Livre de chevet du développeur FreeBSD

Résumé

Bienvenue dans le livre de chevet du développeur

Version française de Frédéric Praca <frederic.praca@freebsd-fr.org>.

N.d.T.: La version française est publiée sur le serveur World Wide Web du groupe de traduction en langue française de la documentation de FreeBSD.

N.d.T.: Contactez la liste de diffusion du groupe d'utilisateurs francophones de FreeBSD <freebsd-questions@FreeBSD-fr.org> si vous voulez collaborer à la traduction.

La traduction de ce manuel est "en cours". Dans la table des matières ci-dessous:

- Les chapitres marqués de deux astérisques sont en cours de traduction.
- Les chapitres marqués de trois astérisques sont à traduire.
- L'astérisque simple est réservé aux chapitres et sections en cours de rédaction dans la version U.S.

Table des matières

I: Introduction . . . 4
1. Développer sous FreeBSD . . . 5
2. La vision BSD . . . 6
3. Survol de l'architecture . . . 7
4. L'agencement de /usr/src . . . 8
II: Les fondamentaux . . . 9
5. Outils de programmation . . . 10
5.1. Synopsis . . . 10
5.2. Introduction . . . 10
5.3. Introduction à la programmation . . . 10
5.4. Compiler avec `cc` . . . 13
5.5. Make . . . 20
5.6. Déverminer . . . 24
5.7. Utiliser Emacs comme environnement de développement . . . 28
5.8. Pour aller plus loin . . . 38
6. Programmation sécurisée . . . 39
6.1. Synopsis . . . 39
6.2. Méthodologie de développement sécurisé . . . 39
6.3. Dépassement de capacité . . . 39
6.4. Les problèmes liés à SetUID . . . 42
6.5. Limiter l'environnement de votre programme . . . 42
6.6. La confiance . . . 43
6.7. Les conditions de course . . . 44
III: Le noyau . . . 45
7. Histoire du noyau Unix . . . 46
8. Notes sur le verrouillage . . . 47
8.1. Les mutex . . . 47
8.2. Les verrous du gestionnaire de verrous (Lock Manager) . . . 51
8.3. Variables protégées atomiquement . . . 51
IV: Mémoire et mémoire virtuelle . . . 52
9. La mémoire virtuelle . . . 53
V: Système E/S (Entrées/Sorties) . . . 54
10. UFS . . . 55
VI: Communication InterProcessus (IPC) . . . 56
11. Les signaux . . . 57
VII: Le réseau . . . 58
12. Les prises . . . 59
VIII: Systèmes de fichiers en réseau . . . 60

13. AFS . . . 61
IX: Gestion du terminal . . . 62
14. Syscons . . . 63
X: Le son . . . 64
15. OSS . . . 65
XI: Pilotes de périphérique . . . 66
16. Ecrire des pilotes de périphériques pour FreeBSD . . . 67
16.1. Introduction . . . 67
16.2. L'éditeur de liens dynamiques du noyau - KLD . . . 67
16.3. Accéder au pilote d'un périphérique . . . 69
16.4. Les périphériques caractères . . . 69
16.5. Pilotes Réseau . . . 73
17. Les périphériques PCI . . . 74
17.1. Rechercher et rattacher . . . 74
17.2. Les ressources du bus . . . 78
18. Contrôleurs SCSI Common Access Method (CAM) ** . . . 81
18.1. En cours de traduction . . . 81
19. Périphériques USB * . . . 82
19.1. Introduction . . . 82
20. NewBus . . . 83
XII: Architectures . . . 84
21. IA-32 . . . 85
22. Alpha . . . 86
23. IA-64 . . . 87
XIII: Déverminage . . . 88
24. Truss . . . 89
XIV: Les couches de compatibilité . . . 90
25. Linux . . . 91
Bibligraphie . . . 92
Bibliographie . . . 93

Partie I: Introduction

Chapitre 1. Développer sous FreeBSD

Ce document a pour but de décrire FreeBSD comme une plateforme de développement, la vision de BSD, un survol de l'architecture, l'agencement de /usr/src, l'histoire, etc.

Merci d'adopter FreeBSD comme votre plateforme de développement ! Nous espérons qu'elle ne vous laissera pas tomber.

Chapitre 2. La vision BSD

Chapitre 3. Survol de l’architecture

Chapitre 4. L’agencement de /usr/src

Le code source complet de FreeBSD est disponible depuis notre base CVS publique. Le code source est normalement installé sous /usr/src qui contient les sous-répertoires suivants.

Répertoire	Description
bin/	Sources des fichiers de /bin
contrib/	Sources des fichiers des logiciels fournis ("contributed").
crypto/	Sources du DES
etc/	Sources des fichiers de /etc
games/	Sources des fichiers de /usr/games
gnu/	Utilitaires sous licence publique GNU
include/	Sources des fichiers de /usr/include
kerberosIV/	Sources de Kerbereros version IV
kerberos5/	Sources de Kerbereros version 5
lib/	Sources des fichiers de /usr/lib
libexec/	Sources des fichiers de /usr/libexec
release/	Fichiers requis pour la production d’une version stable de FreeBSD
sbin/	Sources des fichiers de /sbin
secure/	Sources de FreeSec
share/	Sources des fichiers de /sbin
sys/	Fichiers source du noyau
tools/	Outils utilisés pour la maintenance et les tests de FreeBSD
usr.bin/	Sources des fichiers de /usr/bin
usr.sbin/	Sources des fichiers de /usr/sbin

Partie II: Les fondamentaux

Chapitre 5. Outils de programmation

Ce chapître a été écrit par James Raynard. Les modifications pour le livre de chevet du développeur par Murray Stokely.

5.1. Synopsis

Ce document est une introduction à l'utilisation de quelques outils de programmation fournis avec FreeBSD, toutefois tout sera applicable à beaucoup d'autres versions d'Unix. Cette introduction *n'essaye* pas de décrire la programmation dans le détail. La plupart du document suppose que vous possédez peu ou pas de connaissances en programmation, espérant que les programmeurs trouveront un intérêt dans ce document.

5.2. Introduction

FreeBSD offre un excellent environnement de développement. Des compilateurs pour c, c++ et Fortran ainsi qu'un assembleur sont fournis avec le système de base, sans parler de l'interpréteur PERL ni des outils classiques Unix comme `sed` et `awk`. Si cela n'est pas suffisant, il y a encore plus de compilateurs et d'interpréteurs dans la collection des logiciels portés. FreeBSD est compatible avec les standards comme POSIX et C ANSI, aussi bien qu'avec son propre héritage BSD, aussi il est possible d'écrire des applications qui se compileront et s'exécuteront avec peu ou pas de modifications sur un grand nombre de plateformes.

Toutefois, toute cette puissance peut être plutôt écrasante au premier abord si vous n'avez jamais écrit de programmes sur une plateforme Unix auparavant. Ce document a pour but de vous aider à commencer, sans entrer trop loin dans des sujets plus avancés. L'intention est que ce document devrait vous donner assez de bases pour être capable de donner du sens à la documentation.

La majeure partie du document requiert peu ou pas de connaissance de la programmation, bien qu'il suppose une compétence de base dans l'utilisation d'Unix et dans la bonne volonté d'apprendre !

5.3. Introduction à la programmation

Un programme est un ensemble d'instructions qui disent à l'ordinateur de faire diverses choses; quelques fois, l'instruction qu'il a à exécuter dépend de ce qui s'est passé lors de l'exécution d'une instruction précédente. Cette section donne un aperçu des deux manières par lesquelles vous pouvez donner ces instructions, ou "commandes" comme elles sont habituellement nommées. Une façon utilise un *interpréteur*, l'autre un *compilateur*. Comme les langages humains sont trop difficiles à comprendre sans ambiguïté par un ordinateur, les commandes sont habituellement écrites dans un langage ou un autre spécialement conçus pour cet usage.

5.3.1. Les interpréteurs

Avec un interpréteur, le langage va avec un environnement où vous entrez des commandes à un invite de commandes et l'environnement les exécute pour vous. Pour des programmes plus compliqués, vous pouvez entrer les commandes dans un fichier et demander à l'interpréteur de

charger le fichier et d'exécuter les commandes qui sont à l'intérieur. Si quoique ce soit se passe mal, beaucoup d'interpréteurs vous enverrons dans un dévermineur pour vous aider à débusquer le problème.

L'avantage de cela est que vous pouvez voir les résultats de vos commandes immédiatement, et les erreurs peuvent être corrigées facilement. Le plus gros désavantage arrive quand vous voulez partager vos programmes avec d'autres personnes. Ils doivent avoir le même interpréteur ou bien vous devez avoir un moyen de leur donner, et ils doivent comprendre comment l'utiliser. Par ailleurs, les utilisateurs pourraient ne pas apprécier d'être renvoyés dans un dévermineur s'ils ont appuyé sur la mauvaise touche ! D'un point de vue performance, les interpréteurs peuvent utiliser beaucoup de mémoire et généralement ne génèrent pas un code aussi efficace que les compilateurs.

A mon avis, les langages interprétés sont le meilleur moyen pour démarrer si vous n'avez jamais programmé. Ce genre d'environnement se trouve typiquement avec des langages comme Lisp, Smalltalk, Perl et Basic. Il peut aussi être dit que l'interpréteur de commandes Unix (`sh`, `csh`) est lui-même un interpréteur, et beaucoup de gens écrivent en fait des "scripts" (procédures) pour l'interpréteur pour les aider dans diverses tâches "domestiques" sur leur machine. En effet, une partie de la philosophie d'origine d'Unix était de fournir plein de petits programmes utilitaires qui pouvaient être liés ensemble dans des procédures pour effectuer des tâches utiles.

5.3.2. Les interpréteurs disponibles avec FreeBSD

Voici la liste des interpréteurs qui sont disponibles sous la forme de logiciels pré-compilés pour FreeBSD, avec une brève description de quelques uns des langages interprétés les plus populaires.

Pour obtenir un de ces logiciels pré-compilés, tout ce que vous avez à faire est de cliquer sur le lien du logiciel et d'exécuter

```
# pkg_add nom du logiciel
```

en tant que super-utilisateur. Evidemment, vous aurez besoin d'un FreeBSD 2.1.0 ou plus en état de marche pour que le logiciel fonctionne !

BASIC

Abbréviation de "Beginner's All-purpose Symbolic Instruction Code" (code d'instruction symbolique tout usage pour le débutant). Développé dans les années 50 pour apprendre aux étudiants d'université à programmer et fourni avec tout ordinateur qui se respecte dans les années 80, BASIC a été le premier langage de programmation pour beaucoup de programmeurs. Il est aussi le fondement même du Visual Basic.

L'interpréteur Basic Bywater et l'interpréteur Basic de Phil Cockroft (anciennement Rabbit Basic) sont disponibles pour FreeBSD sous forme de logiciels pré-compilés

Lisp

Un langage qui a été développé à la fin des années 50 comme une alternative aux langages "dévoreurs de calculs" qui étaient très populaires à l'époque. Plutôt qu'être basé sur les nombres, Lisp est basé sur les listes; en fait le nom est l'abbréviation de "List Processing". Très populaire en IA (Intelligence Artificielle).

Lisp est un langage extrèmement puissant et sophistiqué , mais peut être assez lourd et peu maniable.

FreeBSD a GNU Common Lisp de disponible sous la forme d'un logiciel pré-compilé.

Perl

Très populaire auprès des administrateurs système pour la rédaction de procédures; aussi souvent utilisé sur les serveurs Internet pour l'écriture de procédures CGI.

La dernière version (version 5) est fournie avec FreeBSD.

Scheme

Un dérivé du Lisp qui est plutôt plus compact et plus propre que le Common Lisp. Populaire dans les universités étant suffisamment simple à apprendre aux étudiants comme premier langage , il possède un niveau d'abstraction suffisamment important pour être utilisé dans le travail de recherche.

On trouve pour FreeBSD les logiciels pré-compilés interpréteur Scheme Elk, l'interpréteur Scheme du MIT et l'interpréteur Scheme SCM.

Icon

Le langage de programmation Icon.

Logo

l'interpréteur Logo de Brian Harvey.

Python

Le langage orienté objet Python

5.3.3. Les compilateurs

Les compilateurs sont plutôt différents entre eux. Tout d'abord, vous écrivez votre code dans un fichier (ou des fichiers) en utilisant un éditeur de texte. Vous exécutez ensuite le compilateur et vérifiez qu'il accepte votre programme. S'il ne compile pas, grincez des dents et retournez à l'éditeur; s'il compile et vous donne un programme, vous pouvez exécuter ce dernier à l'invite de commande ou dans un dévermineur pour voir s'il fonctionne correctement.

Evidemment, ce n'est pas aussi direct que d'utiliser un interpréteur. Toutefois cela vous permet de faire beaucoup de choses qui sont difficiles ou même impossibles avec un interpréteur, comme écrire du code qui interagit de façon proche du système d'exploitation ou même d'écrire votre propre système d'exploitation ! C'est aussi utile si vous avez besoin d'écrire du code très efficace, étant donné que le compilateur peut prendre son temps et optimiser le code, ce qui ne serait pas acceptable avec un interpréteur. Et distribuer un programme écrit pour un compilateur est habituellement plus évident qu'un écrit pour un interpréteur-vous pouvez juste donner une copie de l'exécutable, en supposant que l'utilisateur possède le même système d'exploitation que vous.

