ELE4205
Systèmes d'exploitation et interfaces matérielles

Richard Gourdeau
Département de génie électrique
École Polytechnique de Montréal

21 octobre 2018

Préface

Ce document est une deuxième version des notes de cours d'ELE4205. Ce document est encore en évolution : mises à jour, corrections, ajout d'exercices et d'exemples. Il y a sûrement encore des erreurs de typographie et des imprécisions. La transmission à l'auteur de tout commentaire ou suggestion de corrections à apporter sera appréciée.

Table des matières

1 Introduction **1**

2 Éléments d'architecture d'ordinateurs **3**
- 2.1 Concepts généraux . . . 3
- 2.2 Architecture Intel IA-32 . . . 4
  - 2.2.1 Environnement d'exécution d'un programme . . . 6
  - 2.2.2 Les registres . . . 6
  - 2.2.3 Niveaux de privilèges . . . 10
  - 2.2.4 Organisation de la mémoire . . . 10
  - 2.2.5 Les indicateurs . . . 14
  - 2.2.6 Les données . . . 15
  - 2.2.7 Les modes d'adressage . . . 18
  - 2.2.8 La pile . . . 19
  - 2.2.9 Les interruptions . . . 20
- 2.3 L'architecture 64 bits Intel® 64 . . . 21
  - 2.3.1 Environnement d'exécution d'un programme . . . 21
  - 2.3.2 Les registres . . . 24
  - 2.3.3 La pile . . . 24
  - 2.3.4 Les interruptions . . . 24
- 2.4 Architecture ARM . . . 25
- 2.5 Architecture ARM64 (AArch64) . . . 28
- 2.6 Les ABI (Application Binary Interface) . . . 30
- 2.7 Mémoire, stockage et I/O . . . 47

3 Systèmes d'exploitation **49**
- 3.1 Éléments d'un système d'exploitation . . . 49
- 3.2 La gestion des processus . . . 50
- 3.3 La gestion de la mémoire . . . 51

3.4 La gestion des périphériques . . . 53
3.5 La gestion des systèmes de fichiers . . . 53
3.6 La gestion des usagers et de la sécurité . . . 54
3.7 Les appels système . . . 54
3.8 Rôle de l'assembleur . . . 55
3.9 Choix d'un système d'exploitation . . . 55
3.9.1 Contraintes temporelles . . . 55
3.9.2 L'empreinte mémoire . . . 56
3.9.3 Les services requis . . . 56
3.9.4 Disponibilité des pilotes d'interface matérielle . . . 56
3.9.5 Choix d'ordonnancement . . . 56
3.9.6 Modèle de protection mémoire . . . 57
3.10 Systèmes d'exploitation les plus utilisés . . . 57
3.10.1 Systèmes de type Unix . . . 57
3.10.2 Systèmes de la famille Windows . . . 58
3.10.3 Systèmes pour le temps réel . . . 58

4 Introduction à Linux **59**
4.1 Le démarrage . . . 59
4.1.1 Pourquoi un `initrd` ? . . . 61
4.2 Le programme `init` . . . 62
4.3 Le système de fichiers . . . 63
4.4 La librairie libc . . . 64
4.5 L'espace mémoire . . . 66

5 Chaîne de développement **67**
5.1 La suite de compilation GCC . . . 67
5.2 Le compilateur C GNU gcc . . . 68
5.3 L'éditeur de liens GNU ld . . . 71
5.4 Les librairies . . . 72
5.4.1 Les librairies statiques . . . 72
5.4.2 Les librairies dynamiques . . . 73
5.5 Les outils de Binutils . . . 76
5.6 La compilation croisée . . . 77
5.6.1 Éléments de base . . . 77

6 Outils de développement **80**
6.1 L'analyseur de dépendances GNU make . . . 80
6.1.1 Règles explicites . . . 83
6.1.2 Règles implicites . . . 85
6.1.3 Macros . . . 87
6.1.4 Catalogue des règles implicites . . . 88
6.1.5 Cibles multiples . . . 88
6.1.6 Règles multiples . . . 90
6.1.7 Directives . . . 91
6.1.8 Exécution conditionnelle . . . 91
6.2 La suite Autotools . . . 92
6.3 Introduction à CMake . . . 92
6.3.1 Limitations de l'analyseur de dépendances GNU make . . . 92
6.3.2 Exemple simple d'utilisation . . . 93
6.3.3 Création d'une librairie . . . 97
6.3.4 Utilisation d'une librairie . . . 97
6.3.5 Versions interactives de CMake . . . 97
6.4 Introduction à GIT . . . 98
6.4.1 Configuration . . . 98
6.4.2 Créer un dépot local . . . 98
6.4.3 Dépot distant . . . 102
6.4.4 Branches . . . 103
6.4.5 GUI pour Git . . . 103
6.5 Introduction à Doxygen . . . 103
6.5.1 Pourquoi un générateur de documentation ? . . . 103
6.5.2 Un exemple simple . . . 104
6.6 La syntaxe Markdown . . . 111

