Les outils de développement sous FreeBSD : Guide de l'utilisateur

James Raynard

jraynard@freebsd.org

17 Août 1997

Ce document est une introduction à l'utilisation de quelques-uns des outils de programmation fournis avec FreeBSD, quoique l'essentiel reste aussi valable pour de nombreuses autres versions d'Unix. Il ne cherche *pas* à expliquer en détail comment coder. La plus grande partie du document suppose que vous n'avez aucune ou peu de notions préalables de programmation, bien que l'on espère que la plupart des programmeurs y trouveront quelque chose qui leur sera utile.

Version française de Frédéric Haby <`frederic.haby@mail.dotcom.fr`>.

1. Introduction

FreeBSD fournit un excellent environnement de développement. Le système de base comprend des compilateurs C, C++ et Fortran, et un assembleur, pour ne pas mentionner l'interpréteur Perl et les outils Unix classiques comme `sed` et `awk`. Si cela ne vous suffit pas, il y a beaucoup d'autres compilateurs et interpréteurs au catalogue des logiciels portés. FreeBSD est très largement compatible avec les standards comme POSIX et ANSI C, de même qu'avec son propre héritage BSD, il est donc possible d'écrire des applications qui compilent et s'exécutent sur une grande variété de plates-formes.

Toute cette puissance, toutefois, peut submerger au premier abord, si vous n'avez jamais auparavant écrit de programme sur une plate-forme Unix. Ce document vise à vous aider à vous y mettre, sans approfondir trop les

questions les plus avancées. L'intention est de vous fournir suffisamment de bases pour vous permettre de tirer ensuite profit de la documentation.

La plus grande partie du document ne demande aucune ou peu de connaissance de la programmation mais suppose une compétence de base dans l'utilisation d'Unix et la volonté d'apprendre!

2. Introduction à la programmation

Un programme est une série d'instructions qui dit à l'ordinateur de faire des choses diverses; l'instruction qu'il doit exécuter dépend parfois de ce qui s'est passé lorsqu'il a exécuté une instruction précédente. Cette section vous donne un aperçu des deux principales méthodes pour transmettre ces instructions, ou "commandes" comme on les appellent. L'une est d'utiliser un *interpréteur*, l'autre de se servir d'un *compilateur*. Comme les langues humaines sont trop compliquées pour être comprises sans ambiguïté par un ordinateur, les commandes sont généralement écrites dans l'un ou l'autre des languages spécialement conçus à cet effet.

2.1. Interpréteurs

Dans le cas d'un interpréteur, le langage s'accompagne d'un environnement, sous lequel vous tapez des commandes à son invite et qui les exécute pour vous. Pour des programmes plus compliqués, vous pouvez saisir les commandes dans un fichier et le faire charger et exécuter les commandes qu'il contient par l'interpréteur. Si quelque chose se passe mal, la plupart des interpréteurs passeront le contrôle à un débogueur pour vous aider à trouver l'origine du problème.

Cela a l'avantage de vous permettre de voir immédiatement le résultat de vos commandes et de corriger sur le champ vos erreurs. Le principal inconvénient survient lorsque vous voulez partager vos programmes avec d'autres. Il faut qu'ils aient le même interpréteur que vous ou que vous ayez le moyen de leur fournir cet interpréteur; il faut aussi qu'ils comprennent comment s'en servir. Les utilisateurs peuvent aussi ne pas apprécier de se retrouver sous un débogueur s'ils appuient sur la mauvaise touche! Du point de vue de la performance, les interpréteurs utilisent parfois beaucoup de mémoire et ne génèrent habituellement pas le code aussi efficacement que les compilateurs.

A mon avis, les langages interprétés sont le meilleur moyen de débuter si vous n'avez jamais programmé auparavant. On trouve typiquement ce genre d'environnement avec des langages tels que Lisp, Smalltalk, Perl et Basic. On peut aussi avancer que le *shell* Unix est lui-même un interpréteur, beaucoup écrivent en fait avec des procédures - *scripts* - pour se faciliter le travail d'administration de leur machine. De fait, une partie de la philosophie d'origine d'Unix était de fournir nombre de petits programmes utilitaires qui puissent être utilisés de concert dans des procédures pour effectuer des tâches utiles.

2.2. Interpréteurs disponibles pour FreeBSD

Voici une liste des interpréteurs disponibles sous forme de "paquetages" (ftp://ftp.freebsd.org/pub/FreeBSD/packages/) FreeBSD, accompagnée d'une brève description des langages interprétés les plus répandus.

Pour vous procurer l'un de ces "paquetages", il vous suffit de cliquer sur le lien correspondant et d'exécuter ensuite:

```
# pkg_add nom_du_paquetage
```

sous le compte super-utilisateur `root`. Il faut bien évidemment que vous ayez un système FreeBSD 2.1.0 ou ultérieur en état de marche pour que le logiciel fonctionne.

BASIC

Abréviation pour "*Beginner's All-purpose Symbolic Instruction Code*" - code d'instructions symbolique universel pour les débutants. Développé dans les années 50 pour apprendre la programmation aux étudiants des Universités et fourni avec tout ordinateur personnel qui se respectait dans les années 80, BASIC a été le premier langage pour de nombreux programmeurs. C'est aussi la base de Visual Basic™.

L'interpréteur Basic Bywater (ftp://ftp.freebsd.org/pub/FreeBSD/packages/lang/bwbasic-2.10.tgz) et l'interpréteur Basic de Phil Cockroft (ftp://ftp.freebsd.org/pub/FreeBSD/packages/lang/pbasic-2.0.tgz) (appelé auparavant "Rabbit Basic") sont disponibles sous forme de "paquetages" FreeBSD (ftp://ftp.freebsd.org/pub/FreeBSD/packages/).

Lisp

Un langage développé à la fin des années 1950 comme alternative aux langages "dévoreurs de nombres" qui e´taient populaires à l'époque. Au lieu d'être basé sur les nombres, Lisp repose sur les listes; de fait, son nom est une abréviation pour "*List Processing*" - traitement de listes. Très répandu dans les milieux de l'IA (Intelligence Artificielle).

Lisp est un langage très puissant et sophistiqué, mais peut être assez lourd et bavard.

FreeBSD dispose de GNU Common Lisp (ftp://ftp.freebsd.org/pub/FreeBSD/packages/gcl-2.0.tgz) sous forme de "paquetage".

Perl

Très employé par les administrateurs système pour écrire des procédures; et souvent aussi sur les serveurs *World Wide Web* pour écrire des procédures CGI.

La Version 4, qui est probablement encore la version la plus largement répandue est fournie avec FreeBSD; le plus récent Perl Version 5 (ftp://ftp.freebsd.org/pub/FreeBSD/packages/lang/perl-5.001.tgz) est disponible sous forme de "paquetage".

Scheme

Un dialecte de Lisp qui est plutôt plus compact et plus propre que Common Lisp. Courant dans les Universités parce qu'il est assez facile à enseigner en premier cycle comme langage d'initiation et présente un niveau d'abstraction suffisant pour être utilisé pour du travail de recherche.

FreeBSD offre en "paquetages" l'Interpréteur Scheme Elk (ftp://ftp.freebsd.org/pub/FreeBSD/packages/lang/elk-3.0.tgz), l'Interpréteur Scheme du MIT (ftp://ftp.freebsd.org/pub/FreeBSD/packages/lang/mit-scheme-7.3.tgz) et l'Interpréteur Scheme SCM (ftp://ftp.freebsd.org/pub/FreeBSD/packages/lang/scm-4e1.tgz).

Icon

Le langage de programmation Icon (ftp://ftp.freebsd.org/pub/FreeBSD/packages/lang/icon-9.0.tgz).

Logo

L'interpréteur LOGO de Brian Harvey (ftp://ftp.freebsd.org:pub/FreeBSD/packages/lang/ucblogo-3.3.tgz).

Python

Le langage de programmation orienté objet Python (ftp://ftp.freebsd.org/pub/FreeBSD/packages/lang/python-1.2).

2.3. Compilateurs

Les compilateurs sont assez différents. Vous écrivez d'abord votre code dans un(des) fichiers(s) à l'aide d'un éditeur. Vous exécutez ensuite le compilateur et voyez s'il valide votre programme. S'il ne le compile pas, serrez les dents et retournez sous l'éditeur. S'il le compile et vous en fait un programme, vous pouvez l'utiliser soit à l'invite de l'interpréteur de commande, soit en vous servant d'un débogueur pour voir s'il fonctionne correctement[1].

Ce n'est évidemment pas aussi immédiat que de se servir d'un interpréteur. Cela vous permet cependant de faire beaucoup de choses qui sont très difficiles ou même irréalisables avec un interpréteur, comme écrire du code qui interagisse étroitement avec le système d'exploitation - ou même écrire votre propre système d'exploitation! C'est aussi utile si vous avez besoin d'écrire du code très efficace, parce que le compilateur peut prendre son temps et optimiser le code, ce qui ne serait pas acceptable d'un interpréteur. Distribuer un programme écrit pour un compilateur est généralement plus facile - il suffit de livrer une copie de l'exécutable, en supposant que les destinataires aient le même système d'exploitation que vous.