Les langages compilés incluent Pascal, C et C. C et C sont des langages plutôt impitoyables et conviennent mieux aux programmeurs expérimentés; Pascal, d'autre part, a été conçu comme un langage éducatif, et est un assez bon langage pour commencer. Malheureusement, FreeBSD ne

possède aucun support Pascal, excepté pour un convertisseur Pascal vers C dans les logiciels portés.

Le cycle édition-compilation-exécution-déverminage étant relativement pénible lors de l'utilisation de programmes séparés, beaucoup de fabricants de compilateur ont produit des environnements de développement intégrés (ou IDE pour Integrated Development Environments et EDI dans la langue de Molière). FreeBSD ne possède pas d'EDI tel quel; toutefois il est possible d'utiliser Emacs à cet effet. Ceci est vu dans Utiliser Emacs comme environnement de développement.

5.4. Compiler avec `cc`

Cette section traite uniquement du compilateur GNU pour C et C++, celui-ci faisant partie du système FreeBSD de base. Il peut être invoqué soit par `cc` ou `gcc`. Les détails de production d'un programme avec un interpréteur varient considérablement d'un interpréteur à l'autre, et sont habituellement bien couverts par la documentation et l'aide en ligne de l'interpréteur.

Une fois que vous avez écrit votre chef d'oeuvre, la prochaine étape est de le convertir en quelque chose qui s'exécutera (espérons !) sur FreeBSD. Cela implique normalement plusieurs étapes, réalisées chacune par un programme différent.

1. Pré-traiter votre code source pour retirer les commentaires et faire d'autres trucs comme développer (expanser) les macros en C.
2. Vérifier la syntaxe de votre code source pour voir si vous avez obéi aux règles du langage. Si vous ne l'avez pas fait, il se plaindra !
3. Convertir le code source en langage assembleur- cela est vraiment proche du code machine, mais reste compréhensible par des humains. Prétendument.
4. Convertir le langage assembleur en code machine -ouais, on parle de bits et d'octets, de uns et de zéros.
5. Vérifier que vous avez utilisé des choses comme des fonctions et des variables globales de façon consistente. Par exemple, si vous avez appelé une fonction inexistente, le compilateur se plaindra.
6. Si vous essayez de produire un exécutable depuis plusieurs fichiers de code source, résoudre comment les faire fonctionner ensemble.
7. Résoudre comment produire quelque chose que le chargeur au vol du système sera capable de charger en mémoire et exécuter.
8. Finalement, écrire l'exécutable dans le système de fichiers.

Le mot *compilation* est souvent utilisé pour les étapes 1 à 4 seules-les autres correspondent au terme *liaison*. Quelquefois, l'étape 1 est appelée *pre-traitement* et les étapes 3-4 *assemblage*.

Heureusement, la plupart de ces détails vous sont cachés, étant donné que `cc` est un frontal qui s'occupe d'appeler tous les programmes avec les arguments corrects pour vous; tapez simplement

```
% cc foobar.c
```

compilera foobar.c avec toutes les étapes au-dessus. Si vous avez plus d'un fichier à compiler, faites simplement quelque chose comme

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

Notez que la vérification de syntaxe n'est que cela-vérifier la syntaxe. Cela ne vérifiera pas les erreurs de logique que vous pouvez avoir faites, comme mettre le programme en boucle infinie ou utiliser un tri à bulles quand vous devriez utiliser un tri binaire.

Il y a beaucoup d'options pour `cc`, qui qui se trouvent toutes dans les pages de manuel en ligne. Voici quelques unes des plus importantes, avec des exemples illustrant leur utilisation.

`-o nom_du_fichier`

Le nom de sortie du fichier. Si vous n'utilisez pas cette option, `cc` produira un exécutable appelé a.out.

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

`-c`

Compile juste le fichier, ne le lie pas. Utile pour les programmes jouets dont vous voulez juste vérifier la syntaxe, ou si vous utilisez un Makefile.

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

Cela va produire un *fichier objet* (pas un exécutable) appelé foobar.o. Celui-ci peut être lié ensuite avec d'autres fichiers objets pour produire un exécutable.

`-g`

Crée une version de déverminage de l'exécutable. Cela oblige le compilateur à placer des informations dans l'exécutable comme telle ligne du fichier source correspond à tel appel de fonction. Un déverminer peut utiliser cette information pour vous montrer le code source au fur et à mesure que vous avancez pas à pas dans le programme, ce qui est *très* utile; le désavantage est que toutes ces informations supplémentaires rendent le programme plus gros. Normalement, vous compilez avec l'option `-g` quand vous êtes en train de développer un programme et compilez ensuite une "version de production" sans `-g` quand vous êtes satisfait du fonctionnement.

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

Cela va produire une version de déverminage du programme foobar.

`-O`

Crée une version optimisée de l'exécutable. Le compilateur effectue différents trucs malins pour

essayer de produire un exécutable qui s'exécute plus rapidement que normal. Vous pouvez ajouter un nombre après l'option `-O` pour spécifier un niveau d'optimisation plus important, mais cela vous expose souvent aux bogues dans l'optimiseur du compilateur. Par exemple, la version de `cc` fournit avec la version 2.1.0 FreeBSD est connue pour produire du mauvais code avec l'option `-O2` dans certaines circonstances.

L'optimisation est habituellement activée uniquement lors de la compilation d'une version de production.

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

Cela va produire une version optimisée de foobar.

Les trois prochaines options vont forcer `cc` à vérifier que votre code est conforme au standard international en cours, se référant souvent à la norme ANSI, qui pour dire précisement est un standard ISO.

`-Wall`

Active tous les avertissements que les auteurs de `cc` pensent valoir le coup. Malgré le nom, il n'active pas tous les avertissements dont `cc` est capable.

`-ansi`

Désactive la plupart, mais pas toutes, des caractéristiques du C fournies par `cc` qui sont non-ANSI . Malgré le nom, cela ne garantit pas strictement que votre code sera conforme au standard.

`-pedantic`

Désactive *toutes* les caractéristiques de `cc` qui ne sont pas ANSI .

Sans ces options, `cc` vous permettrait d'utiliser quelques extensions au standard non-standards. Quelques unes de celles-ci sont très utiles, mais ne fonctionneront pas avec d'autres compilateurs -en fait, un des principaux buts du standard est de permettre aux gens d'écrire du code qui fonctionnera avec n'importe quel compilateur sur n'importe quel système. Cela est connu sous le nom de *code portable*.

Generalement, vous devriez essayer de faire votre code aussi portable que possible, sinon vous pourriez avoir a ré-écrire totalement votre programme plus tard pour le faire fonctionner autre part-et qui sait ce que vous utiliserez dans quelques années?

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

Cela produira un exécutable foobar après avoir vérifié que foobar.c est conforme au standard.

`-l librairie`

Spécifie une librairie de fonctions à utiliser lors de l'édition des liens.

L'exemple le plus commun de cela est lors de la compilation d'un programme qui utilise quelques fonctions mathématiques en C. A l'inverse de la plupart des plateformes, celles-ci se

trouvent dans une librairie standard du C et vous devez dire au compilateur de l'ajouter.

La règle est que si une librairie est appelée libquelque_chose.a, vous donnez à `cc` l'argument `-l quelque_chose`. Par exemple, la librairie des fonctions mathématiques est libm.a, aussi vous donnez à `cc` le paramètre `-lm`. Un "piège" habituel avec la librairie math est qu'elle doit être la dernière sur la ligne de commande.

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

Cela va lier les fonctions de la librairie math à l'interieur de foobar.

Si vous compilez du c++; vous devrez ajouter -lg++, ou -lstdc++ si vous utilisez FreeBSD 2.2 ou ultérieur, à la ligne de commande de `cc` pour lier avec les fonctions de la librairie c++. Alternativement, vous pouvez utiliser c++ plutôt que `cc`, qui fait tout cela pour vous. c++ peut aussi être invoqué par gcc++ sur FreeBSD.

```
% cc -o foobar foobar.cc -lg++     Pour FreeBSD 2.1.6 et antérieur
% cc -o foobar foobar.cc -lstdc++  Pour FreeBSD 2.2 et ultérieur
% c++ -o foobar foobar.cc
```

Chacune de ces commandes va produire un exécutable foobar à partir du fichier source c++ foobar.cc. Notez que, sur les systèmes Unix , les fichiers source c++ se terminent traditionnellement en .C, .cxx ou .cc, plutôt que le style MS-DOS .cpp (qui était déjà utilisé pour autre chose). `gcc` a utilisé cela pour trouver le type de compilateur à utiliser sur le fichier source; toutefois, cette restriction ne s'applique plus, aussi vous pouvez maintenant appeler vos fichiers c++ .cpp en toute impunité !

5.4.1. Questions et problèmes usuels sur `cc`

5.4.1.1. J'essaye d'écrire un programme qui utilise la fonction sin() et je reçois l'erreur suivante. Que cela signifie-t-il ?

Lors de l'utilisation des fonctions mathématiques comme `sin()`, vous devez dire à `cc` de lier avec la librairie math, comme :

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

5.4.1.2. J'ai écrit un programme simple pour m'exercer à l'utilisation de l'option -lm. Tout ce qu'il fait est d'élever 2,1 à la puissance 6.

Quand le compilateur voit que vous appelez une fonction, il vérifie s'il a déjà un prototype pour celle-ci. S'il ne l'a pas vu, il suppose que la fonction retourne un int, ce qui n'est absolument pas ce que vous voulez ici.

5.4.1.3. Alors comment puis-je le réparer?

Les prototypes des fonctions mathématiques sont dans math.h. Si vous incluez ce fichier, le compilateur sera capable de trouver le prototype et il arrêtera de faire des trucs étranges à vos calculs!

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

int main() {
...
```

Après avoir recompilé comme précédemment, exécutez :

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

Si vous utilisez quelques fonctions mathématiques que ce soit, incluez *toujours* math.h et n'oubliez pas de lier avec la librairie math.

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

Souvenez-vous, `cc` appellera l'exécutable a.out sauf si vous lui dites de faire autrement. Utilisez l'option `-o nomfichier`:

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

5.4.1.5. OK, j'ai un exécutable appelé foobar, je peux le voir en exécutant ls, mais quand je tape foobar à l'invite de commandes, la réponse est qu'il n'y a pas de tel fichier. Pourquoi le système ne le trouve pas?

A l'inverse de MS-DOS, Unix ne regarde pas dans le répertoire courant lorsqu'il essaye de trouver un exécutable que vous voulez exécuter, sauf si vous lui avez dit de le faire. Vous pouvez soit taper `./foobar`, ce qui signifie "exécute le fichier nommé foobar dans le répertoire courant", soit changer votre variable d'environnement `PATH` de façon à ce qu'elle ressemble à quelque chose comme

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

Le point à la fin signifie "regarde dans le repertoire courant s'il n'est dans aucun autre".

5.4.1.6. J'ai appelé mon exécutable test, mais rien ne se passe quand je l'exécute. Que se passe-t-il?

La plupart des systèmes Unix ont un programme appelé `test` dans /usr/bin et l'interpréteur prend celui-ci avant de vérifier dans le répertoire courant. Soit vous tapez

```
% ./test
```

soit vous choisissez un meilleur nom pour votre programme !!

5.4.1.7. J'ai compilé mon programme et il semble fonctionner au premier abord, puis il y a une erreur et le système a dit quelque chose comme core dumped. Que cela signifie-t-il?

Le nom *core dump* date des tous premiers jours d'Unix, quand les machines utilisaient la mémoire centralepour stocker les données. Simplement, si le programme a échoué sous certaines conditions, le système va écrire le contenu de la mémoire centrale sur le disque dans un fichier appelé core, que le programmeur peut ensuite examiner de près pour trouver ce qui s'est mal passé.

5.4.1.8. Fascinant, mais que suis-je supposé faire ?

Utilisez `gdb` pour analyser le fichier core (voir Déverminer).

5.4.1.9. Quand mon programme a généré un fichier core, il a parlé d'une erreur segmentation fault. Qu'est-ce que c'est ?

Cela signifie simplement que votre programme a essayé d'effectuer une opération illégale sur la mémoire; Unix est conçu pour protéger le système d'exploitation et les autres programmes des programmes crapuleux.

Les causes habituelles de cela sont :

- Essayer d'écrire dans un pointeur NULL, par exemple

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

- Utiliser un pointeur qui n'a pas été initialisé, par exemple

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

  Le pointeur va avoir une valeur aléatoire qui avec de la chance, pointera dans une zone de la mémoire qui n'est pas disponible pour votre programme et le noyau va tuer votre programme avant qu'il ne fasse des dommages. Si vous êtes malchanceux, il pointera quelque part dans votre propre programme et altèrera une de vos structures de données, faisant planter votre programme mystérieusement.

- Essayer d'accèder la mémoire au-delà de la fin d'un tableau, par exemple

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

- Essayer de stocker quelque chose dans la mémoire en lecture seule, par exemple

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

  Les compilateurs Unix mettent souvent les chaînes comme `"Ma chaine"` dans des zones de mémoire en lecture seule.

- Faire des trucs sales avec `malloc()` et `free()`, par exemple

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

  ou

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

Faire une de ces fautes ne conduit pas toujours à une erreur, mais elles sont toujours de mauvais entrainements. Certains systèmes et compilateurs sont plus tolérants que d'autres, ce qui explique pourquoi des programmes qui fonctionnent bien sur un système peuvent planter si vous les essayer sur un autre.

5.4.1.10. Des fois quand je reçoit une erreur core dump, il est précisé bus error. Il est dit dans mon livre Unix qu'il s'agit d'un problème matériel mais l'ordinateur semble toujours fonctionner. Est-ce vrai ?

Non, heureusement non (sauf si bien sûr vous avez réellement un problème matériel…). Cela est habituellement une manière de dire que vous avez accédé à la mémoire d'une façon que vous n'auriez pas dû.

5.4.1.11. Toute cette affaire de fichier core semble être assez utile, si je peux le faire apparaître quand je le désire. Puis-je faire cela, ou dois-je attendre la prochaine erreur ?

Oui, ouvrez une autre console ou xterm, faites