7 L'IDE Eclipse-CDT **112**
7.1 Projets de compilation native . . . 112
7.2 L'éditeur de code . . . 113
7.2.1 Intégration avec Doxygen . . . 116
7.3 Projets de compilation croisée . . . 117
7.3.1 Exemple : compilation croisée pour le Galileo d'Intel . . . 117
7.3.2 Exemple : compilation croisée pour le Odroid-C2 . . . 121

8 Le système de meta-distribution Yocto **125**
8.1 Architecture du Yocto Project . . . 125
8.2 Éléments de base . . . 127
8.2.1 Recettes (fichiers `*.bb`) . . . 128
8.2.2 Configuration (fichier `*.conf`) . . . 131
8.3 Création d'images et de SDK . . . 132
8.4 Commandes pratiques à connaitre . . . 133

9 Traitement des entrées/sorties **134**
9.1 Introduction . . . 134
9.2 Classes de pilotes . . . 135
9.3 Modes d'exécution . . . 136
9.4 Compilation d'un module . . . 137
9.5 Types de données et variables globales . . . 137
9.6 Aspects généraux d'un pilote . . . 138
9.7 Pour aller plus loin . . . 141

Bibliographie **142**

A Représentation des nombres **143**
A.1 Introduction . . . 143
A.2 Représentation binaire . . . 143
A.2.1 Passage binaire à décimal . . . 143
A.2.2 Passage de décimal à binaire . . . 144
A.3 Représentation octale . . . 144
A.3.1 Passage d'octal à décimal . . . 144
A.3.2 Passage de décimal à octal . . . 144
A.3.3 Passage de binaire à octal . . . 145
A.3.4 Passage d'octal à binaire . . . 145
A.4 Représentation hexadécimale . . . 145
A.4.1 Passage d'hexadécimal à décimal . . . 145
A.4.2 Passage décimal à hexadécimal . . . 146
A.4.3 Passage de binaire à hexadécimal . . . 146
A.4.4 Passage de hexadécimal à binaire . . . 146
A.4.5 Passage de octal à hexadécimal et de hexadécimal à octal . . . 146
A.5 Nombres entiers signés et non signés . . . 147
A.5.1 Représentation des entiers signés en complément à 2 . . . 147
A.6 Addition d'entiers . . . 148

A.7 Soustractions d'entiers 150
A.8 Nombres à virgule flottante : format IEEE 754 151

B Codes ASCII **154**

C Génération d'un compilateur croisé **156**
C.1 Téléchargement 157
C.2 Binutils 157
C.3 Fichier d'entête du noyau 158
C.4 GCC initial 158
C.5 Glibc fichiers d'entête et de *startup* 158
C.6 Librairies de support du compilateur 159
C.7 Compilation de Glibc 159
C.8 Librairie C++ et installation finale 159

Table des figures

2.1 Architecture du processeur i386 5
2.2 Environnement d'exécution d'un programme 32 bits. 7
2.3 Ensemble des registres. 8
2.4 Registres 8 bits, 16 bits et 32 bits. 8
2.5 Anneaux de protection. 10
2.6 Organisation de la mémoire. 11
2.7 Modèles de mémoire. 12
2.8 Modèle de mémoire : *Protected Flat*. 13
2.9 Traduction des adresses virtuelles. 13
2.10 Registre des indicateurs. 14
2.11 Types de données. 16
2.12 Données en mémoire. 16
2.13 Types de données. 17
2.14 Pointeur ou adresse mémoire. 18
2.15 La pile. 19
2.16 Registres de contrôle disponibles au niveau 0. 22
2.17 Environnement d'exécution d'un programme 64 bits. 23
2.18 Les principaux registres sous ARM. 26
2.19 Les modes sous ARM. 27
2.20 Le registre des indicateurs sous ARM. 27
2.21 Résolution d'adresse virtuelle (*Short-descriptor format*) sous ARM. 28
2.22 Les principaux registres sous ARM64. 29
2.23 Les *Exception Level* sous ARM64. 29
2.24 Résolution d'adresse virtuelle (*Short-descriptor format*) sous ARM64. 30
2.25 APCS de ARM. 41
2.26 Registres de l'ABI de ARM64. 45

3.1 Image mémoire d'un processus 52

4.1 Architecture de Linux (source IBM) 60

5.1 Environnement de programmation sous Linux 69

6.1 Dépendances entre différents fichiers 81
6.2 Dépendances entre différents fichiers 93
6.3 Projet dans Eclipse 96
6.4 Interface graphique pour Doxygen 106
6.5 Exemple de page HTML de Doxygen 106
6.6 Exemple de page principale HTML de Doxygen 109
6.7 Exemple de page de fonction documentée HTML de Doxygen . . . 110

7.1 `File->New` sous Eclipse 113
7.2 `File->New->C Project` sous Eclipse 114
7.3 Nouvelle fonction incomplète 114
7.4 Ajout d'un `for` et d'un commentaire 115
7.5 Préférences pour *plugin* Eclox. 116
7.6 Commentaires avec fonctionnalités Doxygen dans Eclipse. 117