Les langages compilés incluent Pascal, C et C++. C et C++ sont des langages qui pardonnent assez peu, et plus adaptés aux programmeurs plus expérimentés. Pascal, d'un autre côté, a été conçu pour l'enseignement, et est un assez bon langage avec lequel commencer. Malheureusement, FreeBSD n'a aucun support pour Pascal, à l'exception d'un convertisseur de Pascal en C, au catalogue des logiciels portés.

Comme le cycle "édition-compilation-exécution-débogage" est assez fastidieux, de nombreux fournisseurs de compilateurs commerciaux ont produit des Environnements de Développement Intégrés (EDI en abrégé). FreeBSD ne dispose pas d'EDI en tant que tel; il est cependant possible d'utiliser Emacs à cet effet. C'est expliqué à la section Utiliser Emacs comme environnement de développement.

3. Compiler avec `cc`

Cette section ne s'occupe que du compilateur GNU pour C et C++, qui est fourni de base avec le système FreeBSD. Il peut être appelé soit avec la commande `cc`, soit avec `gcc`. Les détails de la réalisation d'un programme avec un interpréteur varient considérablement d'un interpréteur à l'autre, est sont généralement bien décrits par l'aide en ligne de l'interpréteur.

Une fois que vous avez écrit votre chef-d'oeuvre, l'étape suivante consiste à le convertir en quelque chose qui (espérons-le!) s'exécutera sous FreeBSD. Cela demande habituellement plusieurs opérations successives, dont chacune est confiée à un programme différent.

1. Pré-processer votre code source pour en éliminer les commentaires et faire diverses autres choses, comme la substitution des macros-instructions en C.
2. Vérifier la syntaxe de votre code pour s'assurer que vous avez respecté les règles du langage. Si ce n'est pas le cas, il rouspétera.
3. Convertir le code source en langage assembleur - c'est très proche du code machine, mais encore compréhensible par des êtres humains. C'est du moins ce que l'on prétend[2].

4. Convertir le langage assembleur en code machine - oui, nous parlons ici de bits et d'octets, de zéros et de uns.
5. Vérifier que vous avez utilisé de façon cohérente les fonctions et les variables globales. Si, par exemple, vous avez appelé une fonction qui n'existe pas, il s'en plaindra.
6. Si vous essayez de générer un programme à partir de plusieurs fichiers de source, faire ce qu'il faut pour les regrouper.
7. S'arranger pour produire quelque chose que le chargeur de programmes du système pourra mettre en mémoire et exécuter.

Le mot *compiler* est souvent utilisé pour ne désigner que les étapes 1 à 4 - les autres sont appelées *édition de liens*. L'étape 1 est parfois appelée *pré-processer* et les étapes 3-4 *assembler*.

Heureusement, pratiquement tous ces détails vous sont transparents, car la commande `cc` est une interface qui gère pour vous l'appel de ces différents programmes avec les bons arguments; taper simplement:

```
% cc foobar.c
```

effectue la compilation de `foobar.c` en passant par toutes les étapes décrites ci-dessus. Si vous avez à compiler plus d'un fichier, faites simplement quelque chose comme:

```
% cc foo.c bar.c
```

Notez que la vérification syntaxique ne consiste qu'en cela: vérifier la syntaxe. Il n'y aura pas de contrôle sur les erreurs logiques que vous auriez commises, comme faire exécuter au programme une boucle infinie ou utiliser un tri à bulles au lieu d'un tri par arbre binaire[3].

`cc` dispose d'une quantité d'options, qui sont toutes décrites dans les pages de manuel. En voici quelques-unes des plus importantes, et la façon de les utiliser.

`-o` *`nom_de_fichier`*

Le nom du fichier résultat. Si vous n'utilisez pas cette option, `cc` générera un exécutable appelé `a.out`[4].

```
% cc foobar.c                 l'exécutable est a.out
% cc -o foobar foobar.c       l'exécutable est foobar
```

`-c`

Uniquement compiler le fichier, ne pas faire l'édition de liens. Utile pour des programmes d'essai, quand vous voulez simplement vérifier la syntaxe, ou si vous vous servez d'un `Makefile`.

```
% cc -c foobar.c
```

Cela générera un *fichier objet* (et non un exécutable) appelé `foobar.o`. Il pourra être lié avec d'autres fichiers objet pour constituer un exécutable.

`-g`

Crée une version débogable de l'exécutable Le compilateur inclut alors dans l'exécutable des informations de correspondance entre les numéros de ligne du fichier source et les fonctions appelées. Un débogueur peut alors utiliser ces informations pour vous afficher le code source tandis que vous exécutez pas à pas le programme, ce qui est *très* utile; l'inconvénient est que toutes ces informations augmentent considérablement la taille du

programme. Normalement, vous compilez avec `-g` quand vous développez le programme, et compilez ensuite une "version de livraison" quand vous êtes satisfait parce qu'il fonctionne correctement.

```
% cc -g foobar.c
```

Cela produira une version débogable du programme[5].

`-O`

Génère une version optimisée de l'exécutable. Le compilateur effectue alors diverses opérations bien pensées pour essayer de construire un programme qui aille plus vite que normalement. Vous pouvez faire suivre `-O` d'un nombre pour demander un degré plus important d'optimisation, mais cela met souvent en évidence des bogues dans l'optimiseur du compilateur. Par exemple, on sait que la version de `cc` de FreeBSD 2.1.0 produit du code incorrect avec l'option `-O2` dans certaines circonstances.

On n'active en général l'optimisation qu'à la compilation de la version de livraison.

```
% cc -O -o foobar foobar.c
```

Cela construira une version optimisée de `foobar`.

Les trois indicateurs suivants demanderont à `cc` de vérifier que votre code est conforme à la norme internationale, souvent appelée norme ANSI, bien que ce soit à proprement parler une norme ISO.

`-Wall`

Active tous les messages d'avertissement que les auteurs du compilateur `cc` ont jugés intéressants. Malgré son nom ("*all*" - tous), cela n'active pas tous les messages d'avertissement dont le compilateur est capable.

`-ansi`

Désactive la plupart, mais pas toutes, les possibilités non-ANSI C fournies par `cc`. Malgré son nom, cela ne garantit pas absolument que votre code soit conforme à la norme.

`-pedantic`

Désactive *toutes* les possibilités non-ANSI C de `cc`.

Sans ces indicateurs, `cc` vous permet d'utiliser ses extensions non-standard de la norme. Quelques-unes sont très utiles, mais ne se retrouveront pas sur d'autres compilateurs - de fait, l'un des objectifs principaux de la norme est de permettre l'écriture de code qui puissent être réutilisé avec n'importe quel compilateur sur n'importe quel système. C'est cela que l'on appelle du *code portable*.

En général, vous devriez vous efforcer de rendre votre code aussi portable que possible, sans quoi vous risquez de devoir récrire entièrement votre programme par la suite pour qu'il fonctionne ailleurs - et qui peut dire ce que vous utiliserez dans quelques années?

```
% cc -Wall -ansi -pedantic -o foobar foobar.c
```

Cela générera un exécutable `foobar` après avoir vérifié que `foobar.c` respecte la norme.

`-l`*bibliothèque*

Définit une bibliothèque de fonctions à utiliser pour l'édition de liens.

L'exemple le plus courant est la compilation d'un programme qui utilise certaines des fonctions mathématiques de C. A l'inverse de la plupart des autres plates-formes, ces fonctions sont dans une bibliothèque différente de la bibliothèque C standard et vous devez préciser au compilateur qu'il doit l'utiliser.

La règle est que si la bibliothèque s'appelle `lib`*`quelque_chose`*`.a`, vous donnez à `cc` l'argument `-l`*`quelque_chose`*. Par exemple, la bibliothèque mathématique s'appelle `libm.a`, vous donnez donc à `cc` l'argument `-lm`. Un détail à connaître à propos de la bibliothèque mathématique est que ce doit généralement être la dernière sur la ligne de commande.

```
% cc -o foobar foobar.c -lm
```

Cela ajoutera à l'édition de liens de `foobar` des fonctions de la bibliothèque mathématique.