```
% ps
```

pour trouver l'identifiant du processus de votre programme, et faites

```
% kill -ABRT identifiant
```

où *identifiant* est l'identifiant du processus que vous avez trouvé.

Ceci est utile si votre programme est bloqué dans une boucle infinie, par exemple. Si votre programme arrive à bloquer le signal SIGABRT, il y a d'autres signaux qui ont des effets similaires.

Alternativement, vous pouvez créer un fichier core depuis votre programme, en appelant la fonction `abort()`. Voir la page de manuel en ligne de abort(3) pour en savoir plus.

Si vous voulez créer un fichier core depuis l'extérieur de votre programme, mais ne voulez pas que le processus s'arrêt,vous pouvez utiliser le programme `gcore`. Voir la page de manuel en ligne de gcore(1) pour plus d'informations.

5.5. Make

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

Quand vous travaillez sur un programme simple avec seulement un ou deux fichiers source, taper

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

n'est pas si mal, mais cela devient rapidement fastidieux quand il y a plusieurs fichiers-et cela peut prendre du temps à compiler aussi.

Une façon de contourner cela est d'utiliser les fichiers objet et de recompiler le fichier source seulement si le code source a changé. Aussi, nous pourrions avoir quelque chose comme ça:

```
% cc fichier1.o fichier2.o ··· fichier37.c ···
```

si nous avions changé le fichier fichier37.c mais aucun des autres depuis la dernère compilation. Cela pourrait accélerer assez bien la compilation mais cela ne resoud pas le problème de la frappe au clavier.

Ou nous pourrions écrire une procédure pour résoudre ce problème de frappe, mais celle-ci devrait tout re-compiler, devenant ainsi inefficace sur un gros projet.

Que se passe-t-il si nous avons des centaines de fichiers source ? Que se passe-t-il si nous travaillons dans une équipe avec d'autres personnes qui oublient de nous dire quand ils ont changé un de leurs fichiers source que nous utilisons ?

Peut-être pourrions nous mettre ensemble les deux solutions et écrire quelque chose comme une procédure qui contiendrait quelque règle magique disant quand notre fichier source doit être compilé. Maintenant, tout ce dont nous avons besoin est un programme qui comprend ces règles, alors que c'est trop compliqué pour l'interpréteur.

Ce programme s'appelle `make`. Il lit dans un fichier, appelé un *makefile*, qui lui dit comment les différents fichiers dépendent les uns des autres, et détermine quels fichiers ont besoin d'être recompilés et quels n'en ont pas besoin. Par exemple, une règle pourrait dire quelque chose comme "si fromboz.o est plus ancien que fromboz.c, cela signifie que quelqu'un a dû changer fromboz.c, aussi il a besoin d'être recompilé." Le makefile possède aussi des règles pour dire à `make` *comment*

re-compiler un fichier source, en faisant ainsi un outil encore plus puissant.

Les Makefiles sont typiquement stockés dans le même répertoire que le source auxquels il s'appliquent, et peuvent être appelés makefile, Makefile ou MAKEFILE. La plupart des programmeurs utilise le nom Makefile, celui-ci se trouvant proche du début de la liste du contenu du répertoire où il peut être facilement vu.

5.5.2. Exemple d'utilisation de `make`

Voici un exemple très simple de Makefile:

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

Il consiste en deux lignes, une ligne de dépendance et une ligne de création.

La ligne de dépendance ici consiste en le nom du programme (connu comme *cible*), suivi de deux-points puis un espace et enfin le nom du fichier source. Quand `make` lit cette ligne, il vérifie si foo existe; s'il existe, il compare la date à laquelle foo a été modifié la dernière fois avec la date de dernière modification de foo.c. Si foo n'existe pas ou est plus ancien que foo.c, il regarde la ligne de création pour trouver ce qu'il doit faire. En d'autres termes, il s'agit de la règle à utiliser quand foo.c a besoin d'être re-compilé.

La ligne de création commence avec un tab (appuyez sur la touche `tab`) suivi de la commande que vous taperiez pour créer foo si vous deviez le faire à l'invite de commandes. Si foo n'est pas à jour ou n'existe pas, `make` exécute alors cette commande pour le créer. En d'autres termes, il s'agit de la règle permettant à make de re-compiler foo.c.

Aussi, quand vous tapez `make`, il vérifiera que foo est à jour en respect de vos derniers changements sur foo.c. Ce principe peut être étendu à des Makefiles de plusieurs centaines de cibles-en fait, sur FreeBSD, il est possible de compiler un système d'exploitation entier en tapant juste `make world` dans le répertoire approprié !

Une autre propriété utile des makefiles est que les cibles n'ont pas nécessairement besoin d'être des programmes. Par exemple, nous pourrions avoir un Makefile qui ressemble à cela:

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

install:
    cp foo /home/moi
```

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

```
% make cible
```

`make` ira seulement voir cette cible et ingorera les autres. Par exemple, si nous tapons `make foo` avec

le Makefile du dessus, `make` ignorera la cible install.

Si nous tapons juste `make`, `make` regardera toujours la première cible et s'arrêtera sans regarder aucune autre. Aussi, si nous avions tapé `make` seul, `make` serait juste allé à la cible foo, aurait recompilé foo si nécessaire et se serait arrêté sans aller à la cible install.

Notez que la cible install ne dépend pour l'instant de rien ! Cela signifie que la commande qui suit est toujours exécutée lorsque nous essayons de créer cette cible en tapant `make install`. Dans ce cas, `make` va copier foo dans le répertoire de l'utilisateur. Cela est souvent utilisé par les Makefiles des applications, ainsi l'application peut être installée dans le répertoire correct quand elle a été correctement compilée

Il s'agit d'un sujet légèrement embrouillant à essayer et expliquer. Si vous ne comprenez pas bien comment `make` fonctionne, la meilleure chose à faire est d'écrire un petit programme comme "bonjour monde" et un fichier Makefile comme le précédent et de le tester. Ensuite continuez en utilisant plus d'un fichier source ou en ayant un fichier source incluant un fichier d'en-tête. La commande `touch` est très utile ici-elle change la date sur un fichier sans avoir à l'éditer.

5.5.3. Les Makefiles de FreeBSD

Les Makefiles peuvent être plutôt compliqués à écrire. Heureusement, les systèmes BSD comme FreeBSD sont fournis avec des fichiers très puissants comme partie intégrante du système. Un très bon exemple est le système des logiciels portés. Voici la partie essentielle d'un Makefile typique des logiciels portés:

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

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

Maintenant, si nous allons dans le répertoire de ce logiciel porté et tapons `make`, la chose suivante se passe :

1. Une vérification est faite pour voir si le code source de ce logiciel porté est déjà dans le système.
2. Si celui-ci n'y est pas, une connexion FTP à l'URL dans MASTER_SITES est faite pour télécharger le source.
3. La somme de contrôle (`checksum`) du source est calculée et comparée avec celle d'une bonne et connue copie du source. Cela est fait pour être sûr que le source n'a pas été corrompu pendant le transfert.
4. Tout changement requis pour faire fonctionner le source sur FreeBSD est appliqué- cela est connu sous le nom de *correctif*.
5. Toute configuration spéciale nécessaire pour le source est faite. (Beaucoup de distributions de programmes Unix essaye de fonctionner quelle que soit la version d'Unix sur laquelle elles sont compilées et quelles que soient les caractéristiques optionnelles qui sont présentes-c'est ce qui se trouve dans le scénario des logiciels portés pour FreeBSD).

6. Le code source pour ce programme est compilé. En effet, nous changeons de répertoire pour le répertoire où le source a été décompressé et faisons `make`-le fichier Makefile du programme contient les informations nécessaires pour construire le programme.
7. Nous avons maintenant une version compilée du programme. Si nous le désirons, nous pouvons le tester maintenant; quand nous sommes confiant dans le programme, nous pouvons taper `make install`. Cela va installer le programme et ses fichiers de soutien nécessaires au bon endroit; une entrée est aussi créée dans la `base de données des logiciels pré-compilés`, ainsi le logiciel porté peut être facilement désinstallé plus tard si nous changeons d'avis sur ce programme.

Maintenant je pense que vous serez d'accord que c'est plus impressionnant qu'une procédure de quatre lignes !

Le secret réside dans la dernière ligne qui dit à `make` de regarder dans le Makefile système appelé bsd.port.mk. Il est facile de fermer les yeux sur cette ligne mais c'est ici que tous les trucs forts se passent-quelqu'un a écrit un Makefile qui dit à `make` de faire tout ce qu'il y a au-dessus (plus un couple d'autres choses que je n'ai pas mentionnées, comme la gestion des erreurs) et n'importe qui peut avoir accès à cela simplement est ajoutant une simple ligne dans son propre fichier Makefile !

Si vous voulez jeter un regard sur ces Makefiles système, ils se trouvent /usr/shared/mk mais il est probablement mieux d'attendre un moment jusqu'à ce que vous ayez un peu d'entrainement avec les Makefiles car ceux-ci sont très compliqués (et si vous les lisez, soyez sûr d'avoir un thermos de café fort à portée de main !)

5.5.4. Utilisations plus avancées de `make`

`Make` est un outil très puissant et peut faire beaucoup plus que le simple exemple précédent ne l'a montré. Malheureusement, il y a différentes versions de `make` et elles diffèrent considérablement. Le meilleur moyen d'apprendre ce qu'elles peuvent faire est probablement de lire la documentation-heureusement cette introduction vous donnera la base à partir de laquelle vous pourrez faire cela.

La version de `make` fournies avec FreeBSD est le Berkeley make(make de Berkeley); il y a un cours d'instruction pour celui-ci dans /usr/shared/doc/psd/12.make. Pour le voir, faites

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

dans ce répertoire.

Beaucoup d'applications dans les logiciels portés utilisent GNU make, qui possède un très bon ensemble de page d'"info". Si vous avez installé un de ces logiciels portés, GNU make aura été automatiquement installé sous le nom de `gmake`. Il est aussi disponible comme logiciel porté ou logiciel pré-compilé seul.

Pour voir les pages d'info pour GNU make, vous devrez editer le fichier dir dans le répertoire /usr/local/info pour ajouter une entrée pour celui-ci. Cela implique d'ajouter une ligne

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

au fichier. Une fois que vous avez fait ceci, vous pouvez taper `info` et ensuite sélectionner make dans le menu (ou dans Emacs, faites `C-h i`).

5.6. Déverminer

5.6.1. Le déverminer

Le déverminer fourni avec FreeBSD est appelé `gdb` (GNU debugger). Vous pouvez le démarrer en tapant

```
% gdb nomprog
```

bien que la plupart des gens préfèrent le démarrer au sein d'Emacs. Vous pouvez faire cela avec:

```
M-x gdb RET nomprog RET
```

Utiliser un déverminer vous permet d'exécuter le programme dans des circonstances plus contrôlées. Typiquement, vous pouvez exécuter le programme ligne à ligne, inspecter la valeur des variables, changer cette dernière, dire au déverminer d'exécuter jusqu'à un certain point puis de s'arrêter etc... Vous pouvez même vous brancher sur un programme en fonctionnement, ou charger un fichier core pour enquêter sur le plantage du programme. Il est même possible de déverminer le noyau, quoique ce soit un peu plus rusé que de déverminer des applications utilisateur dont nous discuterons dans cette section.

`gdb` dispose d'une assez bonne aide en ligne comme d'un ensemble de pages d'info, aussi cette section va se concentrer sur quelques commandes basiques.

Finalement, si vous trouvez son interface texte non fonctionnelle, il y a une interface graphique pour celui-ci, xxgdb, dans la collection des logiciels portés.

Cette section a pour but d'être une introduction à l'utilisation de `gdb` et ne couvre pas les sujets très spécialisés comme le déverminage du noyau.