8.1 Environnement Yocto 126
8.2 Couches sous Yocto 127

Liste des tableaux

5.1 Principales options utilisées avec gcc 70
5.2 Principales opérations et options utilisées avec ar 73

6.1 Principales options utilisées avec make 82
6.2 Macros prédéfinies sous make (liste partielle) 89
6.3 Règles implicites prédéfinies (liste partielle) 89
6.4 Macros prédéfinies des règles implicites (liste partielle) 90

A.1 Représentation binaire de la partie fractionnaire 0.153 152
A.2 Formats standards IEEE 754 153

B.1 Table ASCII (caractères 0-127) 154
B.2 Caractères de contrôle ASCII 155

Chapitre 1

Introduction

Le cours ELE4205 est une suite au cours ELE3312 - Microprocesseurs et applications. Dans ce dernier, vous avez programmé un système avec une tâche (*process*) unique. L'ordonnancement dans le temps de plusieurs fonctionnalités (entrée/sortie) a été réalisé à l'aide d'interruption et/ou de *timers*. Cette approche favorise la simplicité du design, une empreinte mémoire minimale, donc des coûts matériels faibles. Cette approche est appropriée pour des systèmes où un nombre restreint de fonctionnalités est pris en compte ou requis.

Dans un problème où de nombreuses interruptions (matérielles ou logicielles) se produisent, la gestion de tous ces évènements peut devenir fastidieuse et prône à des problèmes d'ordonnancement, de conflit entre les routines de gestion des interruptions et de gestion de la mémoire qui est partagée entre le programme principal et les « services » des interruptions. Par exemple, un système avec plusieurs entrées et sorties, un écran tactile et une connexion réseau avec serveur de configuration amènera le programmeur à peut-être réinventer la roue pour réaliser son design. De plus la solution étant « collée » au matériel, la migration vers une nouvelle architecture peut être très fastidieuse. En conclusion, la programmation sans système d'exploitation (*Bare-Metal programming*) a bien sa place dans les outils d'un ingénieur électrique, mais il faut bien connaitre les limites de cette approche.

Cependant, dans certains types de problème de conception, l'utilisation d'un système d'exploitation est plus appropriée. Le système d'exploitation introduit une couche d'abstraction matérielle qui peut rendre le design plus portable vers de nouvelles architectures. L'ordonnancement des tâches et la gestion des interruptions est sous le contrôle du noyau du système d'exploitation (le type de contraintes temporelles requises par le design auront une influence importante sur le choix

du système d'exploitation). Dans un système d'exploitation avec une gestion de la mémoire en « mode protégé », une application utilisateur ne peut pas accéder directement à l'espace mémoire du noyau donc ne peut pas causer de problème de stabilité. La stabilité du système est une qualité inherente du noyau qui peut être mise en évidence par l'exécution d'un appel d'une application qui causerais un problème. Dans ce cas, l'appel système met en évidence un bogue dans le noyau. Mais s'il n'y a pas de bogue dans le noyau, l'exécution une application fautive sera simplement terminée par le noyau. Le partitionment entre les applications et le noyau d'un système multitâches permet de découpler le design de diffentes fonctionnalités du système permettant ainsi de compartimenter les bogues facilitant ainsi le développement. Un système d'exploitation est souvent assorti d'outils et de bibliothèques logicielles qui rendent le développement plus rapide. Bon nombre de bibliothèques à code source ouvert sont très matures et contiennent très peu (ou pas) de bogues et nous évitent de réinventer la roue (par exemple pour la communication réseau : le *stack TCPIP* et les serveurs web sont des éléments solides, stables et éprouvés difficilement développables à l'interne d'une petite entreprise). Cependant, la disponibilité des pilotes d'interfaces matérielles pour les systèmes choisis et leur périphériques (ou la capacité de les développer soi-même) est un requis pour utiliser les interfaces matériel-logiciel.

Dans le cadre de ce cours, nous allons vous présenter différents éléments méthodologiques qui vous permettront de développer un « système embarqué » simple. Nous utiliserons le système d'exploitation Linux à titre d'exemple. Un bon nombre des éléments présentés pourront aussi s'appliquer à d'autres systèmes d'exploitation. Nous en profiterons aussi pour introduire certains outils d'aide à la programmation, à la documentation et au travail collaboratif.

Ce que vous savez (ou devriez savoir)

Comme le cours ELE3312 est un préalable, normalement, vous êtes familier avec les concepts de base d'architecture d'ordinateur, les langages C et C++. Aucune programmation en langage assembleur ne sera requise, mais nous vous présenterons l'analyse de différents codes assembleur générés par le langage C sous différentes architectures afin de mieux comprendre la partie matérielle de nos systèmes ainsi que l'impact de différentes options de compilation sur la « performance » de nos programmes.

Chapitre 2

Éléments d'architecture d'ordinateurs

Dans ce chapitre, nous allons vous présenter deux des architectures les plus utilisées que ce soit dans les systèmes portables (ou embarqués) allant jusqu'aux serveurs : l'architecture Intel (IA-32 et Intel® 64) et l'architecture ARM (ARMv7-A). Ces processeurs étant des d'architectures CISC et RISC respectivement. Les deux familles supportent la mémoire virtuelle à l'aide d'une unité de gestion de la mémoire (*MMU: memory management unit*) et un mode superviseur d'opération.

