas.

Si vous êtes confronté à ce problème, regardez comment des paquetages existants traitent le problèmes. Par exemple, récupérer une ancienne arborescence des sources de Linux, Elk, les qthreads ou OCAML...

Vous pouvez également redéfinir le renommage implicite de C en assembleur en ajoutant les instructions suivantes:

```
void truc asm("machin") (void);
```

pour s'assurer que la fonction C truc sera réellement appelée machin en assembleur.

Remarquez que l'outil `objcopy`, du paquetage `binutils`, devrait vous permettre de transformer vos fichiers objets a.out en objets ELF et peut-être inversement dans certains cas. D'une manière plus générale, il vous permet d'effectuer de nombreuses conversions de formats de fichiers.

5.1.3 Appels systèmes directs

Il n'est absolument pas recommandé d'effectuer de tels appels par ce que leurs conventions peuvent changer de temps en temps, ou d'un type de noyau à un autre (cf L4Linux), de plus, ce n'est pas portable, difficile à écrire, redondant avec l'effort entrepris par libc, et enfin, cela empêche les corrections et les extensions effectuées à travers la libc, comme par exemple avec le programme `zlibc` qui réalise une décompression à la volée de fichiers compressés avec gzip. La manière standard et recommendée d'effectuer des appels systèmes est et restera de passer par la libc.

Les objets partagés devraient réduire l'occupation mémoire des programmes, et si vous souhaitez absolument avoir de petits exécutables, utilisez `#!` avec un interpréteur qui contiendra tout ce que vous ne voulez pas mettre dans vos binaires.

Maintenant, si pour certaines raisons, vous ne souhaitez pas effectuer une édition des liens avec la libc, récupérez-la et essayez de comprendre comment elle fonctionne! Après tout, vous prétendez bien la remplacer non?

Vous pouvez aussi regarder comment

eforth 1.0c `ftp://ftp.forth.org/pub/Forth/Linux/linux-eforth-1.0c.tgz` le fait.

Les sources de Linux sont fort utiles, en particulier le fichier d'en-tête asm/unistd.h qui décrit comment sont effectués les appels système...

Le principe général est d'utiliser l'instruction `int $0x80` avec le numéro de l'appel système `__NR_`machin (regarder dans `asm/unistd.h`) dans `%eax`, et les paramètres (jusqu'à cinq) dans `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`. Le résultat est renvoyé dans `%eax` avec un résultat négatif étant l'erreur dont l'opposé est tranféré par la libc dans errno. La pile utilisateur n'est pas modificée donc n'avez pas besoin d'en avoir une correcte lors de l'appel.

5.1.4 Entrées/sorties sous Linux

Si vous souhaitez effectuer des entrées/sorties directement sous Linux, soit il s'agit de quelque chose de très simple qui n'a pas besoin de spécificités du système et dans ce cas là, consultez le mini-HOWTO `IO-Port-Programming`, ou alors vous devez créer un nouveau gestionnaire de périphérique et vous devriez alors lire quelques documents sur les méandres du noyau, le développement de gestionnaires de périphériques, les modules du noyau, etc. Vous trouverez d'excellents HOWTO ou autres documents du projet LDP.

Plus particulièrement, si vous souhaitez réaliser des programmes graphiques, rejoignez le projet GGI:

`http://synergy.caltech.edu/~ggi/`

`http://sunserver1.rz.uni-duesseldorf.de/~becka/doc/scrdrv.html`

Dans tous les cas, vous devriez plutôt utiliser l'assembleur en ligne de GCC avec les macros provenant des fichiers linux/asm/*.h que d'écrire des sources en assembleur pur.

5.1.5 Accéder aux gestionnaires 16 bits avec Linux/i386

De telles choses sont théoriquement possibles (preuve: voir comment DOSEMU permet à des programmes d'accéder au port série), et j'ai entendu des rumeurs que certaines personnes le font (avec le gestionnaire PCI? Accès aux cartes VESA? PnP ISA? Je ne sais pas). Si vous avez de plus amples précisions à ce sujet, soyez les bienvenus. Le bon endroit à regarder est les sources du noyau, les sources de DOSEMU (et des autres programmes se trouvant dans

le répertoire DOSEMU `ftp://tsx-11.mit.edu/pub/linux/ALPHA/dosemu/`), ainsi que les sources d'autres programmes bas niveaux (peut-être GGI s'il gère les cartes VESA).

En fait, vous devez utiliser soit le mode protégé 16 bits, soit le mode vm86.

Le premier est plus simple à configurer mais il ne fonctionne qu'avec du code ayant un comportement propre qui n'effectue pas d'arithmétique de segments ou d'adressage absolu de segment (en particulier pour l'adressage du segment 0), à moins que par chance tous les segments utilisés peuvent être configuré à l'avance dans le LDT.

La seconde possiblité permet d'être plus "compatibles" avec les environnements 16 bits mais il nécessite une gestion bien plus compliquée.

Dans les deux cas, avant de sauter sur le code 16 bits, vous devez:

- mmapper toute adresse absolue utilisée dans le code 16 bits (comme la ROM, les tampons vidéo, les adresses DMA et les entrées/sorties passant des zones de mémoires mappées) à partir de /dev/mem dans votre espace d'adressage de votre processus.
- configurer le LDT et/ou le moniteur en mode vm86.
- demander au noyau les droits d'accès nécessaires pour les entrées/sorties (voir plus haut).