5.6.2. Exécuter un programme dans le déverminer

Vous devrez avoir compilé le programme avec l'option `-g` pour avoir la meilleure utilisation de `gdb`. Il fonctionnera sans mais vous ne verrez que le nom de la fonction dans laquelle vous vous trouvez plutôt que son code source. Si vous voyez une ligne comme:

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

quand `gdb` démarre, 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 dévermineur de passer le code préliminaire d'initialisation du programme et de démarrer au début de votre code. Maintenant tapez `run` pour démarrer le programme-cela va démarrer au début du code d'initialisation et ensuite s'arrêtera lors de l'appel à `main()`. (Si vous vous êtes toujours demandé où `main()` était appelé, maintenant vous le savez !).

Vous pouvez maintenant vous déplacer dans le programme ligne par ligne en pressant `n`. Si vous arrivez à l'appel d'une fonction, vous pouvez entrer dans celle-ci en appuyant sur `s`. Une fois que vous êtes dans l'appel de la fonction, vous pouvez retourner dans le code appelant en appuyant sur `f`. Vous pouvez aussi utiliser `up` et `down` pour avoir une vue rapide de l'appelant.

Voici un exemple simple de comment détecter une erreur dans un programme avec `gdb`. Voici notre programme (avec une erreur délibérée):

```
#include <stdio.h>

int bazz(int anint);

main() {
    int i;

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

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

Le programme met i à `5` et le passe à une fonction `bazz()` qui imprime le nombre que nous lui avons donné.

Puis nous compilons et exécutons le programme obtenu

```
% cc -g -o temp temp.c
% ./temp
C'est mon programme
Vous m'avez fourni 4231
```

Ce n'était pas ce que nous attendions ! Il est temps de voir ce qui se 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                passe le code d'initialisation
Breakpoint 1 at 0x160f: file temp.c, line 9.    gdb met un point d'arrêt sur main()
(gdb) run                   Exécute 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                     Va à la ligne suivante
C'est mon programme             Le programme écrit
(gdb) s                     entre dans bazz()
bazz (anint=4231) at temp.c:17          gdb montre la pile
(gdb)
```

Arrêtons-nous une minute! Comment anint a eu la valeur `4231`? Ne l'avons-nous pas mis à `5` dans `main()`? Remontons dans `main()` et regardons.

```
(gdb) up                    Remonte la pile des appels
#1  0x1625 in main () at temp.c:11      gdb montre la pile
(gdb) p i                   Montre la valeur de i
$1 = 4231                   gdb montre 4231
```

Oh ! En regardant dans le code, nous avons oublié d'initialiser i. Nous aurions dû mettre

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

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

mais nous n'avions pas mis la ligne `i=5;`. Comme nous n'avons pas initialisé i, il a pris le nombre se trouvant dans la zone de mémoire quand le programme a démarré, ce qui dans ce cas était `4231`.

`gdb` montre la pile chaque fois que nous entrons ou sortons d'une fonction, même si nous avons utilisé `up` et `down` pour nous déplacer dans la pile des appels. Cela montre le nom de la fonction et les valeurs de ses arguments, ce qui nous aide à garder une trace d'où nous sommes et de ce qui se passe. (La pile est une zone de stockage où le programme stocke les informations sur les arguments passés aux fonctions et où il doit aller quand il revient d'une fonction).

5.6.3. Examiner un fichier core

Un fichier core est basiquement un fichier qui contient l'état complet du processus quand il s'est planté. Dans "le bon vieux temps", les programmeurs devait imprimer des listings en hexadécimal de fichiers core et transpirer sur leur manuels de code machine, mais la vie est maintenant un peu plus facile. Par chance, sous FreeBSD et les autres systèmes 4.4BSD, un fichier core est appelé nomprog.core plutôt que juste core, pour mieux savoir à quel programme appartient un fichier

core.

Pour examiner un fichier core, démarrez `gdb` de façon habituel. Plutôt que de taper `break` ou `run`, tapez

```
(gdb) core nomprog.core
```

Si vous n'êtes pas dans le même répertoire que le fichier core, vous devrez faire `dir /path/to/core/file` d'abord.

Vous devriez voir quelque chose comme cela:

```
% 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 (anint=0x5) at temp.c:17
(gdb)
```

Dans ce cas, le programme a été appelé a.out, aussi le fichier core s'appelle a.out.core. Nous pouvons voir que le programme s'est planté car il a essayé d'accèder à une zone dans la mémoire qui n'était pas disponible dans la fonction appelée `bazz`.

Quelquefois il est utile de pouvoir voir comment une fonction a été appelée car le problème peut avoir eu lieu bien avant dans la pile des appels dans un programme complexe. La commande `bt` demande à `gdb` d'afficher une trace inverse de la pile des appels:

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

La fonction `end()` est appelée lorsque le programme se plante; dans ce cas, la fonction `bazz()` a été appelée `main()`.

5.6.4. Se brancher sur un programme en cours d'exécution

Une des plus belles caractéristiques de `gdb` est qu'il peut se brancher sur un programme qui s'exécute déjà. Bien sûr, cela suppose que vous ayez les privilèges suffisants pour le faire. Un problème habituel est quand vous vous déplacez dans un programme qui se dédouble et que vous voulez tracer le programme fils cependant le dévermineur ne vous laissera seulement tracer le

père.

Ce que vous devez faire est de démarrer un autre `gdb`, utiliser `ps` pour trouver l'ID du processus fils et faire

```
(gdb) attach identifiant_processus
```

dans `gdb`, et déverminer ensuite comme d'habitude.

"C'est tout simple," pensez-vous certainement," mais pendant le temps que je faisais ça, le processus fils sera déjà parti loin". Ne vous en faites pas, noble lecteur, voici comment faire (avec l'appui des pages d'info de `gdb`):

```
...
if ((pid = fork()) < 0)     /* _Toujours_ verifier cela */
    error();
else if (pid == 0) {        /* le fils */
    int PauseMode = 1;

    while (PauseMode)
        sleep(10);  /* Attendre jusqu'a ce que quelqu'un se brache sur nous */
    ...
} else {            /* le pere */
    ...
```

Maintenant tout ce que nous avons à faire est de nous brancher sur le processus fils, de mettre PauseMode à `0` et d'attendre que l'appel à la fonction `sleep()` retourne !

5.7. Utiliser Emacs comme environnement de développement

5.7.1. Emacs

Malheureusement, les systèmes Unix ne sont pas fournis avec des sortes d'environnements de développement intégrés tout-ce-que-vous-avez-toujours-voulu-et-beaucoup-plus-dans-un-ensemble-gigantesque que d'autres sytèmes ont. Toutefois, il est possible de se faire son propre environnement. Cela n'est pas forcément aussi joli et il peut ne pas être autant intégré mais vous pouvez le personnaliser comme vous voulez. Et c'est gratuit. Et vous en avez les sources.

La clé de tout cela est Emacs. Maintenant il y a des gens qui le détestent, mais beaucoup l'aiment. Si vous êtes un du premier groupe, j'ai peur que cette section ait peu d'intérêt pour vous. Vous aurez besoin d'une quantité moyenne de mémoire pour le faire fonctionner-Je recommenderai 8Mo en mode texte et 16Mo dans X pour avoir un minimum de performances.

Emacs est basiquement un éditeur hautement personnalisable -en effet, il a été personnalisé au point de ressembler plus à un système d'exploitation qu'à un éditeur! Beaucoup de développeurs et d'administrateurs système passent en fait pratiquement tout leur temps à travailler dans Emacs, en

ne le quittant qu'à leur déconnexion.

Il est impossible de dire tout ce qu'Emacs peut faire ici, mais voici quelques unes des caractéristiques d'intérêt pour les développeurs:

- Un très puissant éditeur, permettant le chercher-remplacer sur les chaînes et les expressions régulières (motifs), sauter à la fin ou au début d'un bloc, etc, etc.
- Menus déroulants et aide en ligne.
- Colorisation syntaxique en fonction du langage et indentation.
- Totalement personnalisable.
- Vous pouvez compiler et déverminer des programmes dans Emacs.
- Sur erreur de compilation, vous pouvez aller directement à la ligne de code source fautive.
- Une interface amicale au programme `info` utilisé pour lire la documentation hypertexte GNU, incluant la documentation sur Emacs elle-même.
- Une interface agréable à `gdb`, vous permettant de voir le code source au fur et à mesure que vous vous déplacez dans votre programme.
- Vous pouvez lire les nouvelles Usenet et envoyer des e-mails pendant que votre programme est en compilation.

Et sans doute beaucoup plus que je n'ai survolé.

Emacs peut être installé sur FreeBSD en utilisant Emacs le logiciel porté Emacs.

Une fois installé, démarrez-le et faites `C-h t` pour lire un cours sur Emacs-cela signifie maintenir la touche `control`, presser `h`, relâcher la touche `control` et presser `t`. (Alternativement, vous pouvez utiliser la souris pour sélectionner Emacs Tutorial dans le menu **Help**).

Bien que Emacs possède des menus, il est valable d'apprendre les raccourcis clavier, étant plus rapide quand vous éditez quelque chose d'appuyer sur un couple de touches que de reprendre la souris et de cliquer au bon endroit. Et, quand vous discutez avec des utilisateurs expérimentés d'Emacs, vous trouverez qu'ils parlent souvent de choses comme "M-x replace-s RET foo RET bar RET" aussi il est utile de savoir ce que cela veut dire. Et dans tous les cas, Emacs possède beaucoup trop de fonctions pour qu'elles soient dans les barres de menus.

Heureusement, il est assez simple de récupérer les raccourcis clavier car ils sont affichés à côté des éléments des menus déroulants. Mon conseil est d'utiliser un élément de menu pour, disons, ouvrir un fichier jusqu'à ce que vous sachiez comment cela fonctionne et que quand vous vous sentez à l'aise avec, essayez `C-x C-f`. Quand vous serez content avec ça, passez à une autre commande du menu.

Si vous ne pouvez pas vous rappeler de ce qu'une combinaison de touches particulières fait, sélectionnez Describe Key dans le menu **Help** et tapez-la-Emacs vous dira ce qu'elle fait. Vous pouvez aussi utiliser l'élément de menu Command Apropos pour trouver toutes les commandes qui contiennent un mot particulier, avec leur raccourci clavier.

De cette manière, l'expression ci-dessus signifie maintenir la touche `Meta`, taper `x`, relâcher la touche `Meta`, taper `replace-s` (raccourci pour `replace-string`- une autre caractéristique d'Emacs est que

vous pouvez abréger les commandes), appuyer sur la touche `Entrée`, taper `foo` (la chaîne que vous voulez remplacer), presser la touche `Entrée`, taper `bar` (la chaîne que vous voulez substituer à `foo`) puis appuyer sur `Entrée` une dernière fois. Emacs va alors faire l'opération chercher-remplacer que vous avez demandé.

Si vous vous demandez ce qu'est cette touche `Meta`, il s'agit d'une touche spéciale que beaucoup de stations de travail Unix possèdent. Malheureusement, les PC n'en ont pas, aussi c'est habituellement la touche `alt` (ou si vous n'avez pas de chance, la touche `échap`).

Oh, et pour sortir d'Emacs, faites `C-x C-c` (ce qui signifie maintenir la touche `control` appuyée, appuyer `c` et relâcher la touche `control`. Si vous avez des fichiers non sauvegardés ouverts, Emacs vous demandera si vous voulez les sauvegarder. (Ignorez le bout de documentation où il est dit que `C-z` est la manière habituelle de quitter Emacs- qui quitte Emacs en le laissant tourner en tâche de fond et qui n'est vraiment utile que si vous avez un système sans terminal virtuel).

5.7.2. Configurer Emacs

Emacs fait des choses merveilleuses; une partie est intégrée directement, une autre doit être configurée.

Plutôt que d'utiliser un macro langage propriétaire pour la configuration, Emacs utilise une version du Lisp spécialement adaptée pour les éditeurs, connue sous le nom d'Emacs Lisp. Celui-ci peut être assez utile si vous voulez poursuivre et apprendre quelque chose comme le Common Lisp, car il est considérablement plus petit que le Common Lisp (bien que déjà assez gros!).

La meilleure façon d'apprendre l'Emacs Lisp est de télécharger le cours d'Emacs

Toutefois, il n'y a pas besoin de connaître le Lisp pour commencer la configuration d'Emacs, car j'ai inclus un exemple de fichier .emacs qui devrait être suffisant pour commencer. Copiez juste celui-ci dans votre répertoire utilisateur et redémarrez Emacs si celui-ci s'exécute, il lira les commandes du fichier et (si tout va bien!) vous donnera une configuration basique utile.

5.7.3. Un fichier exemple .emacs

Malheureusement, il y a beaucoup trop de choses ici pour les expliquer en détail; toutefois, il y a un ou deux points qui valent d'être mentionnés.