2.1 Concepts généraux

Une architecture typique d'ordinateur comprend une unité centrale de traitement (CPU), un bus de données, de la mémoire (qui contient les programmes et les données), des périphériques (O/I) qui s'interfacent avec différents dispositifs (disques, écrans graphiques, etc).

Le CPU comprends des registres à usages général et spécifique. Un registre de pointeur d'instruction pointe vers la mémoire où se situe la prochaine instruction à exécuter. Cette instruction est chargée dans le registre d'instruction pour être décodée puis exécutée. Cette exécution peut être un chargement d'une valeur en mémoire vers un registre (*load*), une opération entre deux registres (i.e. : addition), une copie d'une valeur d'un registre vers la mémoire (*store*), etc.

Les périphériques sont des contrôleurs situés à des adresses spécifiques en mémoire qui donnent accès à des registres de contrôle du périphérique et à des blocs mémoire. Par exemple, pour un disque dur, on peut écrire un bloc de données dans un tampon et demander que ce bloc soit écrit sur le disque à un endroit précis spécifié par une valeur chargée dans un registre de contrôle du contrôleur

de disque. Ce type de tâche est mis en oeuvre dans les pilotes d'interface (*drivers*) des systèmes d'exploitation.

2.2 Architecture Intel IA-32

Dans l'évolution des x86 d'Intel, le 80386 fut le premier de cette famille de processeurs à combiner une architecture 32 bits, l'adressage virtuel, la pagination et la protection mémoire. Même si les processeurs qui ont suivi le 80386 ont introduit de nouvelles capacités, c'est sur l'architecture du 80386 que repose la base de plusieurs systèmes d'exploitation 32 bits multi-tâches disponibles sous l'architecture Intel : Linux, FreeBSD, Solaris, Windows XP, 7, etc.

Évolution des processeurs Intel

Voici un bref survol de l'évolution des processeurs Intel (de 1978 à 1999) :

- 8086 : Registres 16 bits. Bus des données 16 bits. Bus des adresses 20 bits (espace adressable : 1 MégaOctets).
- 8088 : Registres 16 bits. Bus des données 8 bits. Bus des adresses 20 bits.
- 80286 : Registres 16 bits. Bus des données 16 bits. Bus des adresses 24 bits (espace adressable de 16 MégaOctets dû à l'adressage virtuel).
- 386 : Registres de 32 bits. Ces registres peuvent être utilisés en 16 bits et 32 bits. Bus des adresses de 32 bits (espace adressable de 4 GigaOctets). Adressage virtuel avec des pages de 4 KiloOctets. Cette architecture accepte des variantes du système d'exploitation UNIX (figure 2.1).
- 486 : Registres de 32 bits. Intégration du coprocesseur (pour les calculs en point flottant). Unité d'exécution plus rapide avec un pipeline à cinq étages. Ajout d'une mémoire cache de premier niveau de 8 Ko.
- Pentium : Registres de 32 bits. Bus de données de 64 bits. Bus d'adresse de 32 bits. Chemins internes (128 et 256 bits) au processeur de données sont ajoutés pour accélérer le transfert. Deux pipelines et un cache de premier niveau de 16 Ko.
- Pentium Pro : 3 instructions par coup d'horloge (trois pipelines). Ajout d'une mémoire cache de deuxième niveau de 256-512 Ko. Bus des données de 64 bits. Bus d'adresse de 36 bits (espace adressable de 64 GigaOctets).
- Pentium II et III : instructions MMX, mémoires caches de premier niveau 32 Ko et de deuxième niveau 256 Ko à 2 Mo, etc.

Après 1999, d'autres version et extensions sont apparues : SSE, AVX, etc.