Si vous compilez du code C++, vous devrez ajouter `-lg++`, ou `-lstdc++` si vous utilisez la version 2.2 de FreeBSD ou une version ultérieure, à la ligne de commande pour éditer les liens avec les fonctions de la bibliothèque C++. Au lieu de cela, vous pouvez utiliser la commande `c++` au lieu de `cc`, qui fera la même chose à votre place. Sous FreeBSD, `c++` peut aussi être appelé avec `g++`.

```
% cc -o foobar foobar.cc -lg++       Avec FreeBSD 2.1.6 et antérieurs
% cc -o foobar foobar.cc -lstdc++    Avec FreeBSD 2.2 et ultérieurs
% c++ -o foobar foobar.cc
```

Chacun de ces exemples construira un exécutable `foobar` à partir du fichier source C++ `foobar.cc`. Remarquez que, sur les systèmes Unix, les fichiers sources C++ ont traditionnellement l'extension `.C`, `.cxx` ou `.cc`, plutôt que l'extension `.cpp` de sytle MS-DOS™ (qui est déjà utilisée pour autre chose). `gcc` se fiait autrefois à l'extension pour savoir quel type de compilateur utiliser avec le fichier source, mais cette restriction ne s'applique plus, vous pouvez donc appeler vos fichiers C++ `.cpp` en toute impunité!

3.1. Questions et problèmes `cc`

Q. J'essaie d'écrire un programme qui utilise la fonction `sin()` et j'obtiens une erreur qui ressemble à ce qui suit. Qu'est-ce que cela veut dire?

```
/var/tmp/cc0143941.o: Undefined symbol `_sin' referenced from text segment
```

R. Quand vous utilisez des fonctions mathématiques telles que `sin()`, vous devez dire à `cc` d'inclure la bibliothèque mathématique à l'édition de liens, comme ceci:

```
% cc -o foobar foobar.c -lm
```

Q. D'accord, j'ai écrit ce petit programme pour m'entraîner à utiliser `-lm`. Il ne fait que calculer 2.1 à la puissance 6:

```
#include <stdio.h>

int main() {
      float f;

      f = pow(2.1, 6);
```

```
        printf("2.1 ^ 6 = %f\n", f);
        return 0;
}
```

et l'ai compilé comme ceci:

```
% cc temp.c -lm
```

comme vous avez dit qu'il fallait le faire, mais voilà ce que j'obtiens à l'exécution:

```
% ./a.out
2.1 ^ 6 = 1023.000000
```

Ce n'est *pas* la bonne réponse! Que se passe-t-il?

R. Quand le compilateur voit que vous appelez une fonction, il regarde s'il en a déjà vu un prototype. Si ce n'est pas le cas, il suppose que la fonction retourne un int - *entier*, ce qui n'est évidemment pas ce que vous souhaitez dans ce cas.

Q. Comment alors régler ce problème?

R. Les prototypes des fonctions mathématiques sont dans `math.h`. Si vous incluez ce fichier, le compilateur trouvera le prototype et cessera de vous fournir un résultat bizarre!

```
#include <math.h>
#include <stdio.h>

int main() {
...
```

Après l'avoir recompilé de la même façon qu'auparavant, exécutez-le:

```
% ./a.out
2.1 ^ 6 = 85.766121
```

Si vous utilisez la moindre fonction mathématique, incluez *toujours* `math.h` et n'oubliez pas d'utiliser la bibliothèque mathématique à l'édition de liens.

Q. J'ai compilé un fichier appelé `foobar.c` et je ne trouve pas d'exécutable appelé `foobar`. Où est-il passé?

R. N'oubliez pas, `cc` appellera l'exécutable `a.out` à moins que vous ne lui disiez de faire autrement. Utilisez l'option `-o` *`nom_de_fichier`*:

```
% cc -o foobar foobar.c
```

Q. OK, si j'ai un exécutable appelé `foobar`, je le vois avec `ls`, mais quand je tape `foobar` sur la ligne de commande, il me dit que le fichier n'existe pas. Pourquoi ne le trouve-t-il pas?

R. A l'inverse de MS-DOS, Unix ne regarde pas dans le répertoire courant quand il cherche le programme que vous voulez exécuter, à moins que vous ne le lui disiez. Soit tapez `./foobar`, ce qui veut dire "exécuter le fichier appelé `foobar` du répertoire courant", ou modifiez votre variable d'environnement `PATH` pour qu'elle ressemble à:

```
bin:/usr/bin:/usr/local/bin:.
```

Le dernier point signifie "chercher dans le répertoire courant si le fichier n'est pas dans les autres répertoires".

Q. J'ai appelé mon exécutable `test`, mais il ne se passe rien quand je le lance. Pourquoi?

R. Il y a un programme appelé `test` dans `/usr/bin` sur la plupart des systèmes Unix et c'est celui-là que trouve l'interpréteur de commandes avant de regarder dans le répertoire courant. Soit tapez:

```
% ./test
```

ou choisissez un meilleur nom pour votre programme!

Q. J'ai compilé et tout a commencé à fonctionner correctement, puis il y a eu une erreur et il m'a dit quelque chose à propos de "`core dumped`". Qu'est-ce que cela veut dire?

A. L'expression *core dump* date des tous premiers jours d'Unix, quand les machines utilisaient la mémoire centrale - "*core memory*" pour stocker les informations. Essentiellement, si le programme "plantait" dans certaines conditions, le système enregistrait sur disque le contenu de la mémoire centrale dans un fichier appelé `core`, que le programmeur pouvait ensuite disséquer pour trouver où les choses avaient mal tournées.

Q. Fascinant, mais que suis-je censé faire maintenant?

A. Servez-vous de `gdb` pour analyser l'image mémoire (Reportez-vous à la section Déboguer).

R. Quand mon programme a généré une image mémoire, il a dit quelque chose à propose de "`segmentation fault`" - "erreur de segmentation". Qu'est-ce que c'est?

Q. Cela signifie essentiellement que votre programme a essayé d'effectuer une quelconque opération illégale sur la mémoire; Unix est conçu pour protéger le système d'exploitation des programmes mal éduqués.

Les raisons les plus courantes en sont:

- Essayer d'écrire en mémoire adressée par un pointeur NULL, e.g.:

  ```
  char *foo = NULL;
  strcpy(foo, "bang!");
  ```

- Utiliser un pointeur qui n'a pas été initialisé, e.g.:

  ```
  char *foo;
  strcpy(foo, "bang!");
  ```

  Le pointeur aura une valeur aléatoire qui, avec de la chance, adressera une zone mémoire non accessible à votre programme, de sorte que le noyau tuera ce dernier avant qu'il ne provoque de dégât. Si vous manquez de chance, il pointera quelque part à l'intérieur de votre programme et endommagera l'une de vos structures de données, provoquant un dysfonctionnement mystérieux de votre programme.

- Tenter d'accéder au-delà du dernier élément d'un tableau, e.g.:

```
int bar[20];
bar[27] = 6;
```

- Essayer d'enregistrer quelque chose dans une zone de mémoire accessible en lecture seule, e.g.:

  ```
  char *foo = "Mon texte";
  strcpy(foo, "bang!");
  ```

  Les compilateurs Unix stockent souvent les chaînes de caractères constantes comme `"Mon texte"` en mémoire accessible en lecture seule.

- Utiliser incorrectement les fonctions `malloc()` et `free()`, e.g.:

  ```
  char bar[80];
  free(bar);
  ```

  ou:

  ```
  char *foo = malloc(27);
  free(foo);
  free(foo);
  ```

Commettre l'une de ces fautes ne provoquera pas toujours une erreur, mais ce sont malgré tout des choses à ne pas faire. Certains systèmes et compilateurs sont plus tolérants que d'autres, ce qui fait que des programmes qui s'exécutent correctement sur un système peuvent ne plus fonctionner sur un autre.

Q. Parfois, le programme provoque la génération d'une image mémoire avec le message "`bus error`". Mon manuel Unix dit qu'il s'agit d'un erreur matériel, mais l'ordinateur fonctionne apparemment correctement. Est-ce vrai?

R. Fort heureusement, non (à moins bien sûr que vous n'ayez aussi un problème matériel). C'est habituellement une autre façon de dire que vous avez accédé incorrectement à la mémoire.

Q. Il me semble que cette histoire de *core dump* peut être très utile, si je peux la provoquer quand je veux. Est-ce possible, ou dois-je attendre qu'il se produise une erreur?

R. Oui, allez simplement sur une autre console ou fenêtre **xterm** et tapez:

```
% ps
```

pour connaître l'IDentifiant de processus de votre programme, puis:

```
% kill -ABRT pid
```

où *`pid`* est l'ID de processus que vous avez recherché.

C'est par exemple utile si votre programme est parti dans une boucle infinie. Au cas où votre programme piégerait les interruptions SIGABRT, il y a plusieurs autres signaux qui auront le même effet.

4. Make

4.1. Qu'est-ce que `make`?

Lorsque vous travaillez sur un programme simple avec seulement un ou deux fichiers de source, taper:

```
% cc fichier1.c fichier2.c
```

n'est pas trop gênant, mais cela devient rapidement très fastidieux lorsqu'il y a plusieurs fichiers - et cela peut aussi mettre du temps à compiler.