- Tout ce qui commence avec un `;` est un commentaire et est ignoré par Emacs.
- A la première ligne, le `-- Emacs-Lisp --` est tel que vous pouvez éditer le fichier .emacs lui-même à l'intérieur d'Emacs et d'obtenir tous les fantaisistes dispositifs pour l'édition en Emacs Lisp. Emacs essaye habituellement de deviner cela en se basant sur le nom du fichier, et ne trouvera peut-être pas pour .emacs.
- La touche `tab` est liée à la fonction d'indentation dans certains modes, aussi quand vous l'enfoncez, cela va indenter la ligne courante de code. Si vous voulez mettre un caractère tab dans quoique ce soit que vous tapiez, maintenez la touche `control` enfoncée pendant que vous appuyez sur `tab`.
- Ce fichier supporte la colorisation de syntaxe pour les langages C, C++, Perl, Lisp et Scheme en devinant le langage par leur nom.

- Emacs possède déjà une fonction pré-définie appelée `next-error`. Dans la fenêtre de sortie d'une compilation, cela vous permet de vous déplacer d'une erreur de compilation à la suivante en faisant `M-n`; nous définissons une fonction complémentaire `previous-error`, qui vous permet d'aller à l'erreur précédente en faisant `M-p`. Le plus beau dispositif de tous est que `C-c C-c` va ouvrir le fichier source dans lequel l'erreur a eu lieu et sautera à la ligne appropriée.
- Nous autorisons la capacité d'Emacs à agir comme un serveur ainsi si vous faites quelque chose en dehors d'Emacs et voulez éditer un fichier, tapez juste

  ```
  % emacsclient nomfichier
  ```

  et alors vous pouvez éditer le fichier dans votre Emacs!

Exemple 1. Un fichier exemple .emacs

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

;; Ce fichier est concu pour etre reevalue, utiliser la variable first-time
;; pour eviter tout probleme.
(defvar first-time t
  "Valeur signifiant que c'est la premiere fois que .emacs a ete evalue"
  )

;; Meta
(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)

;; Function keys
(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)

;; Keypad bindings
(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")   ; Control tab quotes a tab.
(setq backup-by-copying-when-mismatch t)

;; Traite 'y' ou <CR> comme yes, 'n' comme 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)

;; Charge les ajouts
(require 'desktop)
(require 'tar-mode)

;; Diff mode sympa
(autoload 'ediff-buffers "ediff" "Intelligent Emacs interface to diff" t)
(autoload 'ediff-files "ediff" "Intelligent Emacs interface to diff" t)
(autoload 'ediff-files-remote "ediff"
  "Intelligent Emacs interface to 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)))
```

```
;; Mode de verrouillage automatique de la police de caracteres
(defvar font-lock-auto-mode-list
  (list 'c-mode 'c++-mode 'c++-c-mode 'emacs-lisp-mode 'lisp-mode 'perl-mode
'scheme-mode)
  "List of modes to always start in font-lock-mode")

(defvar font-lock-mode-keyword-alist
  '((c++-c-mode . c-font-lock-keywords)
    (perl-mode . perl-font-lock-keywords))
  "Associations between modes and keywords")

(defun font-lock-auto-mode-select ()
  "Automatically select font-lock-mode if the current major mode is
in font-lock-auto-mode-list"
  (if (memq major-mode font-lock-auto-mode-list)
      (progn
    (font-lock-mode t))
    )
  )

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

;; New dabbrev stuff
;(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)
;; Style d'indentation 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))

;; Ajoute tout le reste...
(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)

;; Le complement a next-error
(defun previous-error (n)
  "Visit previous compilation error message and corresponding source code."
  (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 des 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)

;; Mode de verrouillage de police
(defun my-make-face (face colour &optional bold)
  "Create a face from a colour and optionally make it bold"
  (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 utilisant les ecritures directes a l'ecran
(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)         ; For slow serial connections
  )

;; Terminal de 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))))

;; Add any face changes here
(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")
    )))

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

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

;; Pas besoin de deverminer quoique ce soit maintenant

(setq debug-on-error nil)

;; Tout est fait
(message "All done, %s%s" (user-login-name) ".")
```

5.7.4. Etendre la palette de langages qu'Emacs comprend

Maintenant, Emacs est très bien si vous voulez seulement programmer dans des langages déjà fournis dans le fichier .emacs (C, C++, Perl, Lisp et Scheme), mais qu'arrive-t-il si un nouveau langage appelé "whizbang" sort, plein d'excitantes fonctionnalités ?

La première chose à faire est de savoir si whizbang est fourni avec des fichiers de configuration pour Emacs. Ceux-ci se terminent habituellement par .el, raccourci pour "Emacs Lisp". Par exemple, si whizbang est un logiciel porté FreeBSD, nous pouvons localiser ces fichiers en faisant

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

et les installer en les copiant dans le répertoire Lisp d'Emacs. Sur FreeBSD 2.1.0-RELEASE, il s'agit de /usr/local/shared/emacs/site-lisp.

Aisni par exemple, si la sortie de la commande `find` était

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

nous ferions

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

Ensuite, nous devons décider quel extension les fichiers source whizbang ont. Disons qu'il s'agit de fichiers se terminant par .wiz. Nous devons ajouter une entrée dans notre fichier .emacs pour être sûr qu'Emacs sera capable d'utiliser les informations dans whizbang.el.

Trouvez l'entrée auto-mode-alist dans .emacs et ajoutez une ligne pour whizbang, comme :

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

Cela signifie qu'Emacs ira automatiquement dans la fonction `whizbang-mode` quand vous éditerez un fichier se terminant par .wiz.

Juste en-dessous, vous trouverez l'entrée font-lock-auto-mode-list. Ajoutez `whizbang-mode` à celle-ci 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-mode)
```

```
  "List of modes to always start in font-lock-mode")
```

Cela signifie qu'Emacs autorisera toujours `font-lock-mode` (ie colorisation de la syntaxe) pendant l'édition d'un fichier .wiz.

Et c'est tout ce qui est nécessaire. S'il y a quoique ce soit que vous voulez de fait automatiquement quand vous ouvrez un fichier .wiz, vous pouvez ajouter un `whizbang-mode hook` (voir `my-scheme-mode-hook` pour un exemple simple qui ajoute `auto-indent`, l'auto-indentation).

5.8. Pour aller plus loin

- Brian Harvey and 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 and Berthold Klaus Paul Horn *Lisp (3rd Edition)* Addison-Wesley 1989 ISBN 0-201-08319-1
- Brian W. Kernighan and Rob Pike *The Unix Programming Environment* Prentice-Hall 1984 ISBN 0-13-937681-X
- Brian W. Kernighan and 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

Chapitre 6. Programmation sécurisée

Ce chapître a été écrit par Murray Stokely.

6.1. Synopsis

Ce chapître décrit quelques problèmes de sécurité qui ont tourmenté les programmeurs Unix depuis des dizaines d'années et quelques uns des nouveaux outils disponibles pour aider les programmeurs à éviter l'écriture de code non sécurisé.

6.2. Méthodologie de développement sécurisé

Ecrire des applications sécurisées demande une très minutieuse et pessimiste vision de la vie. Les applications devrait fonctionner avec le principe du "privilège moindre" de façon à ce qu'aucun processus ne fonctionne avec plus que le strict minimum dont il a besoin pour accomplir sa tâche. Le code pré-testé devrait être réutilisé autant que possible pour éviter les erreurs communes que d'autres peuvent déjà avoir réparées.

Un des pièges de l'environnement Unix est qu'il est facile d'affecter la stabilité de l'environnement. Les applications ne devraient jamais avoir confiance dans la saisie de l'utilisateur (sous toutes ses formes), les ressources système, la communication inter-processus, ou l'enchaînement des évènements. Les processus Unix n'exécutent pas de manière synchrone aussi, logiquement, les opérations sont rarement atomiques.

6.3. Dépassement de capacité

Les dépassements de capacité ("Buffer Overflows") existent depuis les débuts de l'architecture de Von-Neuman 1. Ils gagnèrent une grande notoriété en 1988 avec le ver pour Internet de Morris. Malheureusement, la même attaque basique reste effective aujourd'hui. Des 17 rapports de sécurité du CERT de 1999, 10 furent causés directement des bogues logiciels de dépassement de capacité. De loin la plus commune de types d'attaques par dépassement de capacité est basée sur la corruption de la pile.

La plupart des systèmes informatiques modernes utilise une pile pour passer les arguments aux procédures et stocker les variables locales Une pile est une zone mémoire dernier entré-premier sorti (Last In-First Out : LIFO) dans la zone de mémoire haute de l'image d'un processus. Quand un programme invoque une fonction une nouvelle structure pile est créée. Cette structure pile consiste dans les arguments passés à la fonction aussi bien que dans une quantité dynamique d'espace pour la variable locale. Le pointeur de pile est un registre qui référence la position courante du sommet de la pile. Etant donné que cette valeur change constamment au fur et à mesure que de nouvelles valeurs sont ajoutées au sommet de la pile, beaucoup d'implémentations fournissent aussi un pointeur de structure qui est positionné dans le voisinage du début de la structure pile de façon à ce que les variables locales soient plus facilement adressables relativement à cette valeur. 1 L'adresse de retour des appels de fonction est aussi stocké dans la pile, et cela est la cause des découvertes des dépassements de pile puisque faire déborder une variable locale dans une fonction peut écraser l'adresse de retour de cette fonction, permettant potentiellement à un utilisateur malveillant d'exécuter le code qu'il ou elle désire.

Bien que les attaques basées sur les dépassements de pile soient de loin les plus communes, il serait aussi possible de faire exploser la pile avec une attaque du tas (malloc/free).

Le langage de programmation C ne réalise pas de vérifications automatiques des limites sur les tableaux ou pointeurs comme d'autres langages le font. De plus, la librairie standard du C est remplie d'une poignée de fonctions très dangereuses.

`strcpy`(char *dest, const char *src)	Peut faire déborder le tampon dest
`strcat`(char *dest, const char *src)	Peut faire déborder le tampon dest
`getwd`(char *buf)	Peut faire déborder le tampon buf
`gets`(char *s)	Peut faire déborder le tampon s
`[vf]scanf`(const char *format, ...)	Peut faire déborder ses arguments.
`realpath`(char *path, char resolved_path[])	Peut faire déborder le tampon path
`[v]sprintf`(char *str, const char *format, ...)	Peut faire déborder le tampon str.

6.3.1. Exemple de dépassement de capacité

L'exemple de code suivant contient un dépassement de capacité conçu pour écraser l'adresse de retour et "sauter" l'instruction suivant immédiatement l'appel de la fonction. (Inspiré par 4)

```
#include stdio.h

void manipulate(char *buffer) {
  char newbuffer[80];
  strcpy(newbuffer,buffer);
}

int main() {
  char ch,buffer[4096];
  int i=0;

  while ((buffer[i++] = getchar()) != '\n') {};

  i=1;
  manipulate(buffer);
  i=2;
  printf("La valeur de i est : %d\n",i);
  return 0;
}
```

Examinons quel serait l'aspect de l'image mémoire de ce processus si nous avions entré 160 espaces dans notre petit programme avant d'appuyer sur `Entrée`.

Evidemment une entrée plus malveillante pourrait être imaginée pour exécuter des instructions déjà compilées (comme exec(/bin/sh)).

6.3.2. Eviter les dépassements de capacité

La solution la plus simple au problème de débordement de pile est de toujours utiliser de la mémoire restreinte en taille et les fonctions de copie de chaîne de caractères. `strncpy` et `strncat` font parties de la libraire standard du C. Ces fonctions acceptent une valeur de longueur comme paramètre qui qui ne devrait pas être plus grande que la taille du tampon de destination. Ces fonctions vont ensuite copier `taille` octets de la source vers la destination. Toutefois, il y a un certain nombre de problèmes avec ces fonctions. Aucune fonction ne garantit une terminaison par le caractère NULL si la taille du tampon d'entrée est aussi grand que celui de destination. Le paramètre taille est aussi utilisé de façon illogique entre `strncpy` et `strncat` aussi il est facile pour les programmeurs d'être déroutés sur leur utilisation convenable. Il y a aussi une perte significative des performances comparé à `strcpy` lorsque l'on copie une courte chaîne dans un grand tampon puisque `strncpy` remplit de caractères NULL jusqu'à la taille spécifiée.

Dans OpenBSD, une autre implémentation de la copie de mémoire a été créée pour contourner ces problèmes. Les fonctions `strlcpy` et `strlcat` garantissent qu'elles termineront toujours le tampon de destination par un caractère NULL losque l'argument de taille est différent de zéro. Pour plus d'informations sur ces fonctions, voir 6. Les fonctions `strlcpy` et `strlcat` d'OpenBSD ont été inclues dans FreeBSD depuis la version 3.5.

6.3.2.1. V#233;rifications des limites en fonctionnement basées sur le compilateur

Malheureusement il y a toujours un très important assortiment de code en utilisation publique qui copie aveuglément la mémoire sans utiliser une des routines de copie limitée dont nous venons juste de discuter. Heureusement, il y a une autre solution. Plusieurs produits complémentaires pour compilateur et librairies existent pour faire de la vérification de limites pendant le fonctionnement en C/C++.