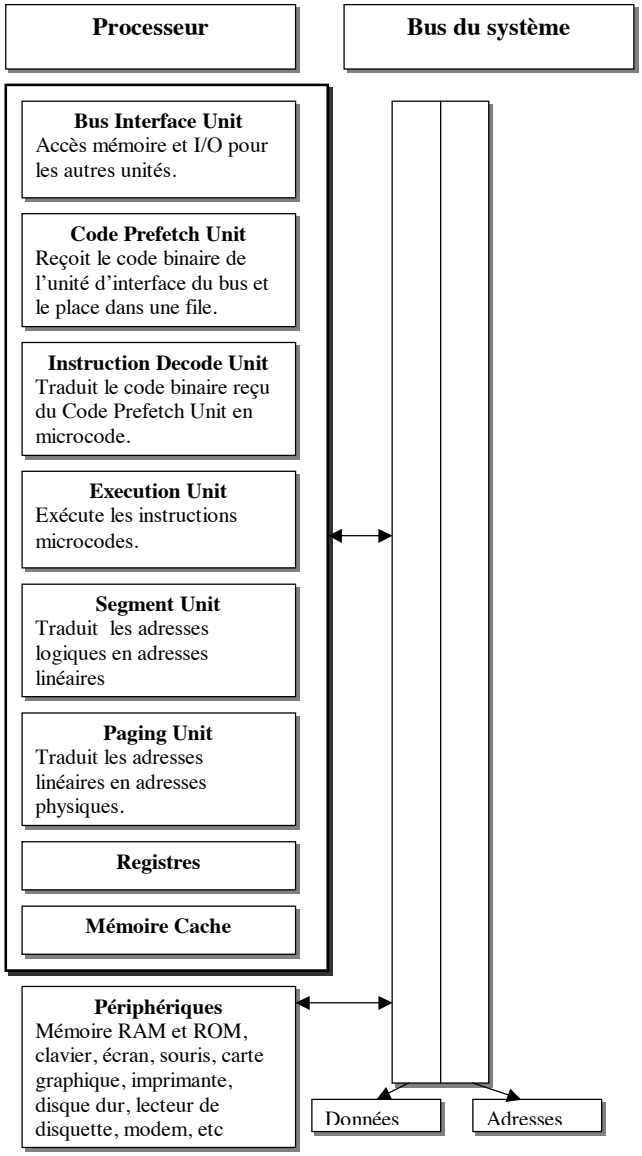


FIGURE 2.1 – Architecture du processeur i386

En 2003, AMD a introduit une extension de l'architecture 32 bits d'Intel, x86-64. Celle-ci sera reprise par Intel suite au peu de succès de l'Itanium-64.

Sous gcc (la suite de compilation), on pourra spécifier la génération du processeur cible : i386, i486, i586, i686, etc.

2.2.1 Environnement d'exécution d'un programme

L'environnement d'exécution d'un programme (figure 2.2) est composé des éléments suivants :

- L'espace adressable contenant l'espace mémoire pour le code du programme, les données et la pile. La pile est très importante lors de l'exécution d'un programme, car lors de l'appel de fonctions, le programme utilise cet espace de façon temporaire pour les arguments de la fonction, l'adresse de retour à l'appelant, les variables locales et la sauvegarde de certains registres.
- Huit registres généraux de 32 bits pouvant être utilisés avec diverses instructions.
- Six registres de segments de 16 bits (ces registres ne sont pas utilisés directement par les programmes utilisateurs en allocation mémoire 32 bits linéaire, mais sont utilisés par le noyau dans la gestion de la protection).
- Un registre de 32 bits contenant l'état des indicateurs : EFLAGS.
- Un registre pointeur d'instruction EIP.

2.2.2 Les registres

La figure 2.3 présente l'ensemble des registres du i386 (registres généraux, de segments, des indicateurs et du pointeur d'instruction).

Les registres généraux

Les registres généraux peuvent être utilisés comme

- opérandes dans les opérations logiques et arithmétiques ;
- opérandes dans les calculs d'adresses ;
- pointeurs mémoires.

Les huit registres généraux sont [1] :

1. Dans un programme assembleur sous GAS les noms des registres sont précédés du préfixe % afin d'éviter les conflits potentiels avec des noms d'étiquettes. Ainsi, le registre EAX est donné par `%eax`.