Un façon d'éviter ces problèmes est d'utiliser des fichiers *objets* et de ne recompiler que les fichiers de source dont le contenu a changé. Nous pourrions alors avoir quelque chose du style:

```
% cc fichier1.o fichier2.o ... file37.c ...
```

si nous avons modifé `fichier37.c`, et celui-là uniquement, depuis notre compilation précédente. Cela peut sérieusement accélérer la compilation, mais ne résout pas le problème de saisie à répétition de la commande.

Nous pourrions aussi écrire une procédure pour résoudre ce dernier problème, mais ne ne pourrions alors que lui faire tout recompiler, ce qui serait très peu efficace sur un gros projet.

Que ce passe-t-il si nous avons des centaines de fichiers de sources? Si nous travaillons en équipe et que d'autres oublient de nous prévenir des modifications qu'ils ont apportées à un des fichiers que nous utilisons?

Peut-être pourrions-nous rassembler les deux solutions et écrire quelque chose qui ressemble à une procédure et comporte une sorte de règle magique qui dise quand tel fichier de source doit être compilé. Nous n'aurions plus besoin que d'un programme qui comprennent ces règles, parce que c'est un peu trop compliqué pour une procédure.

Ce programme s'appelle `make`. Il lit un fichier, qu'on appelle un *makefile*, qui lui dit quelles sont les dépendances entre les différents fichiers, et en déduit lesquels ont besoin ou non d'être recompilés. Par exemple, une règle peut signifier quelque chose comme "si `fromboz.o` est plus ancien que `fromboz.c`, cela veut dire que `fromboz.c` doit avoir été modifié, il faut donc le recompiler". Le fichier "*makefile*" inclut aussi des règles qui lui disent *comment* recompiler, ce qui en fait un outil encore plus puissant.

Ces fichiers "makefiles" sont habituellement rangés dans le même répertoire que les sources auxquels ils s'appliquent, et peuvent être appelés `makefile`, `Makefile` ou `MAKEFILE`. La plupart des programmeurs utilisent le nom `Makefile`, ce qui fait qu'ils se trouvent alors vers le début de la liste des fichiers et sont ainsi facilement repérables [6].

4.2. Exemple d'utilisation de `make`

Voici un fichier `Makefile` élémentaire :

```
foo: foo.c
	cc -o foo foo.c
```

Il contient deux lignes, une pour la dépendance et une pour la génération.

La ligne décrivant la dépendance contient le nom du programme (qu'on appelle la *cible*), suivi de "deux points", puis d'un blanc et du nom du fichier source. Quand `make` lit cette ligne, il regarde si `foo` existe; s'il existe, il compare la

date de dernière modification de `foo` à celle de dernière modification de `foo.c`. Si `foo` n'existe pas, ou s'il est antérieur à `foo.c`, il regarde alors la ligne de génération pour savoir ce qu'il faut faire. En d'autres termes, c'est la règle à appliquer pour savoir si `foo.c` doit être recompilé.

La ligne de génération commence par une tabulation (appuyez sur la touche **Tab**) suivie de la commande que vous taperiez pour compiler `foo` si vous le faisiez sur la ligne de commande. Si `foo` n'est pas à jour ou s'il n'existe pas, `make` exécute alors cette commande pour le créer. En d'autres termes, c'est la règle qui dit à `make` comment recompiler `foo.c`.

Ainsi, quand vous tapez **`make`**, il fera en sorte que `foo` soit en phase avec les dernières modifications que vous avez apportées à `foo.c`. Ce principe s'étend aux `Makefile`s avec des centaines de cibles - de fait, sur FreeBSD, il est possible de compiler tout le système d'exploitation en tapant simplement **`make world`** dans le répertoire adéquat!

Une autre particularité de `Makefile`s est que les cibles ne sont pas nécessairement des programmes. Nous pourrions par exemple avoir le `Makefile` suivant:

```
foo: foo.c
      cc -o foo foo.c

install:
      cp foo /home/me
```

Nous pouvons dire à `make` quelle cible nous voulons atteindre en tapant:

```
% make cible
```

`make` examinera alors cette cible et ignorera toutes les autres. Par exemple, si, avec le `Makefile` précédent, nous tapons **`make foo`**, `make` ignorera la cible install.

Si nous tapons simplement **`make`** tout court, il examinera toujours la première cible et s'arrêtera ensuite sans s'occuper des autres. Si nous avions tapé **`make`** dans ce cas, il serait simplement allé à la cible foo, aurait recompilé `foo` si nécessaire, et se serait arrêté sans passer à la cible install.

Remarquez que la cible install ne dépend en fait de rien du tout! Cela signifie que la commande sur la ligne suivante est toujours exécutée si nous essayons de reconstruire cette cible en tapant **`make install`**. Dans ce cas, il copiera `foo` dans le répertoire de l'utilisateur. C'est souvent utilisé par les `Makefile`s de logiciels, de sorte que l'application soit installée dans le bon répertoire, une fois correctement compilée.

C'est un point un peu délicat à expliquer. Si vous ne comprenez pas exactement comment `make` fonctionne, la meilleure chose à faire est d'écrire un programme simple comme le classique "Bonjour, le monde!", un fichier `Makefile` et de faire des essais. Compilez ensuite en utilisant plus d'un fichier source, ou en ayant un fichier source qui inclut un fichier d'en-tête. La commande `touch` vous sera très utile - elle modifie la date d'un fichier sans que vous ayez à l'éditer.

4.3. Makefiles FreeBSD

L'écriture de `Makefile`s peut être assez compliquée. Heureusement, les systèmes basés sur BSD, comme FreeBSD, en fournissent de très puissants, intégrés au système. Le catalogue des logiciels portés de FreeBSD en est un

excellent exemple. Voici l'essentiel d'un de leurs `Makefile`s typiques:

```
MASTER_SITES=   ftp://freefall.cdrom.com/pub/FreeBSD/LOCAL_PORTS/
DISTFILES=      scheme-microcode+dist-7.3-freebsd.tgz

.include <bsd.port.mk>
```

Si nous allons maintenant dans le répertoire associé à ce logiciel et tapons **`make`**, voici ce qui se passe:

1. Il regarde si le code source de ce logiciel est déjà présent sur le système.
2. S'il n'y est pas, une connexion FTP à l'URL indiquée par MASTER_SITES est établie pour télécharger le source.
3. La somme de contrôle est calculée sur le source et comparée à celle calculée sur une version connue et validée. Cela pour s'assurer que le source n'a pas été corrompu pendant le transfert.
4. Les modifications nécessaires pour que le code fonctionne sous FreeBSD sont appliquées - c'est ce que l'on appelle *patcher*.
5. Les opérations particulières de configuration du source sont effectuées. (De nombreuses distributions de programmes Unix essayent de déterminer sur quel système elles sont compilées et de quelles fonctionnalités Unix optionnelles il dispose - c'est à ce stade du scénario d'installation de logiciels sous FreeBSD que leur sont fournies ces informations).
6. Le code source du programme est compilé. De fait, on passe dans le répertoire où le code a été décompacté et `make` y est exécuté - le `Makefile` du programme lui-même contient les informations nécessaires à sa compilation.
7. Nous disposons maintenant d'une version compilée du programme. Si nous le voulons, nous pouvons maintenant la tester; si nous avons confiance dans le programme, nous pouvons taper **`make install`**. Cela recopiera le programme et tous les fichiers d'environnement dont il a besoin à l'endroit adéquat; une entrée sera aussi créée dans une base de données des logiciels, de façon à ce qu'il puisse être désinstallé par la suite, si nous changeons d'avis.

Je pense que vous serez maintenant d'accord pour trouver que c'est assez impressionnant pour une simple procédure de quatre lignes!

Le secret se trouve à la dernière ligne, qui dit à `make` d'aller voir ce qu'il y a dans le `Makefile` appelé `bsd.port.mk`. Il est facile de rater cette ligne, mais c'est pourtant de là que vient toute la mécanique subtile. Quelqu'un a écrit un `Makefile` qui dit à `make` de faire tout ce qui a été décrit ci-dessus (plus deux ou trois autres choses dont je n'ai pas parlé, dont le traitement des erreurs qui pourraient se produire) et tout le monde peut l'utiliser en mettant simplement cette unique ligne dans son propre `Makefile`!

Si vous voulez jeter un oeil à ces `Makefile`s systèmes, ils sont dans le répertoire `/usr/share/mk`, mais il vaut mieux attendre d'avoir un peu d'expérience des `Makefile`s, parce qu'ils sont très compliqués (et si vous les regardez, ayez sous la main une bonne dose de café serré!).

4.4. Utilisation plus poussée de `make`

`make` est un outil très puissant, et peut faire beaucoup plus que l'exemple élémentaire que nous avons donné. Il y a malheureusement plusieurs versions de `make`, et elles sont très différentes. La meilleure façon de savoir ce qu'elles peuvent faire est certainement de lire la documentation - espérons que cette introduction vous aura fourni les bases pour le faire.