StackGuard est un de ces produits qui est implémenté comme un petit correctif ("patch") pour le générateur de code gcc. Extrait du site Internet de StackGuard, http://immunix.org/stackguard.html :

"StackGuard détecte et fait échouer les attaques par débordement de pile en empêchant l'adresse de retour sur la pile d'être altérée. StackGuard place un mot "canari" à côté de l'adresse de retour quand la fontion est appelée. Si le mot "canari" a été altéré au retour de la fonction, alors une attaque par débordement de pile a été tentée et le programme répond en envoyant une alerte d'intrusion dans la trace système (syslog) et s'arrête alors."

"StackGuard est implémenté comme un petit correctif au générateur de code gcc, spécifiquement sur les routines function_prolog() et function_epilog(). function_prolog() a été amélioré pour laisser des "canaris" sur la pile quand les fonctions démarrent, et function_epilog vérifie l'intégrité des "canaris" quand la fonction se termine. Tout essai pour corrompre l'adresse de retour est alors détectée avant que la fonction ne retourne."

Recompiler votre application avec StackGuard est un moyen efficace pour stopper la plupart des attques par dépassement de capacité, mais cela peut toujours être compromis.

6.3.2.2. Vérifications des limites en fonctionnement basées sur les librairies

Les mécanismes basés sur le compilateur sont totalement inutiles pour logiciel seulement binaire que vous ne pouvez recompiler. Pour ces situations, il existe un nombre de librairies qui re-implémente les fonctions peu sûres de la librairie C (`strcpy`, `fscanf`, `getwd`, etc..) et assurent que ces fonctions ne peuvent pas écrire plus loin que le pointeur de pile.

- libsafe
- libverify
- libparnoia

Malheureusement ces défenses basées sur les librairies possèdent un certain nombre de défauts. Ces librairies protègent seulement d'un très petit ensemble de problèmes liés à la sécurité et oublient de réparer le problème actuel. Ces défenses peuvent échouer si l'application a été compilée avec -fomit-frame-pointer. De même, les variables d'environnement LD_PRELOAD et LD_LIBRARY_PATH peuvent être réécrites/non définies par l'utilisateur.

6.4. Les problèmes liés à SetUID

Il y a au moins 6 differents ID (identifiants) associés à un processus donné. A cause de cela, vous devez être très attentif avec l'accès que votre processus possède à un instant donné. En particulier, toutes les applications ayant reçu des privilèges par seteuid doivent les abandonnés dès qu'ils ne sont plus nécessaires.

L'identifiant de l'utilisateur réel (real user ID) peut seulement être changé par un processus super-utilisateur. Le programme login met celui à jour quand un utilisateur se connecte et il est rarement changé.

L'identifiant de l'utilisateur effectif (effective user ID) est mis à jour par les fonctions `exec()` si un programme possède son bit seteuid placé. Une application peut appeler `seteuid()` à n'importe quel moment pour règler l'identifiant de l'utilisateur effectif sur l'identifiant d'un utilisateur réel ou sur le "set-user-ID" sauvé. Quand l'identifiant de l'utilisateur effectif est placé par les fonctions `exec()`, la valeur précédente est sauvée dans le "set-user-ID" sauvé.

6.5. Limiter l'environnement de votre programme

La méthode traditionnelle pour restreindre l'accès d'un processus se fait avec l'appel système `chroot()`. Cet appel système change le répertoire racine depuis lequel tous les autres chemins sont référencés pour un processus et ses fils. Pour que cet appel réussisse, le processus doit avoir exécuté (recherché) la permission dans le répertoire référencé. Le nouvel environnement environment ne prend pas effet que lorsque vous appelez `chdir()` dans celui-ci. Il doit être aussi noté qu'un processus peut facilement s'échapper d'un environnement chroot s'il a les privilèges du super-utilisateur. Cela devrait être accompli en créant des fichiers spéciaux de périphérique pour la mémoire du noyau, en attachant un dévermineur à un processus depuis l'extérieur de sa "prison", ou par d'autres manières créatrices.

Le comportement de l'appel système `chroot()` peut être un peu contrôlé avec la commande `sysctl` et la variable kern.chroot_allow_open_directories. Quand cette valeur est règlée à 0, `chroot()`

échouera avec EPERM s'il y a un répertoire d'ouvert. Si la variable est règlée sur la valeur par défaut 1, alors `chroot()` échouera avec EPERM s'il y a un répertoire d'ouvert et que le processus est déjà sujet à un appel `chroot()`. Pour toute autre valeur, la vérification des répertoires ouverts sera totalement court-circuitée.

6.5.1. La fonctionnalité "prison" de FreeBSD

Le concept de Prison ("Jail") étend `chroot()` en limitant les droits du super-utilisateur pour créer un véritable `serveur virtuel`. Une fois qu'une prison est mise en place, toute communication réseau doit avoir lieu au travers de l'adresse IP spécifiée, et le droit du "privilège super-utilisateur" dans cette prison est sévèrement gêné.

Tant qu'il se trouve en prison, tout test avec les droits du super-utilisateur dans le noyau au travers d'un appel à `suser()` échouera. Toutefois, quelques appels à `suser()` ont été changés par la nouvelle interface `suser_xxx()`. Cette fonction est responsable de fournir ou de retirer les accès aux droits du super-utilisateur pour les processus emprisonnés.

Un processus super-utilisateur dans un environnement emprisonné a le pouvoir de :

- Manipuler les identitifications avec `setuid`, `seteuid`, `setgid`, `setegid`, `setgroups`, `setreuid`, `setregid`, `setlogin`
- Règler les limites en ressources avec `setrlimit`
- Modifier quelques variables du noyau par sysctl (kern.hostname)
- `chroot()`
- Règler les paramètres d'un noeud virtuel (vnode): `chflags`, `fchflags`
- Règler les attributs d'un noeud virtuel comme les permissions d'un fichier, le propriétaire, le groupe, la taille, la date d'accès, et la date de modification.
- Se lier à des ports privilégiés sur Internet (ports < 1024)

`Jail` est un outil très utile pour exécuter des applications dans un environnement sécurisé mais il a des imperfections. Actuellement, les mécanismes IPC (Inter-Process Communications) n'ont pas été convertis pour utiliser `suser_xxx` aussi des applications comme MySQL ne peuvent être exécutée dans une prison. L'accès super-utilisateur peut avoir un sens très limité dans une prison, mais il n'y aucune façon de spécifier exactement ce que très limité veut dire.

6.5.2. Les capacités des processus POSIX.1e

Posix a réalisé un document de travail qui ajoute l'audit d'évènement, les listes de contrôle d'accès, les privilèges fins, l'étiquetage d'information, et le contrôle d'accès mandaté.

Il s'agit d'un travail en cours et c'est l'objectif du projet TrustedBSD. Une partie du travail initial a été intégré dans FreeBSD-current (cap_set_proc(3)).

6.6. La confiance

Une application ne devrait jamais supposer que tout est sain dans l'environnement des utilisateurs.

Cela inclut (mais n'est certainement pas limité à) : la saisie de l'utilisateur, les signaux, les variables d'environnement, les ressources, les communication inter-processus, les mmaps, le répertoire de travail du système de fichiers, les descripteurs de fichier, le nombre de fichiers ouverts, etc.

Vous ne devriez jamais supposer que vous pouvez gérer toutes les formes de saisie invalide qu'un utilisateur peut entrer. Votre application devrait plutôt utiliser un filtrage positif pour seulement permettre un sous-ensemble spécifique que vous jugez sain. Une mauvaise validation des entrées a été la cause de beaucoup découvertes de bogues, spécialement avec les scripts CGI sur le web. Pour les noms de fichier, vous devez être tout particulièrement attentif aux chemins ("../", "/"), liens symboliques et caractères d'échappement de l'interpréteur de commandes.

Perl possède une caractéristique tès sympathique appelée mode "Taint" qui peut être utilisée pour empêcher les scripts d'utiliser des données externes au programme par un moyen non sûr. Ce mode vérifiera les arguments de la ligne de commandes, les variables d'environnement, les informations localisées (propres aux pays), les résultats de certains appels système (`readdir()`, `readlink()`, `getpwxxx()`) et toute entrée de fichier.

6.7. Les conditions de course

Une condition de course est un comportement anormal causé par une dépendance inattendue sur le séquencement relatif des évènements. En d'autres mots, un programmeur a supposé à tort qu'un évènement particulier se passerait avant un autre.

Quelques causes habituelles de conditions de course sont les signaux, les vérifications d'accès et les fichiers ouverts. Les signaux sont des évènements asynchrones par nature aussi un soin particulier doit être pris pour les utiliser. Vérifier les accès avec `access(2)` puis `open(2)` n'est clairement pas atomique. Les utilisateurs peuvent déplacer des fichiers entre les deux appels. Les applications privilégiées devraient plutôt faire un appel à `seteuid()` puis appeler `open()` directement. Dans le même esprit, une application devrait toujours règler un umask correct avant un appel à `open()` pour prévenir le besoin d'appels non valides à `chmod()`.

Partie III: Le noyau

Chapitre 7. Histoire du noyau Unix

Un peu d’histoire sur le noyau Unix/BSD, les appels système, comment fonctionnent les processus, bloquer, planifier, les threads (noyau), le basculement de contexte, les signaux, les interruptions, les modules, etc.

Chapitre 8. Notes sur le verrouillage

Ce chapître est maintenu par The FreeBSD SMP Next Generation Project. Envoyez leur directement les commentaires et les suggestions à link:{freebsd-smp}.

Ce document souligne le verrouillage utilisé dans le noyau FreeBSD pour permettre d'utiliser du vrai multi-processeur à l'intérieur du noyau. Le verrouillage peut être réalisé par différents moyens. Les structures de données puvent être protégées par des mutex ou lockmgr(9) verrous. Quelques variables sont protégées simplement par l'utilisation continuelle d'opérations atomiques pour y accéder.

8.1. Les mutex

Un mutex est simplement un verrou utilisé pour garantir exclusion mutuelle. Spécifiquement, un mutex ne peut appartenir qu'à une entité à la fois. Si une autre entité désire obtenir un mutex déjà pris , elle doit attendre jusqu'à ce que le mutex soit relaché. Dans le noyau FreeBSD, les mutex appartiennent aux processus.

Les mutex peuvent être acquis récursivement, mais ils sont conçus pour n'être pris que pendant une courte période. Spécifiquement, le détenteur ne doit pas se suspendre pendant qu'il retient un mutex. Si vous avez besoin de maintenir un verrouillage pendant une suspension, utilisez un lockmgr(9) verrou ("lock").

Chaque mutex a plusieurs intérêts :

Nom de la variable

Nom de la variable struct mtx dans le code source du noyau.

Nom logique

Le nom du mutex lui est assigné par `mtx_init`. Ce nom est affiché dans les messages de trace KTR, témoigne des erreurs et avertissements et est utilisé pour distinguer les mutex dans les traces.

Type

Le type du mutex en termes de constantes nommées MTX_*. La signification de chaque constante nommée est documentée dans mutex(9).

MTX_DEF

Un mutex endormi

MTX_SPIN

Un mutex tournant

MTX_COLD

Ce mutex est initialisé très tard. Toutefois, il doit être déclaré via `MUTEX_DECLARE`, et la constante nommée MTX_COLD doit être passée à `mtx_init`.

MTX_TOPHALF

Ce mutex tournant ne désactive pas les interruptions.

MTX_NORECURSE

Ce mutex n’a pas la permission d’être recursif.

Protégés

Une liste de structures de données ou des membres de structure de données que cette entrée protège. Pour les membres de structures de données, le nom sera de la forme `structure name .member name`.

Fonctions dépendantes

Les fonctions qui peuvent seulement être appelées si ce mutex est pris.

Tableau 1. Liste du mutex

Nom de la variable	Nom logique	Type	Protégés	Fonctions dépendantes
sched_lock	"sched lock"	MTX_SPIN | MTX_COLD	_gmonparam, cnt.v_swtch, cp_time, curpriority, mtx .mtx_blocked, mtx .mtx_contested, proc.p_contested, proc.p_blocked, proc.p_flag (P_PROFIL XXX, P_INMEM, P_SINTR, P_TIMEOUT, P_SWAPINREQ XXX, P_INMEN XXX), proc.p_nice, proc.p_procq, proc .p_blocked, proc .p_estcpu, proc .p_nativepri, proc .p_priority, proc .p_usrpri, proc .p_rtprio, proc .p_rqindex, proc .p_stats→p_prof, proc.p_stats→p_ru, proc.p_stat, proc .p_cpticks, proc .p_iticks, proc .p_uticks, proc .p_sticks, proc .p_swtime, proc .p_slptime, proc .p_runtime, proc .p_pctcpu, proc .p_oncpu, proc .p_asleep, proc .p_wchan, proc .p_wmesg, proc .p_slpq, proc .p_vmspace (XXX - in statclock), pscnt, slpque, itqueuebits, itqueues, rtqueuebits, rtqueues, queuebits, queues,	setrunqueue, remrunqueue, mi_switch, chooseproc, schedclock, resetpriority, updatepri, maybe_resched, cpu_switch, cpu_throw

Nom de la variable	Nom logique	Type	Protégés	Fonctions dépendantes
vm86pcb_lock	"vm86pcb lock"	MTX_DEF | MTX_COLD	vm86pcb	vm86_bioscall
Giant	"Giant"	MTX_DEF | MTX_COLD	nearly everything	lots
callout_lock	"callout lock"	MTX_SPIN	callfree, callwheel, nextsoftcheck, proc.p_itcallout, proc.p_slpcallout, softticks, ticks	

8.2. Les verrous du gestionnaire de verrous (Lock Manager)

Les verrous qui sont fournis par l'interface lockmgr(9) sont les verrous du gestionnaire de verrous. Ces verrous sont des verrous lecture-écriture et peuvent être retenus par un process suspendu.