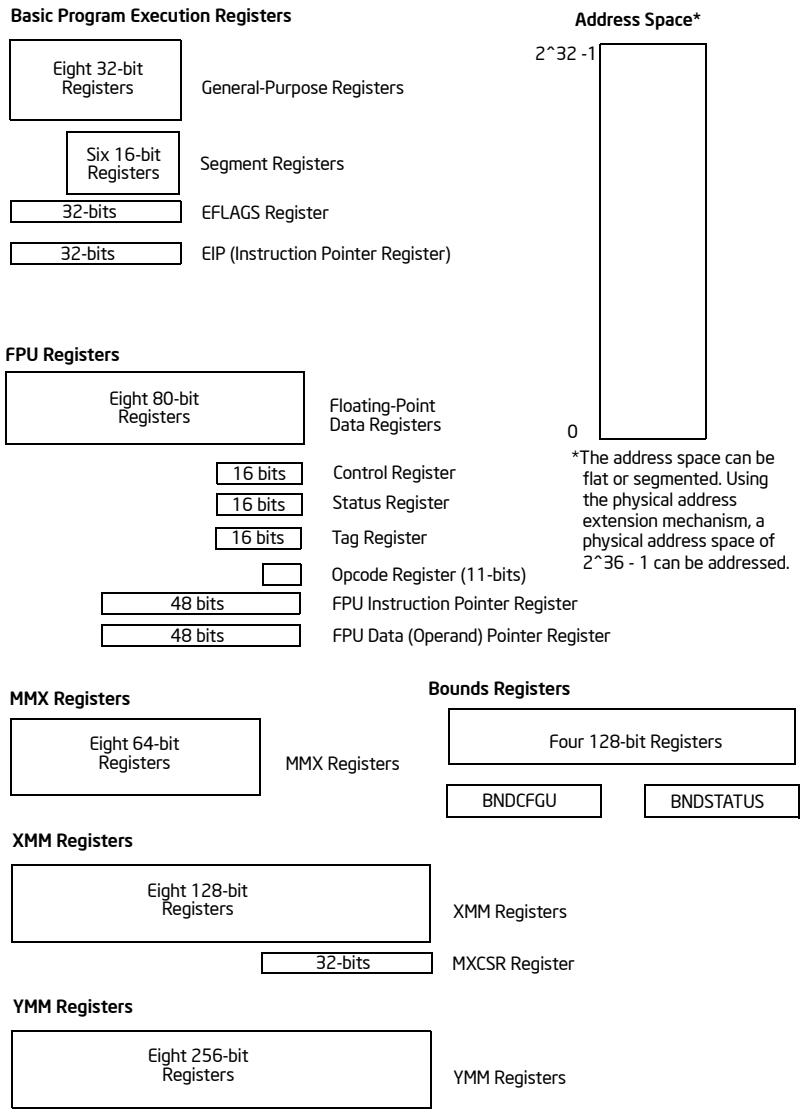


FIGURE 2.2 – Environnement d'exécution d'un programme 32 bits. Tiré de [4].

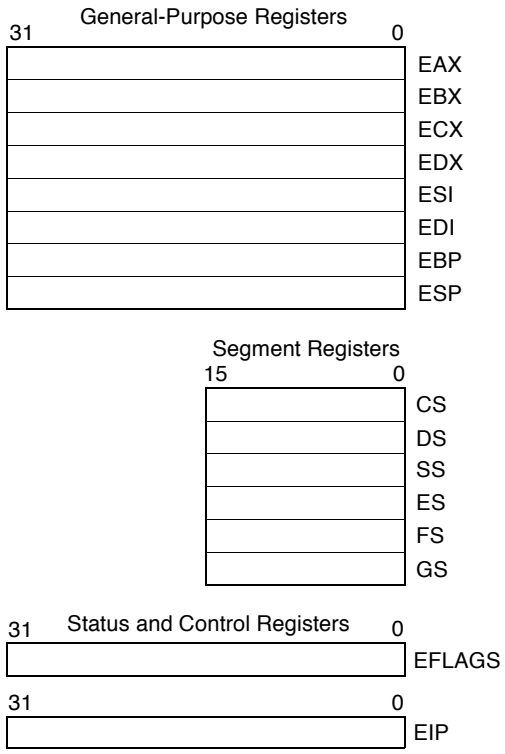


FIGURE 2.3 – Ensemble des registres. Tiré de [4].

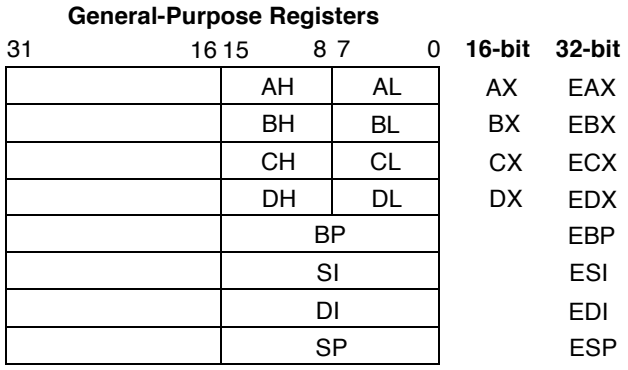


FIGURE 2.4 – Registres 8 bits, 16 bits et 32 bits. Tiré de [4].

— EAX : Accumulateur. Pour les instructions arithmétiques (multiplication et division).
— EBX : Base. Pointeur à des données.
— ECX : Compteur : Compteur pour les instructions sur les chaînes de caractères. Compteur pour les boucles.
— EDX : Data. Pour les instructions arithmétiques (multiplication et division) et comme pointeur d'entrée/sortie.
— ESI et EDI : Source Index et Destination Index. Utiles pour les instructions des chaînes de caractères.
— ESP : Stack Pointer. Adresse du sommet de la pile lors de l'exécution d'un programme. À modifier avec beaucoup de soin !
— EBP : Base Pointer. Registre à utiliser avec la pile d'un programme. Ce registre est très important lorsque l'on abordera l'interface avec le C.

Les 16 bits les moins significatifs des registres EAX, EBX, ECX et EDX peuvent être utilisés : AX, BX, CX et DX, de même que la partie haute de 8 bits (AH, BH, CH et DH) et la partie basse (AL, BL, CL et DL).

Les registres de segments : 16 bits

Ces segments sont utilisés pour l'adressage segmenté :
— CS : Code Segment. L'adresse du code du programme.
— DS : Data Segment. L'adresse des données du programme.
— SS : Stack Segment. L'adresse de la pile du programme.
— ES : Extra Segment. Généralement une adresse des données.
— FS et GS : pour le mode protégé.

Le registre des indicateurs EFLAGS

Ce registre a une signification bit par bit. Le processeur place dans ce registre le résultat de certaines opérations (débordement de capacité, signe d'un nombre, etc). Ce registre n'est pas accessible directement au programmeur. Par contre, on peut à l'aide de certaines instructions, sauver son contenu dans le registre EAX.

Le pointeur d'instruction

EIP : contient l'adresse dans l'espace mémoire du code de la prochaine instruction à exécuter. Ce registre n'est pas accessible par le programmeur. D'ailleurs,

il n'en n'a pas besoin. On peut voir son contenu lorsque l'on dévermine un programme.