La version de `make` fournie avec FreeBSD est **Berkeley make**; elle s'accompagne d'un guide dans `/usr/share/doc/psd/12.make`. Pour le visualiser, tapez:

```
% zmore paper.ascii.gz
```

dans ce répertoire.

Il y a des nombreux logiciels du catalogue des logiciels portés qui utilisent **GNU make**, qui est très bien documenté dans les pages "info". Si vous avez installé un de ces logiciels, **GNU make** sera automatiquement installé sont le nom `gmake`. Il est aussi disponible sous forme de logiciel porté ou précompilé autonome.

Pour visualiser les pages "info" de **GNU make**, il vous faut éditer le fichier `dir` du répertoire `/usr/local/info` et y ajouter une ligne pour ce programme. C'est une ligne du genre:

```
* Make: (make).                 L'utilitaire GNU Make.
```

Une fois que c'est fait, vous pouvez taper **`info`** puis sélectionner make dans le menu (ou sous **Emacs**, taper **`C-h i`**).

5. Déboguer

5.1. Le débogueur

Le débogueur qui est fourni avec FreeBSD s'appelle `gdb` (**GNU débogueur**). Vous le lancez en tapant:

```
% gdb nom_du_programme
```

bien que la plupart des gens préfèrent l'exécuter sous **Emacs**. Ce qui se fait avec:

```
M-x gdb RET progname RET
```

Se servir d'un débogueur vous permet d'exécuter le programme sous contrôle. Vous pouvez typiquement l'exécuter pas à pas, inspecter les valeurs des variables, les modifier, dire au débogueur d'exécuter le programme jusqu'à un certain endroit et de s'y arrêter, et ainsi de suite. Vous pouvez même le rattacher à un programme qui est déjà en cours d'exécution, ou charger une image mémoire - "*core*". Il est même possible de déboguer le noyau, bien que cela soit un peu plus compliqué que dans le cas des programmes utilisateurs, dont nous parlerons dans cette section.

`gdb` dispose d'une assez bonne aide en ligne, ainsi que d'un jeu de pages "info", cette section se concentrera donc sur quelques commandes de base.

Enfin, si le mode de fonctionnement en ligne de commande vous rebute, il existe une interface graphique appelée xxgdb (http://www.freebsd.org/ports/devel.html) au catalogue des logiciels portés.

Cette section est destinée à servir d'introduction à l'utilisation du débogueur et ne couvre pas les questions spécialisées comme le débogage du noyau.

5.2. Exécuter un programme sous le débogueur

Il faudra que vous ayez compilé le programme avec l'option `-g` pour tirer le meilleur parti de `gdb`. Cela fonctionnera sans cela, mais vous ne verrez que le nom de la fonction dans laquelle vous êtes, au lieu du code source. Si vous avez un message du genre:

```
... (no debugging symbols found) ...
```

au démarrage de `gdb`, vous saurez que le programme n'a pas été compilé avec l'option `-g`.

A l'invite de `gdb`, tapez **`break main`**. Cela dira au débogueur d'exécuter le code préliminaire de mise en oeuvre dans le programme et de s'arrêter au début de votre programme. Tapez maintenant **`run`** pour lancer le programme - il commencera au début du code préliminaire et sera interrompu par le débogueur sur l'appel de `main()`. (Si vous vous étiez jamais demandé d'où la fonction `main()` était appelée, vous le savez maintenant!).

Vous pouvez maintenant exécuter le programme une ligne à la fois, en appuyant sur `n`. Si vous arrivez sur un appel de fonction, vous pouvez passer dans la fonction en appuyant sur `s`. Une fois dans la fonction, vous pouvez terminer son exécution et en sortir en tapant `f`. Vous pouvez aussi utiliser `up` et `down` pour jeter un coup d'oeil au code appelant.

Voici un exemple simple de la manière de diagnostiquer une erreur dans un programme avec `gdb`. Voici notre programme (intentionnellement faux):

```
#include <stdio.h>

int bazz(int un_entier);

main() {
      int i;

      printf("C'est mon programme\n");
      bazz(i);
      return 0;
}

int bazz(int un_entier) {
      printf("Vous m'avez donné %d\n", un_entier);
      return un_entier;
}
```

Ce programme affecte à i la valeur `5` et la passe à la fonction `bazz()` qui affiche la valeur que nous lui donnons en paramètre.

Quand nous compilons et exécutons le programme, nous obtenons:

```
% cc -g -o temp temp.c
% ./temp
C'est mon programme
```

```
Vous m'avez donné 4231
```

Ce n'est pas ce à quoi nous nous attendions! C'est le moment d'aller voir ce qui ce passe!

```
% gdb temp
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.13 (i386-unknown-freebsd), Copyright 1994 Free Software Foundation, Inc.
(gdb) break main      Exécuter le code d'initialisation
Breakpoint 1 at 0x160f: file temp.c, line 9.  gdb met un pont d'arrêt à l'appel de main()
(gdb) run       Exécuter jusqu'à main()
Starting program: /home/james/tmp/temp   Le programme démarre

Breakpoint 1, main () at temp.c:9   gdb s'arrête à main()
(gdb) n       Aller à la ligne suivante
C'est mon programme      Le programme imprime
(gdb) s       Aller dans bazz()
bazz (un_entier=4231) at temp.c:17   gdb affiche la position dans la pile d'appel
(gdb)
```

Une minute! Comment un_entier peut-il valoir `4231`? Ne l'avons-nous pas initialisé à `5` dans `main()`? Revenons à `main()` et jetons un oeil.

```
(gdb) up       Remonter d'un cran dans la pile d'appel
#1  0x1625 in main () at temp.c:11   gdb affiche la position dans la pile d'appel
(gdb) p i      Afficher la valeur de i
$1 = 4231      gdb affiche 4231
```

Et oui! Si nous regardons le code, nous avons oublié d'initialiser i. Nous voulions mettre:

```
...
main() {
      int i;

      i = 5;
      printf("C'est mon programme\n");
...
```

mais nous avons oublié la ligne `i=5;`. Comme nous n'avons pas initialisé i, il prend la valeur qui se trouve à cet endroit de la mémoire quand le programme s'exécute, dans notre cas, il s'est trouvé que c'était `4231`.

> **Note :** `gdb` affiche l'endroit où nous nous trouvons dans la pile d'appel, chaque fois que nous entrons ou sortons d'une fonction, même si nous utilisons `up` et `down` pour nous déplacer dans la pile. Cela nous donne le nom de la fonction et les valeurs de ses paramètres, ce qui nous aide à repérer où nous sommes et ce qu'il se passe. (La pile d'appel est une zone de mémoire où le programme enregistre les informations sur les paramètres passés aux fonctions et où aller quand il ressort d'un fonction appelée.)

5.3. Examiner un fichier "*core*"

Un fichier "*core*" est essentiellement un fichier qui contient l'état complet du programme au moment où il s'est "planté". Au "bon vieux temps", les programmeurs devaient imprimer le contenu en hexadécimal des fichiers "*core*" et transpirer sur des manuels de code machine, mais la vie est aujourd'hui un peu plus facile. Au passage, sous FreeBSD et les autres systèmes 4.4BSD, un fichier "*core*" s'appelle `nom_du_programme.core`, et non `core` tout court, de façon à ce que l'on sache à quel programme il correspond.

Pour examiner un fichier "*core*", lancez `gdb` comme d'habitude. Au lieu de taper `break` ou `run`, tapez:

```
(gdb) core nom_du_programme.core
```

Si vous n'êtes pas dans le même répertoire que le fichier "*core*", vous devrez d'abord faire **`dir /ou/se/trouve/le/fichier/core`**.

Vous devriez voir quelque chose comme:

```
% gdb a.out
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.13 (i386-unknown-freebsd), Copyright 1994 Free Software Foundation, Inc.
(gdb) core a.out.core
Core was generated by `a.out'.
Program terminated with signal 11, Segmentation fault.
Cannot access memory at address 0x7020796d.
#0  0x164a in bazz (un_entier=0x5) at temp.c:17
(gdb)
```

Dans ce cas, le programme s'appelait `a.out`, le fichier "*core*" s'appelle donc `a.out.core`. Nous constatons que le programme s'est terminé en erreur à cause d'une tentative d'accès à une zone de mémoire qui n'était pas accessible dans une fonction appelé `bazz`.

Il est parfois utile de pouvoir savoir comment une fonction a été appelée, parce que le problème peut s'être produit bien au-dessus dans la pile d'appel dans un programme complexe. La commande `bt` dit à `gdb` d'imprimer en remontant la pile d'appel:

```
(gdb) bt
#0  0x164a in bazz (un_entier=0x5) at temp.c:17
#1  0xefbfd888 in end ()
#2  0x162c in main () at temp.c:11
(gdb)
```

La fonction `end()` est appelée quand un programme échoue; dans le cas présent, la fonction `bazz()` a été appelée par `main()`.