Tableau 2. lockmgr(9) List de verrou

Nom de la variable	Protégés		
allproc_lock	allproc zombproc pidhashtbl proc.p_list proc.p_hash nextpid	proctree_lock	proc.p_children proc.p_sibling

8.3. Variables protégées atomiquement

Une variable protégée atomiquement est une variable spéciale qui n'est pas protégé par un verrou explicite. Toutefois, tous les accès de données aux variables utilisent des opérations atomiques spéciales comme décrit dans atomic(9). Très peu de variables sont traitées de cette façon, bien que les autres primitives de synchronisation comme les mutex soient implémentées avec des variables protégées atomiquement.

- astpending
- mtx.mtx_lock

Partie IV: Mémoire et mémoire virtuelle

Chapitre 9. La mémoire virtuelle

MV, gestion par page, gestion sur disque, allouer de la mémoire, tester les fuites de mémoires, mmap, vnodes, etc.

Partie V: Système E/S (Entrées/Sorties)

Chapitre 10. UFS

UFS, FFS, Ext2FS, JFS, inodes, mémoire tampon, mettre à jour les données d’un disque, verrouillage, metadata, soft-updates, LFS, portalfs, procfs, vnodes, partage de mémoire, objets en mémoire, TLBs, mettre en cache

Partie VI: Communication InterProcessus (IPC)

Chapitre 11. Les signaux

Signaux, tubes, sémaphores, files de message, segments de mémoire partagée, ports, prises, portes

Partie VII: Le réseau

Chapitre 12. Les prises

Prises, bpf, IP, TCP, UDP, ICMP, OSI, ponts, pare-feux, translation d’adresses (NAT), séparation de réseaux, etc

Partie VIII: Systèmes de fichiers en réseau

Chapitre 13. AFS

AFS, NFS, SANs etc]

Partie IX: Gestion du terminal

Chapitre 14. Syscons

Syscons, tty, PCVT, console en liaison série, économiseurs d’écran, etc

Partie X: Le son

Chapitre 15. OSS

OSS, formes d’ondes, etc

Partie XI: Pilotes de périphérique

Chapitre 16. Ecrire des pilotes de périphériques pour FreeBSD

Ce chapitre a été écrit par Murray Stokely <murray@FreeBSD.org> avec des sélections depuis une variété de codes source inclus dans la page de manuel d'intro(4) de Joerg Wunsch.

16.1. Introduction

Ce chapitre fournit une brève introduction sur l'écriture de pilotes de périphériques pour FreeBSD. Un périphérique, dans ce contexte, est un terme utilisé le plus souvent pour tout ce qui est lié au matériel et qui dépend du système, comme les disques, imprimantes, ou un écran avec son clavier. Un pilote de périphérique est un composant logiciel du système d'exploitation qui contrôle un périphérique spécifique. Il y a aussi ce que l'on appelle les pseudo-périphériques ("pseudo-devices") où un pilote de périphérique émule le comportement d'un périphérique dans un logiciel sans matériel particulier sous-jacent. Les pilotes de périphériques peuvent être compilés dans le système statiquement ou chargé à la demande via l'éditeur de liens dynamique du noyau "kld".

La plupart des périphériques dans un système d'exploitation de type Unix sont accessibles au travers de fichiers spéciaux de périphérique (device-nodes), appelés parfois fichiers spéciaux. Ces fichiers sont habituellement stockés dans le répertoire /dev de la hiérarchie du système de fichiers. Jusqu'à ce que devfs soit totalement intégré dans FreeBSD, chaque fichier spécial de périphérique doit être créé statiquement et indépendamment de l'existence du pilote de périphérique associé. La plupart des fichiers spéciaux de périphérique du système sont créés en exécutant `MAKEDEV`.

Les pilotes de périphérique peuvent être en gros séparés en deux catégories; les pilotes de périphérique en mode caractère et les pilotes de périphériques réseau.

16.2. L'éditeur de liens dynamiques du noyau - KLD

L'interface kld permet aux administrateurs système d'ajouter et d'enlever dynamiquement une fonctionnalité à un système en marche. Cela permet aux développeurs de pilote de périphérique de charger leurs nouveaux changements dans le noyau en fonctionnement sans redémarrer constamment pour tester ces derniers.

L'interface kld est utilisé au travers des commandes d'administrateur suivantes :

- `kldload` - charge un nouveau module dans le noyau
- `kldunload` - décharge un module du noyau
- `kldstat` - liste les modules chargés dans le noyau

Structure squelettique d'un module de noyau

```
/*
 * Squelette KLD
 * Inspiré de l'article d'Andrew Reiter paru sur Daemonnews
 */
```

```
#include <sys/types.h>
#include <sys/module.h>
#include <sys/systm.h>  /* uprintf */
#include <sys/errno.h>
#include <sys/param.h>  /* defines utilise dans kernel.h */
#include <sys/kernel.h> /* types utilise dans le module d'initialisation */

/*
 * charge le gestionnaire quit traite du chargement et déchargement d'un KLD.
 */

static int
skel_loader(struct module *m, int what, void *arg)
{
  int err = 0;

  switch (what) {
  case MOD_LOAD:                /* kldload */

    uprintf("Skeleton KLD charge.\n");
    break;
  case MOD_UNLOAD:
    uprintf("Skeleton KLD decharge.\n");
    break;
  default:
    err = EINVAL;
    break;
  }
  return(err);
}

/* Declare ce module au reste du noyau */

DECLARE_MODULE(skeleton, skel_loader, SI_SUB_KLD, SI_ORDER_ANY);
```

16.2.1. Makefile

FreeBSD fournit un fichier d'inclusion "makefile" que vous pouvez utiliser pour compiler rapidement votre ajout au noyau.

```
SRCS=skeleton.c
KMOD=skeleton

.include <bsd.kmod.mk>
```

Lancer simplement la commande `make` avec ce fichier Makefile créera un fichier skeleton.ko qui peut être chargé dans votre système en tapant :

```
# kldload -v ./skeleton.ko
```

16.3. Accéder au pilote d'un périphérique

Unix fournit un ensemble d'appels système communs utilisable par les applications de l'utilisateur. Les couches supérieures du noyau renvoient ces appels au pilote de périphérique correspondant quand un utilisateur accède au fichier spécial de périphérique. Le script `/dev/MAKEDEV` crée la plupart des fichiers spéciaux de périphérique pour votre système mais si vous faites votre propre développement de pilote, il peut être nécessaire de créer vos propres fichiers spéciaux de périphérique avec la commande `mknod`

16.3.1. Créer des fichiers spéciaux de périphériques statiques

La commande `mknod` nécessite quatre arguments pou créer un fichier spécial de périphérique. Vous devez spécifier le nom de ce fichier spécial de périphérique, le type de périphérique, le numéro majeur et le numéro mineur du périphérique.

16.3.2. Les fichiers spéciaux de périphérique dynamiques

Le périphérique système de fichiers, ou devfs, fournit l'accès à l'espace des noms des périphériques du noyau dans l'espace du système de fichiers global. Ceci élimine les problèmes de pilote sans fichier spécial statique, ou de fichier spécial sans pilote installé. Devfs est toujours un travail en cours mais il fonctionne déjà assez bien.

16.4. Les périphériques caractères

Un pilote de périphérique caractère est un pilote qui transfère les données directement au processus utilisateur ou vers celui-ci. Il s'agit du plus commun des types de pilote de périphérique et il y en a plein d'exemples simples dans l'arbre des sources.

Cet exemple simple de pseudo-périphérique enregistre toutes les valeurs que vous lui avez écrites et peut vous les renvoyer quand vous les lui demandez.

```
/*
 * un simple pseudo-périphérique `echo' KLD
 *
 * Murray Stokely
 */

#define MIN(a,b) (((a) < (b)) ? (a) : (b))

#include <sys/types.h>
#include <sys/module.h>
#include <sys/systm.h> /* uprintf */
#include <sys/errno.h>
#include <sys/param.h>  /* defines utilises dans kernel.h */
#include <sys/kernel.h> /* types utilises dans me module d'initialisation */
```

```
#include <sys/conf.h>   /* cdevsw struct */
#include <sys/uio.h>    /* uio struct */
#include <sys/malloc.h>

#define BUFFERSIZE 256

/* Prototypes des fonctions */
d_open_t      echo_open;
d_close_t     echo_close;
d_read_t      echo_read;
d_write_t     echo_write;

/* Points d'entrée du périphérique Caractère */
static struct cdevsw echo_cdevsw = {
  echo_open,
  echo_close,
  echo_read,
  echo_write,
  noioctl,
  nopoll,
  nommap,
  nostrategy,
  "echo",
  33,                   /* reserve pour lkms - /usr/src/sys/conf/majors */
  nodump,
  nopsize,
  D_TTY,
  -1
};

typedef struct s_echo {
  char msg[BUFFERSIZE];
  int len;
} t_echo;

/* variables */
static dev_t sdev;
static int len;
static int count;
static t_echo *echomsg;

MALLOC_DECLARE(M_ECHOBUF);
MALLOC_DEFINE(M_ECHOBUF, "echobuffer", "cache pour le module echo");

/*
 * Cette fonction est appelee par les appels systeme kld[un]load(2) pour
 * determiner quelles actions doivent etre faites quand le
 * module est charge ou decharge
 */

static int
```

```
echo_loader(struct module *m, int what, void *arg)
{
  int err = 0;

  switch (what) {
  case MOD_LOAD:                /* kldload */
    sdev = make_dev(&echo_cdevsw,
            0,
            UID_ROOT,
            GID_WHEEL,
            0600,
            "echo");
    /* aloocation de mémoire noyau pour l'utilisation de ce module */
    /*    malloc(256,M_ECHOBUF,M_WAITOK); */
    MALLOC(echomsg, t_echo *, sizeof(t_echo), M_ECHOBUF, M_WAITOK);
    printf("Peripherique Echo charge.\n");
    break;
  case MOD_UNLOAD:
    destroy_dev(sdev);
    FREE(echomsg,M_ECHOBUF);
    printf("Peripherique Echo decharge.\n");
    break;
  default:
    err = EINVAL;
    break;
  }
  return(err);
}

int
echo_open(dev_t dev, int oflags, int devtype, struct proc *p)
{
  int err = 0;

  uprintf("Peripherique \"echo\" ouvert avec succes.\n");
  return(err);
}

int
echo_close(dev_t dev, int fflag, int devtype, struct proc *p)
{
  uprintf("Fermeture du peripherique \"echo.\"\n");
  return(0);
}

/*
 * La fonction read prend juste comme parametre
 * le cache qui a ete sauve par l'appel à echo_write()
 * et le retourne a l'utilisateur pour acces.
 * uio(9)
 */
```

```
int
echo_read(dev_t dev, struct uio *uio, int ioflag)
{
  int err = 0;
  int amt;

  /* De quelle taille est cette operation read ?  Aussi grande que l'utilisateur le
veut,
     ou aussi grande que les donnees restantes */
  amt = MIN(uio->uio_resid, (echomsg->len - uio->uio_offset > 0) ? echomsg->len - uio-
>uio_offset : 0);
  if ((err = uiomove(echomsg->msg + uio->uio_offset,amt,uio)) != 0) {
    uprintf("uiomove echoue!\n");
  }

  return err;
}

/*
 * echo_write prend un caractere en entree et le sauve
 * dans le cache pour une utilisation ulterieure.
 */

int
echo_write(dev_t dev, struct uio *uio, int ioflag)
{
  int err = 0;

  /* Copie la chaine d'entree de la memoire de l'utilisateur a la memoire du noyau*/
  err = copyin(uio->uio_iov->iov_base, echomsg->msg, MIN(uio->uio_iov-
>iov_len,BUFFERSIZE));

  /* Maintenant nous avons besoin de terminer la chaine par NULL */
  *(echomsg->msg + MIN(uio->uio_iov->iov_len,BUFFERSIZE)) = 0;
  /* Enregistre la taille */
  echomsg->len = MIN(uio->uio_iov->iov_len,BUFFERSIZE);

  if (err != 0) {
    uprintf("Ecriture echouee: mauvaise adresse!\n");
  }

  count++;
  return(err);
}

DEV_MODULE(echo,echo_loader,NULL);
```

Pour installer ce pilote, vous devrez d’abord créer un fichier spécial dans votre système de fichiers avec une commande comme :

```
# mknod /dev/echo c 33 0
```

Avec ce pilote chargé, vous devriez maintenant être capable de taper quelque chose comme :

```
# echo -n "Test Donnees" > /dev/echo
# cat /dev/echo
Test Donnees
```

Périphériques réels dans le chapitre suivant.