Autres registres

Il existe d'autres registres pour le débogage, la gestion de la mémoire virtuelle, le calcul point flottant, les MMX, les SSE, la virtualisation, etc. On se référera au document [4] pour plus de détails.

2.2.3 Niveaux de privilèges

En mode protégé le système a quatre niveaux de privilèges : le niveau 0 étant de privilège élevée alors que le niveau 3 est de privilège minimum. La figure 2.5 illustre le concept sous forme d'anneaux de privilèges (*protection rings*). Si seulement deux niveaux sont utilisés, le noyau sera au niveau 0 alors que les applications usager seront au niveau 3.

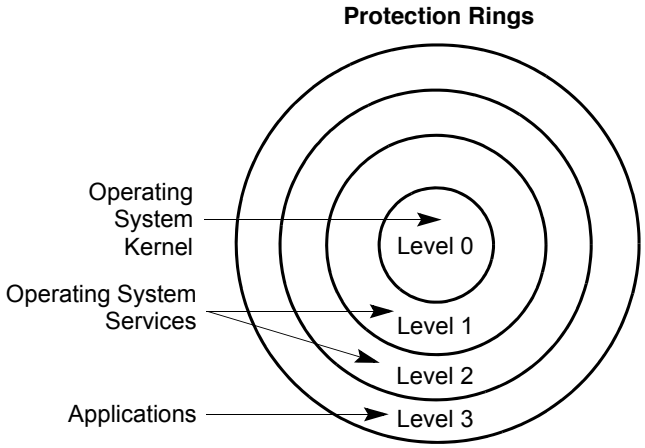


FIGURE 2.5 – Anneaux de protection. Tiré de [4].

2.2.4 Organisation de la mémoire

Chaque octet de mémoire a une adresse. La mémoire est une série d'octets. Les processeurs Intel sont des machines "petit bout" (*little endian*, voir [3]), c'est-

à-dire lorsqu'on veut stocker une donnée de plusieurs octets en mémoire, on place d'abord l'octet le moins significatif à l'adresse la plus basse (figure 2.6). La plus petite unité adressable est un octet.

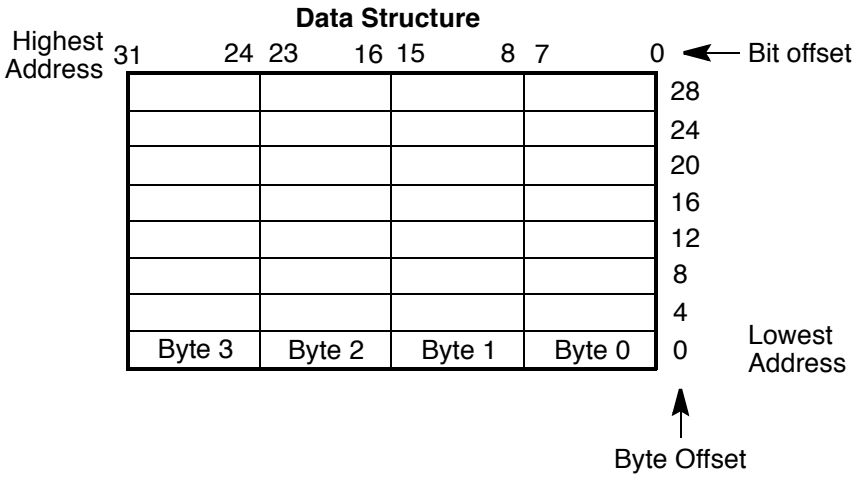


FIGURE 2.6 – Organisation de la mémoire. Tiré de [4].

Définitions

— Adresse physique : adresse mémoire envoyée dans le bus des adresses.
— Adresse linéaire : une adresse dans l'espace mémoire. (*flat model*, voir figures 2.7 et 2.8).
— Adresse segmentée : adresse mémoire composée d'un registre de segment et un offset (voir figure 2.7).

Le système d'exploitation Linux utilise des adresses linéaires avec protection.

Principes de la mémoire virtuelle

La mémoire vive est divisée en pages de 4 KiloOctets chacune. De même, certaines de ces pages pourront être sur une unité de disque rigide. Pour accéder à la mémoire physique ou pour constater que la page n'est pas en mémoire mais sauvegardée sur le disque rigide, une adresse linéaire est décomposée en :

— bit 31 à bit 22 : un indice dans le répertoire des pages. Le répertoire de pages peut contenir 1024 entrées correspondants à des tables de pages.