5.4. Prendre le contrôle d'un programme en cours d'exécution

Une des possibilités les plus intéressantes de `gdb` est qu'il peut se rattacher à un programme en cours d'exécution. Il faut bien sûr que vous ayez les autorisations suffisantes pour le faire. Le cas d'un programme qui

"fourche" - *fork* - est un problème classique, lorsque vous voulez suivre le déroulement du processus fils, alors que le débogueur ne vous permet que de tracer le processus père.

Vous lancez alors un autre `gdb`, utilisez `ps` pour connaître l'IDentifiant de processus du fils, puis faites:

```
(gdb) attach pid
```

sous `gdb`, et déboguez alors comme d'habitude.

"Tout cela est bien beau", vous dites vous peut-être, "mais le temps que j'ai fait tout ça, le processus fils aura déjà fait un bon bout de chemin". Rien à craindre, aimable lecteur, voici ce qu'il faut faire (emprunté aux pages "info" de `gdb`):

```
...
if ((pid = fork()) < 0)         /* _Toujours_ effectuer se contrôle */
      error();
else if (pid == 0) {            /* C'est le processus fils */
      int PauseMode = 1;

      while (PauseMode)
            sleep(10);          /* Attendre que quelqu'un se rattache à nous */
      ...
} else {                        /* parent */
      ...
```

Il n'y a plus qu'à se rattacher au fils, positionner PauseMode à `0`, et attendre que l'appel de `sleep()` nous rende la main!

6. Utiliser Emacs comme environnement de développement

6.1. Emacs

Les systèmes Unix ne s'accompagnent malheureusement pas du type d'environnement de développement intégré du genre "tout ce que vous avez toujours voulu et encore beaucoup plus en un seul monstrueux paquetage" dont disposent d'autres systèmes[7]. Il est cependant possible de mettre au point votre propre environnement. Il ne sera peut-être pas aussi esthétique et il ne sera peut-être pas aussi intégré, mais vous pourrez le configurer comme vous le voulez. Et il est gratuit. En plus, vous en avez les sources.

Emacs est la clé de tout. Il y a bien des gens qui le décrient, mais nombreux sont ceux qui l'aiment. Si vous êtes des premiers, j'ai peur que cette section n'ait que peu d'intérêt pour vous. Il vous faudra aussi pas mal de mémoire pour l'utiliser. Je conseille 8Mo en mode texte et 16Mo sous X comme strict minimum pour avoir des temps de réponse raisonnables.

Emacs est essentiellement un éditeur extrêmement configurable - il a de fait été configuré au point de ressembler plus à un système d'exploitation qu'à un éditeur! De nombreux développeurs et administrateurs système passent le plus clair de leur temps à travailler sous Emacs, le quittant seulement pour se déconnecter.

Il est impossible de même résumer ici tout ce qu'Emacs est capable de faire, mais voici quelques fonctionnalités qui intéressent les développeurs:

- Editeur très puissant qui permet et de rechercher et remplacer des chaînes de caractères et d'utiliser pour le faire des expressions régulières (motifs), d'aller au début ou à la fin de blocs syntaxiques, etc, etc.
- Menus déroulants et aide en ligne.
- Mise en valeur et indentation en fonction de la syntaxe du langage utilisé.
- Complètement configurable.
- Vous pouvez compiler et déboguer des programmes depuis Emacs.
- En cas d'erreur à la compilation, vous pouvez aller directement à la ligne qui en est la cause.
- Interface ergonomique au programme `info` qui sert à lire la documentation hypertexte GNU, dont celle d'Emacs lui-même.
- Interface conviviale pour `gdb`, qui vous permet de visualiser le code source en même temps que vous exécutez pas à pas le programme.
- Vous pouvez lire les "news" Usenet et votre courrier électronique pendant que votre programme compile.

Et sans aucun doute bien d'autres choses qui m'ont échappées.

Emacs peut être installé sous FreeBSD sous forme de logiciel porté (http://www.freebsd.org/ports/editors.html).

Une fois qu'il est installé, lancez-le et tapez **`C-h t`** - ce qui signifie maintenir enfoncée la touche **Ctrl**, taper **h**, relâcher la touche **Ctrl**, et appuyer ensuite sur **t** - pour lire le guide d'Emacs (Vous pouvez aussi utiliser la souris pour sélectionner Emacs Tutorial - "*Guide Emacs*" - depuis le menu Help - "*Aide*").

Bien qu'Emacs ait des menus, il vaut la peine d'apprendre à utiliser les raccourcis claviers, parce qu'il est bien plus rapide quand vous éditez quelque chose d'appuyer sur deux ou trois touches que de courir après la souris et cliquer ensuite au bon endroit. Si, par ailleurs, vous discutez avec des utilisateurs expérimentés d'Emacs, vous vous apercevrez qu'ils utilisent assez couramment des expressions comme "`M-x replace-s RET foo RET bar RET`", il peut donc servir de comprendre ce qu'ils veulent dire. Et de toute façon, Emacs a bien plus de fonctions utiles qu'il ne peut en tenir sur une barre de menus.

Il est heureusement assez facile de découvrir les raccourcis claviers, ils sont affichés dans les menus. Je vous conseille d'utiliser les menus pour, par exemple, ouvrir un fichier jusqu'à ce que vous compreniez comment cela marche et ayez suffisamment confiance en vous, puis d'essayer C-x C-f. Une fois que cela vous convient, passez à une autre des commandes des menus.

Si vous ne vous rappelez pas ce que fait une combinaison donnée de touches, choisissez Describe Key - "*Description d'une touche*" - dans le menu Help - "*Aide*" - et tapez cette combinaison - Emacs vous dira ce qu'elle fait. Vous pouvez aussi utiliser le choix Command Apropos - "*A propos d'une commande*" - pour connaître toutes les commandes comportant un mot donné et les touches qui leur correspondent.

Au fait, l'expression plus haut signifie: enfoncer la touche Méta, appuyer sur x, relâcher la touche Méta, taper **`replace-s`** (abréviation de `replace-string` - "*remplacer une chaîne de caractères*" - une autre caractéristique d'Emacs est de vous permettre d'abréger les commandes), appuyer sur Entrée, taper **`foo`** (la chaîne que vous voulez remplacer), appuyer sur Entrée, taper **`bar`** (la chaîne avec laquelle vous voulez remplacer `foo`) et appuyer encore sur Entrée. Emacs effectuera alors l'opération de recherche et remplacement que vous venez de demander.

Si vous vous demandez ce qu'est la touche Méta, c'est une touche spéciale qu'ont beaucoup de stations Unix. Malheureusement, les PCs n'en ont pas, c'est habituellement la touche Alt qui est utilisée (ou si vous n'avez pas de chance, la touche Echap).

Oh, et pour sortir d'Emacs, tapez `C-x C-c` (Ce qui signifie: enfoncer la touche Ctrl, appuyer sur c, appuyer sur x et relâcher la touche Ctrl). S'il y a des fichiers ouverts que vous n'avez pas sauvegardés, Emacs vous demandera si vous voulez les sauvegarder. (Oubliez que la documentation dit que la méthode habituelle pour quitter Emacs est d'utiliser `C-z` - cela laisse Emacs actif en tâche de fond et n'est réellement utile que si vous êtes sur un système qui ne gère pas de terminaux virtuels).

6.2. Configurer Emacs

Emacs fait des choses admirables; certaines fonctionnalités sont incorporées, d'autres doivent être configurées.

Au lieu d'utiliser un langage de macros-instructions propriétaires, Emacs se sert d'une version de Lisp spécialement adaptée aux éditeurs, connue sous le nom de Emacs Lisp. Ce peut être très utile si vous voulez aller plus loin et apprendre ensuite par exemple Common Lisp, parce qu'il est considérablement plus léger que Common Lisp (quoique qu'encore assez imposant!).

La meilleure façon d'apprendre Emacs Lisp est de télécharger le Guide Emacs Lisp (ftp://prep.ai.mit.edu/pub/gnu/elisp-manual-19-2.4.tar.gz).

Il n'y a cependant pas besoin de connaître quoique ce soit à Lisp pour commencer à configurer Emacs, parce que j'ai inclu un fichier `.emacs` d'exemple, qui devrait suffire au début. Copiez-le simplement dans votre répertoire utilisateur et relancez Emacs, s'il s'exécute déjà; il lira les commandes du fichier et (je l'espère) vous fournira une configuration de base utile.

6.3. Un exemple de fichier `.emacs`

Il contient malheureusement beaucoup trop de choses pour tout expliquer en détails; il y a cependant un ou deux points intéressants à mentionner.