Informations additionnelles

- Dynamic Kernel Linker (KLD) Facility Programming Tutorial - Daemonnews October 2000
- How to Write Kernel Drivers with NEWBUS - Daemonnews July 2000

16.5. Pilotes Réseau

Les pilotes pour périphérique réseau n'utilisent pas les fichiers spéciaux pour pouvoir être accessibles. Leur sélection est basée sur d'autres décisions faites à l'intérieur du noyau et plutôt que d'appeler open(), l'utilisation d'un périphérique réseau se fait généralement en se servant de l'appel système socket(2).

man ifnet(), périphérique "en boucle", drivers de Bill Paul, etc..

Chapitre 17. Les périphériques PCI

Ce chapître traitera des mécanismes de FreeBSD pour écrire un pilote de périphérique pour un périphérique sur bus PCI.

17.1. Rechercher et rattacher

Informations ici sur comment le code du bus PCI fait un cycle sur les périphériques non rattachés et voir si le nouvellement chargé pilote de périphérique chargeable dans le noyau (kld) sera rattaché à l'un d'eux.

```
/*
 * Simple KLD pour jouer avec les fonctions PCI.
 *
 * Murray Stokely
 */

#define MIN(a,b) (((a) < (b)) ? (a) : (b))

#include <sys/types.h>
#include <sys/module.h>
#include <sys/systm.h>  /* uprintf */
#include <sys/errno.h>
#include <sys/param.h>  /* defines used in kernel.h */
#include <sys/kernel.h> /* types used in module initialization */
#include <sys/conf.h>   /* cdevsw struct */
#include <sys/uio.h>    /* uio struct */
#include <sys/malloc.h>
#include <sys/bus.h>  /* structs, prototypes for pci bus stuff */

#include <pci/pcivar.h> /* For get_pci macros! */

/* Prototypes des fonctions */
d_open_t      mypci_open;
d_close_t     mypci_close;
d_read_t      mypci_read;
d_write_t     mypci_write;

/* Points d'entrée du pilote de périphérique caractère */

static struct cdevsw mypci_cdevsw = {
  mypci_open,
  mypci_close,
  mypci_read,
  mypci_write,
  noioctl,
  nopoll,
  nommap,
  nostrategy,
```

```
  "mypci",
  36,                   /* reserved for lkms - /usr/src/sys/conf/majors */
  nodump,
  nopsize,
  D_TTY,
  -1
};

/* variables */
static dev_t sdev;

/* Nous sommes plus interresses dans la recherche/attachement
que dans l'ouverture/fermeture/lecture/ecriture a ce point */

int
mypci_open(dev_t dev, int oflags, int devtype, struct proc *p)
{
  int err = 0;

  uprintf("Peripherique \"monpci\" ouvert avec succes.\n");
  return(err);
}

int
mypci_close(dev_t dev, int fflag, int devtype, struct proc *p)
{
  int err=0;

  uprintf("Peripherique \"monpci.\ "ferme\n");
  return(err);
}

int
mypci_read(dev_t dev, struct uio *uio, int ioflag)
{
  int err = 0;

  uprintf("lecture dans monpci!\n");
  return err;
}

int
mypci_write(dev_t dev, struct uio *uio, int ioflag)
{
  int err = 0;

  uprintf("Ecriture dans monpci!\n");
  return(err);
}

/* PCI Support Functions */
```

```
/*
 * Retourne la chaine d'identification si ce peripherique est le notre
 */
static int
mypci_probe(device_t dev)
{
  uprintf("MonPCI Probe\n"
      "ID Fabricant: 0x%x\n"
      "ID Peripherique : 0x%x\n",pci_get_vendor(dev),pci_get_device(dev));

  if (pci_get_vendor(dev) == 0x11c1) {
    uprintf("Nous avons le WinModem, recherche reussi!\n");
    return 0;
  }

  return ENXIO;
}

/* La fonction Attach n'est appelée que si
la recherche est reussie*/

static int
mypci_attach(device_t dev)
{
  uprintf("Rattachement de MonPCI pour: ID Peripherique: 0x%x\n",pci_get_vendor(dev));
  sdev = make_dev(&mypci_cdevsw,
          0,
          UID_ROOT,
          GID_WHEEL,
          0600,
          "monpci");
  uprintf("Peripherique Monpci charge.\n");
  return ENXIO;
}

/* Detach le peripherique. */

static int
mypci_detach(device_t dev)
{
  uprintf("Monpci detache!\n");
  return 0;
}

/* Appele lors de l'arret du systeme apres sync. */

static int
mypci_shutdown(device_t dev)
{
  uprintf("Monpci arrete!\n");
```

```
  return 0;
}

/*
 * routine de suspension du peripherique
 */
static int
mypci_suspend(device_t dev)
{
  uprintf("Monpci suspendu!\n");
  return 0;
}

/*
 * routine de reprise du peripherique
 */

static int
mypci_resume(device_t dev)
{
  uprintf("Monpci resume!\n");
  return 0;
}

static device_method_t mypci_methods[] = {
    /* Interface Peripherique*/
    DEVMETHOD(device_probe,     mypci_probe),
    DEVMETHOD(device_attach,    mypci_attach),
    DEVMETHOD(device_detach,    mypci_detach),
    DEVMETHOD(device_shutdown,  mypci_shutdown),
    DEVMETHOD(device_suspend,   mypci_suspend),
    DEVMETHOD(device_resume,    mypci_resume),

    { 0, 0 }
};

static driver_t mypci_driver = {
    "monpci",
    mypci_methods,
    0,
    /*  sizeof(struct mypci_softc), */
};

static devclass_t mypci_devclass;

DRIVER_MODULE(mypci, pci, mypci_driver, mypci_devclass, 0, 0);
```

Informations complémentaires

- PCI Special Interest Group

- PCI System Architecture, Fourth Edition by Tom Shanley, et al.

17.2. Les ressources du bus

FreeBSD fournit un mécanisme orienté objet pour demander des ressources du bus parent. Pratiquement tous les périphériques seront un fils membre d'un type de bus (PCI, ISA, USB, SCSI, etc) et ces périphériques nécessite des ressources issues de leur bus parent (comme des segments de mémoire, des interruptions or des canaux DMA).

17.2.1. Registres d'adresse de base

Pour faire de particulièrement utile avec un périphérique PCI, vous aurez besoin d'obtenir les *registres d'adresse de base* (Base Address Registers ou BARs) de l'espace de configuration PCI. Les détails spécifiques au PCI sur l'obtention du registre d'adresse de base sont masqués dans la fonction `bus_alloc_resource()`.

Par exemple, un pilote typique aura sa fonction `attach()` similaire à ceci :

```
    sc->bar0id = 0x10;
    sc->bar0res = bus_alloc_resource(dev, SYS_RES_MEMORY, &(sc->bar0id),
                  0, ~0, 1, RF_ACTIVE);
    if (sc->bar0res == NULL) {
        uprintf("Allocation memoire du registre PCI de base 0 echouee!\n");
        error = ENXIO;
        goto fail1;
    }

    sc->bar1id = 0x14;
    sc->bar1res = bus_alloc_resource(dev, SYS_RES_MEMORY, &(sc->bar1id),
                  0, ~0, 1, RF_ACTIVE);
    if (sc->bar1res == NULL) {
        uprintf("Allocation memoire du registre PCI de base 1 echouee!\n");
        error =  ENXIO;
        goto fail2;
    }
    sc->bar0_bt = rman_get_bustag(sc->bar0res);
    sc->bar0_bh = rman_get_bushandle(sc->bar0res);
    sc->bar1_bt = rman_get_bustag(sc->bar1res);
    sc->bar1_bh = rman_get_bushandle(sc->bar1res);
```

Des références pour chaque registre d'adresse de base sont gardées dans la structure `softc` afin qu'elle puisse être utilisée pour écrire dans le périphérique plus tard.

Ces références peuvent alors être utilisées pour lire ou écrire dans les registres du périphérique avec les fonctions `bus_space_*`. Par exemple, un pilote peut contenir une fonction raccourci pour lire dans un registre spécifique à une carte comme cela :

```
uint16_t
```

```
board_read(struct ni_softc *sc, uint16_t address) {
    return bus_space_read_2(sc->bar1_bt, sc->bar1_bh, address);
}
```

De façon similaire, une autre peut écrire dans les registres avec :

```
void
board_write(struct ni_softc *sc, uint16_t address, uint16_t value) {
    bus_space_write_2(sc->bar1_bt, sc->bar1_bh, address, value);
}
```

Ces fonctions existent en versions 8bit, 16bit et 32bit et vous devriez utiliser `bus_space_{read|write}_{1|2|4}` en conséquence.

17.2.2. Les interruptions

Les interruptions sont alloués à partir du code orienté objet du bus de façon similaire aux ressources mémoire. D'abord une ressource IRQ doit être allouée à partir du bus parent, et alors le gestionnaire d'interruption doit être règlé pour traiter cet IRQ.

A nouveau, un exemple de fonction `attach()` en dit plusqu'un long discours.

```
/* Recupere la ressource IRQ */

    sc->irqid = 0x0;
    sc->irqres = bus_alloc_resource(dev, SYS_RES_IRQ, &(sc->irqid),
                  0, ~0, 1, RF_SHAREABLE | RF_ACTIVE);
    if (sc->irqres == NULL) {
    uprintf("Allocation IRQ echouee!\n");
    error = ENXIO;
    goto fail3;
    }

    /* Maitnenant nous choisissons notre gestionnaire d'interruption */

    error = bus_setup_intr(dev, sc->irqres, INTR_TYPE_MISC,
               my_handler, sc, &(sc->handler));
    if (error) {
    printf("Ne peut regler l'IRQ\n");
    goto fail4;
    }

    sc->irq_bt = rman_get_bustag(sc->irqres);
    sc->irq_bh = rman_get_bushandle(sc->irqres);
```

17.2.3. DMA

Sur les PC, les périphériques qui veulent utiliser la gestion de bus DMA doivent travailler avec des adresses physiques. C'est un problème puisque FreeBSD utilise une mémoire virtuelle et travaille presque exclusivement avec des adresses virtuelles. Heureusement, il y a une fonction `vtophys()` pour nous aider.

```
#include <vm/vm.h>
#include <vm/pmap.h>

#define vtophys(virtual_address) (...)
```

La solution est toutefois un peu différente sur Alpha, et ce que nous voulons réellement est une fonction appelée `vtobus()`.

```
#if defined(__alpha__)
#define vtobus(va)      alpha_XXX_dmamap((vm_offset_t)va)
#else
#define vtobus(va)      vtophys(va)
#endif
```

17.2.4. Désallouer les resources

Il est très important de désallouer toutes les ressources qui furent allouées pendant `attach()`. Unsoin tout particulier doit être pris pour désallouer les bonnes choses même lors d'un échec afin que le système reste utilisable lorsque votre driver meurt.

Chapitre 18. Contrôleurs SCSI Common Access Method (CAM) **

18.1. En cours de traduction

En cours de traduction

Chapitre 19. Périphériques USB *

Ce chapître a été écrit par Nick Hibma <n_hibma@FreeBSD.org>. Les modifications pour le manuel par Murray Stokely <murray@FreeBSD.org>.

19.1. Introduction

Chapître à traduire.

Chapitre 20. NewBus

Ce chapître traitera de l’architecture NewBus de FreeBSD.

Partie XII: Architectures

Chapitre 21. IA-32

Traite des spécificités de l’architecture x86 sous FreeBSD.

Chapitre 22. Alpha

Traite des spécificités de l'architecture Alpha sous FreeBSD.

Explication des erreurs d'alignements, comment les réparer, comment les ignorer.

Exemple de code assembleur pour FreeBSD/alpha.

Chapitre 23. IA-64

Traite des spécificités de l’architecture IA-64 sous FreeBSD.

Partie XIII: Déverminage

Chapitre 24. Truss

diverses descriptions sur les méthodes de déverminage de certains aspects du système utilisant truss, ktrace, gdb, kgdb, etc

Partie XIV: Les couches de compatibilité

Chapitre 25. Linux

Linux, SVR4, etc

Bibligraphie

Bibliographie

[1] Dave A Patterson and John L Hennessy. Copyright® 1998 Morgan Kaufmann Publishers, Inc. 1-55860-428-6. Morgan Kaufmann Publishers, Inc. Computer Organization and Design. The Hardware / Software Interface. 1-2.

[2] W. Richard Stevens. Copyright® 1993 Addison Wesley Longman, Inc. 0-201-56317-7. Addison Wesley Longman, Inc. Advanced Programming in the Unix Environment. 1-2.

[3] Marshall Kirk McKusick and George Neville-Neil. Copyright® 2004 Addison-Wesley. 0-201-70245-2. Addison-Wesley. The Design and Implementation of the FreeBSD Operating System. 1-2.

[4] Aleph One. Phrack 49; "Smashing the Stack for Fun and Profit".

[5] Chrispin Cowan, Calton Pu, and Dave Maier. StackGuard; Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.

[6] Todd Miller and Theo de Raadt. strlcpy and strlcat — consistent, safe string copy and concatenation.