Encore une fois, lisez attentivement les codes sources situés dans le répertoire de DOSEMU et consorts, en particulier ces mini-émulateurs permettant de faire tourner des programmes ELKS et/ou des .COM assez simples sous Linux/i386.

5.2 DOS

La plupart des émulateurs DOS sont livrés avec certaines interfaces d'accès aux services DOS. Lisez leur documentation à ce sujet, mais bien souvent, ils ne font que simuler `int $0x21` et ainsi de suite, donc c'est comme si vous étiez en mode réel (je doute qu'ils aient de possibilités de fonctionner avec des opérandes 32 bits: ils ne font que réfléchir l'interruption dans le mode réel ou dans le gestionnaire vm86).

Certaines documentations concernant DPMI (ou ses variantes peuvent) être trouvées sur

`ftp://x2ftp.oulu.fi/pub/msdos/programming/`

DJGPP est livré avec son propre sous-ensemble, dérivé, ou remplacement (limité) de la glibc.

Il est possible d'effectuer une compilation croisée de Linux vers DOS. Consultez le répertoire devel/msdos/ de votre miroir FTP de sunsite.unc.edu. Voir également le dos-extender MOSS du projet Flux d'utah.

D'autres documentations et FAQ sont plus consacrés à DOS. Nous déconseillons le développement sous DOS.

5.3 Windauberies...

Heu, ce document ne traite que de libre logiciel. Téléphonez-moi lorsque Windaube le deviendra ou du moins ses outils de développement!

En fait, après tout, cela existe:

Cygnus Solutions `http://www.cygnus.com`

a développé la bibliothèque cygwin32.dll pour que les programmes GNU puissent fonctionner sur les machines MicroMerdiques. Donc, vous pouvez utiliser GCC, GAS et tous les outils GNU ainsi que bon nombre d'applications Unix. Consultez leur site Web. Je (Faré) ne souhaite pas m'étendre sur la programmation sous Windaube, mais je suis sûr que vous trouverez tout un tas d'informations partout...

5.4 Votre propre système d'exploitation

Le contrôle sur le système étant ce qui attire de nombreux programmeurs vers l'assembleur, une prémisse ou un corollaire naturel de son utilisation est la volonté de développer son propre système d'exploitation. Remarquons tout d'abord que tout système permettant son auto-développement pourrait être qualifié de système d'exploitation, combien même tournerait-il au-dessus d'un autre système sur lequel il se déchargerait de la gestion du multitâche (Linux sur Mach) ou des entrées/sorties (OpenGenera sur Digital Unix), etc. Donc, pour simplifier le débogage, vous pouvez souhaiter développer votre système d'exploitation comme étant un processus fonctionnant sous Linux (au prix d'un certain ralentissement), puis, utiliser le

Flux OS kit `http://ww.cs.utah.edu/projects/flux/`

(qui permet l'utilisation des drivers Linux et BSD dans votre propre système d'exploitation) pour le rendre indépendant. Lorsque votre système est stable, il est toujours temps d'écrire vos propres gestionnaires de matériels si c'est vraiment votre passion.

Ce HowTo ne couvrira pas des sujets comme le code de chargement du système, le passage en mode 32 bits, la gestion des interruptions, les bases concernant les horreurs des processeurs Intel (mode protégé, V86/R86), la définition de votre format d'objets ou de vos conventions d'appel. L'endroit où vous pourrez trouver le plus d'informations concernant tous ces sujets est le code source de système déjà existants.

Un grand nombre de pointeurs se trouvent dans la page:

`http://www.eleves.ens.fr:8080/home/rideau/Tunes/Review/OSes.html`

6 A faire et pointeurs

- compléter les sections incomplètes;
- ajouter des pointeurs sur des programmes et des documentations;
- ajouter des exemples de tous les jours pour illustrer la syntaxe, la puissance et les limitation de chacune des solutions proposées;
- demander aux gens de me donner un coup de main;
- trouver quelqu'un qui a assez de temps pour prendre en charge la maintenance de ce HOWTO;
- peut-être dire quelques mots sur l'assembleur d'autres plates-formes?
- Quelques pointeurs (en plus de ceux qui se trouvent dans ce document)
  - *pages de manuel pour pentium* `http://www.intel.com/design/pentium/manuals/`
  - *hornet.eng.ufl.edu pour les codages assembleurs* `http://www.eng.ufl.edu/ftp`
  - *ftp.luth.se* `ftp://ftp.luth.se/pub/msdos/demos/code/`
  - *PM FAQ* `ftp://zfja-gate.fuw.edu.pl/cpu/protect.mod`

- *Page Assembleur 80x86* `http://www.fys.ruu.nl/~faber/Amain.html`
- *Courseware* `http://www.cit.ac.nz/smac/csware.htm`
- *programmation de jeux* `http://www.ee.ucl.ac.uk/~phart/gameprog.html`
- *experiences de programmation sous Linux exclusivement en assembleur* `http://bewoner.dma.be/JanW`

• Et bien sur, utilisez vos outils habituels de recherche sur Internet pour trouver les informations. Merci de m'envoyer tout ce que vous trouvez d'interessant.

Signature de l'auteur:

```
--    ,                                          ,           _ v    ~  ^  --
--
-- Fare -- rideau@clipper.ens.fr -- Francois-Rene Rideau -- +)ang-Vu Ban --
--                                       '                   / .          --
Join the TUNES project for a computing system based on computing freedom!
                 TUNES is a Useful, Not Expedient System
WWW page at URL: http://www.eleves.ens.fr:8080/home/rideau/Tunes/
```