- Tout ce qui commence par un `;` est en commentaire et est ignoré par Emacs.
- La première ligne, `-*- Emacs-Lisp -*-` permet d'éditer le fichier `.emacs` lui-même sous Emacs et de profiter de toutes les fonctionnalitées liées à l'édition de code Emacs Lisp. Emacs tente habituellement de deviner le type de fichier en fonction de son nom, mais risque de ne pas y arriver pour le fichier `.emacs`.
- La touche Tab est utilisée pour l'indentation dans certains modes, de sorte que si vous appuyez sur cette touche cela indente la ligne de code courante. Si vous voulez mettre un caractère tabulation dans votre texte, enfoncer la touche Ctrl en même temps que vous appuyez sur Tab.
- Ce fichier permet le mise en valeur syntaxique de code C, C++, Perl, Lisp et Scheme, en déterminant le langage d'après le nom du fichier édité.
- Emacs a déjà une fonction prédéfinie appelée `next-error` - "*erreur suivante*". Dans la fenêtre de résultats d'une compilation, cela vous permet d'aller d'une erreur à la suivante avec `M-n`; nous définissons la fonction complémentaire `previous-error` - "*erreur précédente*", qui vous permet de retourner à l'erreur précédente avec `M-p`. Le plus sympathique est que `C-c C-c` ouvrira le fichier source où l'erreur s'est produite et ira à la ligne concernée.

- Nous activons la possibilité qu'a Emacs d'agir comme serveur, de façon à ce que si vous travaillez hors d'Emacs et voulez éditer un fichier, il vous suffise de taper:

  `% emacsclient nom_du_fichier`

  pour pouvoir ensuite le modifier avec Emacs![8].

Exemple 1. Un exemple de fichier `.emacs`

```
;; -*-Emacs-Lisp-*-

;; Ce fichier est conçu pour être relu; la variable
;; first-time est utilisée pour éviter les problèmes
;; que cela pourra poser.
(defvar first-time t
  "Indicateur signifiant que le fichier .emacs est lu pour la première fois")

;; Méta
(global-set-key "\M- " 'set-mark-command)
(global-set-key "\M-\C-h" 'backward-kill-word)
(global-set-key "\M-\C-r" 'query-replace)
(global-set-key "\M-r" 'replace-string)
(global-set-key "\M-g" 'goto-line)
(global-set-key "\M-h" 'help-command)

;; Touches fonction
(global-set-key [f1] 'manual-entry)
(global-set-key [f2] 'info)
(global-set-key [f3] 'repeat-complex-command)
(global-set-key [f4] 'advertised-undo)
(global-set-key [f5] 'eval-current-buffer)
(global-set-key [f6] 'buffer-menu)
(global-set-key [f7] 'other-window)
(global-set-key [f8] 'find-file)
(global-set-key [f9] 'save-buffer)
(global-set-key [f10] 'next-error)
(global-set-key [f11] 'compile)
(global-set-key [f12] 'grep)
(global-set-key [C-f1] 'compile)
(global-set-key [C-f2] 'grep)
(global-set-key [C-f3] 'next-error)
(global-set-key [C-f4] 'previous-error)
(global-set-key [C-f5] 'display-faces)
(global-set-key [C-f8] 'dired)
(global-set-key [C-f10] 'kill-compilation)

;; Touches curseur
(global-set-key [up] "\C-p")
(global-set-key [down] "\C-n")
(global-set-key [left] "\C-b")
(global-set-key [right] "\C-f")
(global-set-key [home] "\C-a")
(global-set-key [end] "\C-e")
```

```
(global-set-key [prior] "\M-v")
(global-set-key [next] "\C-v")
(global-set-key [C-up] "\M-\C-b")
(global-set-key [C-down] "\M-\C-f")
(global-set-key [C-left] "\M-b")
(global-set-key [C-right] "\M-f")
(global-set-key [C-home] "\M-<")
(global-set-key [C-end] "\M->")
(global-set-key [C-prior] "\M-<")
(global-set-key [C-next] "\M->")

;; Souris
(global-set-key [mouse-3] 'imenu)

;; Divers
(global-set-key [C-tab] "\C-q\t") ; Ctrl tab = caractère tabulation.
(setq backup-by-copying-when-mismatch t)

;; 'y' ou <CR> équivaut à yes, 'n' à no.
(fset 'yes-or-no-p 'y-or-n-p)
    (define-key query-replace-map [return] 'act)
    (define-key query-replace-map [?\C-m] 'act)

;; Paquetages à charger
(require 'desktop)
(require 'tar-mode)

;; Mode diff évolué
(autoload 'ediff-buffers "ediff" "Interface Emacs intelligente pour diff" t)
(autoload 'ediff-files "ediff" "Interface Emacs intelligente pour diff" t)
(autoload 'ediff-files-remote "ediff"
  "Interface Emacs intelligente pour diff")
(if first-time
    (setq auto-mode-alist
          (append '(("\\.cpp$" . c++-mode)
                    ("\\.hpp$" . c++-mode)
                    ("\\.lsp$" . lisp-mode)
                    ("\\.scm$" . scheme-mode)
                    ("\\.pl$" . perl-mode)
                   ) auto-mode-alist)))

;; Mise en valeur syntaxique automatique
(defvar font-lock-auto-mode-list
  (list 'c-mode 'c++-mode 'c++-c-mode 'emacs-lisp-mode 'lisp-mode 'perl-mode 'scheme-mode)
  "Listes des modes à démarrer toujours avec mise en valeur")

(defvar font-lock-mode-keyword-alist
  '((c++-c-mode . c-font-lock-keywords)
    (perl-mode . perl-font-lock-keywords))
  "Associations entre modes et mots-clés")

(defun font-lock-auto-mode-select ()
  "Sélectionne automatiquement type de mise en valeur si le major mode courant est dans font-lock-au
```

```
  (if (memq major-mode font-lock-auto-mode-list)
      (progn
      (font-lock-mode t))
    )
  )

(global-set-key [M-f1] 'font-lock-fontify-buffer)

;; Nouveau dabbrev
;(require 'new-dabbrev)
(setq dabbrev-always-check-other-buffers t)
(setq dabbrev-abbrev-char-regexp "\\sw\\|\\s_")
(add-hook 'emacs-lisp-mode-hook
          '(lambda ()
            (set (make-local-variable 'dabbrev-case-fold-search) nil)
            (set (make-local-variable 'dabbrev-case-replace) nil)))
(add-hook 'c-mode-hook
          '(lambda ()
            (set (make-local-variable 'dabbrev-case-fold-search) nil)
            (set (make-local-variable 'dabbrev-case-replace) nil)))
(add-hook 'text-mode-hook
          '(lambda ()
            (set (make-local-variable 'dabbrev-case-fold-search) t)
            (set (make-local-variable 'dabbrev-case-replace) t)))

;; Mode C++ et C...
(defun my-c++-mode-hook ()
  (setq tab-width 4)
  (define-key c++-mode-map "\C-m" 'reindent-then-newline-and-indent)
  (define-key c++-mode-map "\C-ce" 'c-comment-edit)
  (setq c++-auto-hungry-initial-state 'none)
  (setq c++-delete-function 'backward-delete-char)
  (setq c++-tab-always-indent t)
  (setq c-indent-level 4)
  (setq c-continued-statement-offset 4)
  (setq c++-empty-arglist-indent 4))

(defun my-c-mode-hook ()
  (setq tab-width 4)
  (define-key c-mode-map "\C-m" 'reindent-then-newline-and-indent)
  (define-key c-mode-map "\C-ce" 'c-comment-edit)
  (setq c-auto-hungry-initial-state 'none)
  (setq c-delete-function 'backward-delete-char)
  (setq c-tab-always-indent t)
;; indentation style BSD
  (setq c-indent-level 4)
  (setq c-continued-statement-offset 4)
  (setq c-brace-offset -4)
  (setq c-argdecl-indent 0)
  (setq c-label-offset -4))

;; Mode Perl...
(defun my-perl-mode-hook ()
```

```
  (setq tab-width 4)
  (define-key c++-mode-map "\C-m" 'reindent-then-newline-and-indent)
  (setq perl-indent-level 4)
  (setq perl-continued-statement-offset 4))

;; Mode Scheme...
(defun my-scheme-mode-hook ()
  (define-key scheme-mode-map "\C-m" 'reindent-then-newline-and-indent))

;; Mode Emacs-Lisp...
(defun my-lisp-mode-hook ()
  (define-key lisp-mode-map "\C-m" 'reindent-then-newline-and-indent)
  (define-key lisp-mode-map "\C-i" 'lisp-indent-line)
  (define-key lisp-mode-map "\C-j" 'eval-print-last-sexp))

;; Ajouts des précédents
(add-hook 'c++-mode-hook 'my-c++-mode-hook)
(add-hook 'c-mode-hook 'my-c-mode-hook)
(add-hook 'scheme-mode-hook 'my-scheme-mode-hook)
(add-hook 'emacs-lisp-mode-hook 'my-lisp-mode-hook)
(add-hook 'lisp-mode-hook 'my-lisp-mode-hook)
(add-hook 'perl-mode-hook 'my-perl-mode-hook)

;; L'inverse de next-error
(defun previous-error (n)
  "Aller à l'erreur de compilation précédente et au code correspondant."
  (interactive "p")
  (next-error (- n)))
;; Divers...
(transient-mark-mode 1)
(setq mark-even-if-inactive t)
(setq visible-bell nil)
(setq next-line-add-newlines nil)
(setq compile-command "make")
(setq suggest-key-bindings nil)
(put 'eval-expression 'disabled nil)
(put 'narrow-to-region 'disabled nil)
(put 'set-goal-column 'disabled nil)

;; Recherche dans les archives Elisp
(autoload 'format-lisp-code-directory "lispdir" nil t)
(autoload 'lisp-dir-apropos "lispdir" nil t)
(autoload 'lisp-dir-retrieve "lispdir" nil t)
(autoload 'lisp-dir-verify "lispdir" nil t)

;; Mise en valeur syntaxique
(defun my-make-face (face colour &optional bold)
  "Créer une apparence pour une couleur, éventuellement en gras"
  (make-face face)
  (copy-face 'default face)
  (set-face-foreground face colour)
  (if bold (make-face-bold face))
  )
```

```
(if (eq window-system 'x)
    (progn
      (my-make-face 'blue "blue")
      (my-make-face 'red "red")
      (my-make-face 'green "dark green")
      (setq font-lock-comment-face 'blue)
      (setq font-lock-string-face 'bold)
      (setq font-lock-type-face 'bold)
      (setq font-lock-keyword-face 'bold)
      (setq font-lock-function-name-face 'red)
      (setq font-lock-doc-string-face 'green)
      (add-hook 'find-file-hooks 'font-lock-auto-mode-select)

      (setq baud-rate 1000000)
      (global-set-key "\C-cmm" 'menu-bar-mode)
      (global-set-key "\C-cms" 'scroll-bar-mode)
      (global-set-key [backspace] 'backward-delete-char)
                                      ;      (global-set-key [delete] 'delete-char)
      (standard-display-european t)
      (load-library "iso-transl")))

;; X11 ou PC écrivant directement à l'écran
(if window-system
    (progn
      ;;      (global-set-key [M-f1] 'hilit-repaint-command)
      ;;      (global-set-key [M-f2] [?\C-u M-f1])
      (setq hilit-mode-enable-list
            '(not text-mode c-mode c++-mode emacs-lisp-mode lisp-mode
                  scheme-mode)
            hilit-auto-highlight nil
            hilit-auto-rehighlight 'visible
            hilit-inhibit-hooks nil
            hilit-inhibit-rebinding t)
      (require 'hilit19)
      (require 'paren))
  (setq baud-rate 2400)   ; Pour les connections série lentes
  )

;; Terminal type TTY
(if (and (not window-system)
         (not (equal system-type 'ms-dos)))
    (progn
      (if first-time
          (progn
            (keyboard-translate ?\C-h ?\C-?)
            (keyboard-translate ?\C-? ?\C-h)))))

;; Sous UNIX
(if (not (equal system-type 'ms-dos))
    (progn
      (if first-time
          (server-start))))
```

```
;; Ajouter ici toute modification d'apparence des caractères
(add-hook 'term-setup-hook 'my-term-setup-hook)
(defun my-term-setup-hook ()
  (if (eq window-system 'pc)
      (progn
;; (set-face-background 'default "red")
 )))

;; Restaurer le "bureau" - le faire le plus tard possible
(if first-time
    (progn
      (desktop-load-default)
      (desktop-read)))

;; Indique que le fichier a été lu au moins une fois
(setq first-time nil)

;; Plus besoin de déboguer quoi que ce soit maintenant.
(setq debug-on-error nil)

;; C'est tout
(message "OK, %s%s" (user-login-name) ".")
```

6.4. Permettre à Emacs de comprendre de nouveaux langages

Bon, tout cela est très bien si vous ne voulez programmer qu'avec les langages déjà introduits dans le fichier `.emacs` (C, C++, Perl, Lisp et Scheme), mais que se passe-t-il quand un nouveau langage appelé "whizbang" fait son apparition, avec plein de nouvelles fonctionnalités attrayantes?

La première chose à faire est de regarder si whizbang s'accompagne de fichiers de configuration d'Emacs pour ce langage. Ces fichiers ont généralement comme extension `.el`, raccourci pour "Emacs Lisp". Par exemple, si whizbang est un logiciel porté pour FreeBSD, ces fichiers peuvent être repérés par:

```
% find /usr/ports/lang/whizbang -name "*.el" -print
```

et il faut les installer en le copiant dans le répertoire du "site Lisp" d'Emacs. Sous FreeBSD 2.1.0-RELEASE, c'est le répertoire `/usr/local/share/emacs/site-lisp`.

Ainsi par exemple, si la commande précédente nous donnait:

```
/usr/ports/lang/whizbang/work/misc/whizbang.el
```

nous le copierions alors comme suit:

```
% cp /usr/ports/lang/whizbang/work/misc/whizbang.el /usr/local/share/emacs/site-lisp
```

Décidons ensuite de l'extension que doivent avoir les fichiers source whizbang. Supposons, pour les besoins de l'exemple, qu'ils se terminent tous par `.wiz`. Il faut ajouter une entrée à notre fichier `.emacs`, pour être sûr qu'Emacs puisse utiliser les informations du fichier `whizbang.el`.

Recherchons l'entrée auto-mode-alist dans `.emacs` et ajoutons une ligne pour whizbang, par exemple:

```
...
("\\.lsp$" . lisp-mode)
("\\.wiz$" . whizbang-mode)
("\\.scm$" . scheme-mode)
...
```

Cela signifie qu'Emacs passera automatiquement en `whizbang-mode` à l'édition d'un fichier d'extension `.wiz`.

Juste après, il y a une entrée font-lock-auto-mode-list. Ajoutez-y `whizbang-mode` comme ceci:

```
;; Auto font-lock-mode
(defvar font-lock-auto-mode-list
  (list 'c-mode 'c++-mode 'c++-c-mode 'emacs-lisp-mode 'whizbang-mode 'lisp-mode 'perl-mode 'scheme-
  "Listes des modes à démarrer toujours en font-lock-mode")
```

Ce qui signifie qu'Emacs activera toujours le `font-lock-mode` (i.e., la mise en valeur syntaxique) à l'édition d'un fichier `.wiz`.

Cela suffit. S'il y a autre chose que vous voulez automatiser à l'ouverture d'un fichier `.wiz`, vous pouvez ajouter un `whizbang-mode hook` (voyez mon `my-scheme-mode-hook` pour avoir un exemple simple qui ajoute un `auto-indent` - indentation automatique).

7. A lire pour aller plus loin

- Brian Harvey et Matthew Wright *Simply Scheme* MIT 1994. ISBN 0-262-08226-8
- Randall Schwartz *Learning Perl* O'Reilly 1993 ISBN 1-56592-042-2
- Patrick Henry Winston et Berthold Klaus Paul Horn *Lisp (3rd Edition)* Addison-Wesley 1989 ISBN 0-201-08319-1
- Brian W. Kernighan et Rob Pike *The Unix Programming Environment* Prentice-Hall 1984 ISBN 0-13-937681-X
- Brian W. Kernighan et Dennis M. Ritchie *The C Programming Language (2nd Edition)* Prentice-Hall 1988 ISBN 0-13-110362-8
- Bjarne Stroustrup *The C++ Programming Language* Addison-Wesley 1991 ISBN 0-201-53992-6
- W. Richard Stevens *Advanced Programming in the Unix Environment* Addison-Wesley 1992 ISBN 0-201-56317-7
- W. Richard Stevens *Unix Network Programming* Prentice-Hall 1990 ISBN 0-13-949876-1

Notes

1. Dans le cas contraire, si vous l'exécutez sur la ligne de commande, il peut éventuellement planter - "*core dump*".

2. Pour être rigoureusement exact, `cc` convertit le code source en un *P-code* qui lui est propre, et ne dépend pas de la machine, et non en assembleur à ce stade.
3. Au cas où vous ne le sauriez pas, un tri par arbre binaire est une manière efficace d'ordonner des données, ce qui n'est pas le cas du tri à bulles.
4. Les raisons de cela se sont perdues dans les brumes de l'histoire.
5. Remarquez que nous n'avons pas utilisé l'indicateur `-o` pour préciser le nom de l'exécutable, celui-ci s'appellera donc `a.out`. Générer une version débogable appelée `foobar` est laissé à titre d'exercice aux soins du lecteur!
6. Ils n'utilisent pas la variante `MAKEFILE` parce que les noms en majuscules servent souvent à désigner les fichiers de documentation comme `README`.
7. Au moins, pas à moins que vous ne soyez prêt à les payer une somme astronomique.
8. De nombreux utilisateurs d'Emacs affectent à leur variable d'environnement `EDITOR` la valeur `emacsclient` de façon à ce que ce soit ce qui se produise chaque fois qu'ils ont besoin d'éditer un fichier.