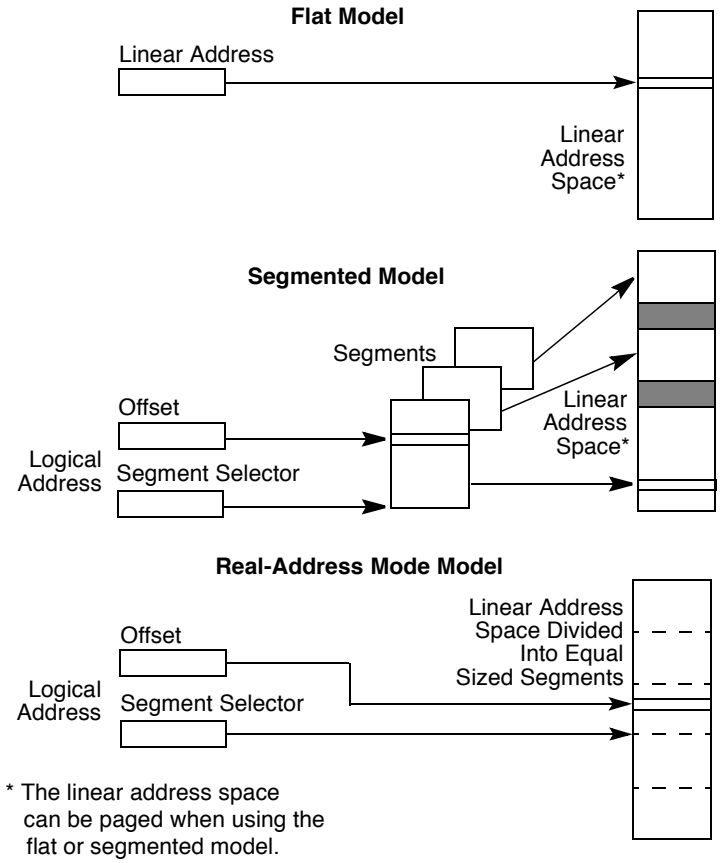


FIGURE 2.7 – Modèles de mémoire. Tiré de [4].

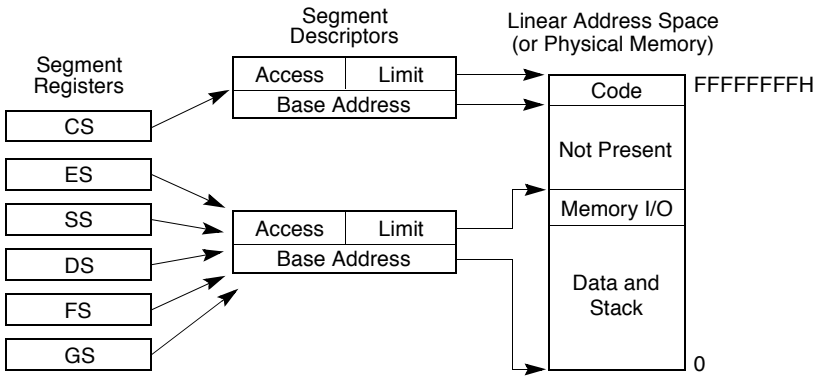


FIGURE 2.8 – Modèle de mémoire : *Protected Flat*. Tiré de [4].

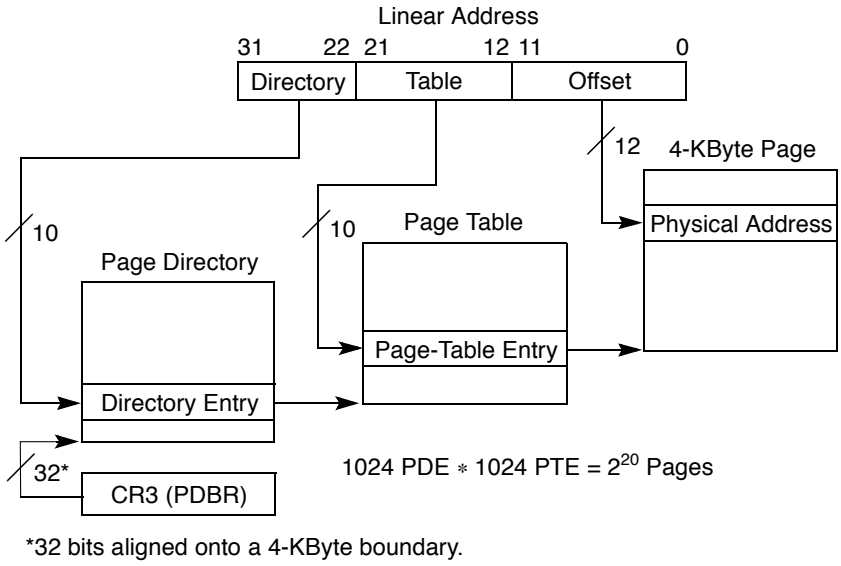


FIGURE 2.9 – Traduction des adresses virtuelles. Tiré de [4].

— bit 21 à 12 : un indice dans la table des pages. 1024 entrées contenant l'adresse physique de la page mémoire et de l'information quand au statut de la page (page présente en mémoire, permissions lecture/écriture, etc : voir la section B-4 sur la mémoire virtuelle de [3]).
— bit 11 à bit 0 : déplacement par rapport à l'adresse de la page mémoire. L'intervalle de la distance est [0,4095] pour 4096 octets.

Une série de pages est réservée pour le code d'un programme, pour les données globales initialisées, les données globales non initialisées et la pile. Une adresse linéaire est donc une adresse virtuelle. Toutes les pages d'un programme n'ont pas à être simultanément chargées dans la mémoire physique. (figure 2.9).

2.2.5 Les indicateurs

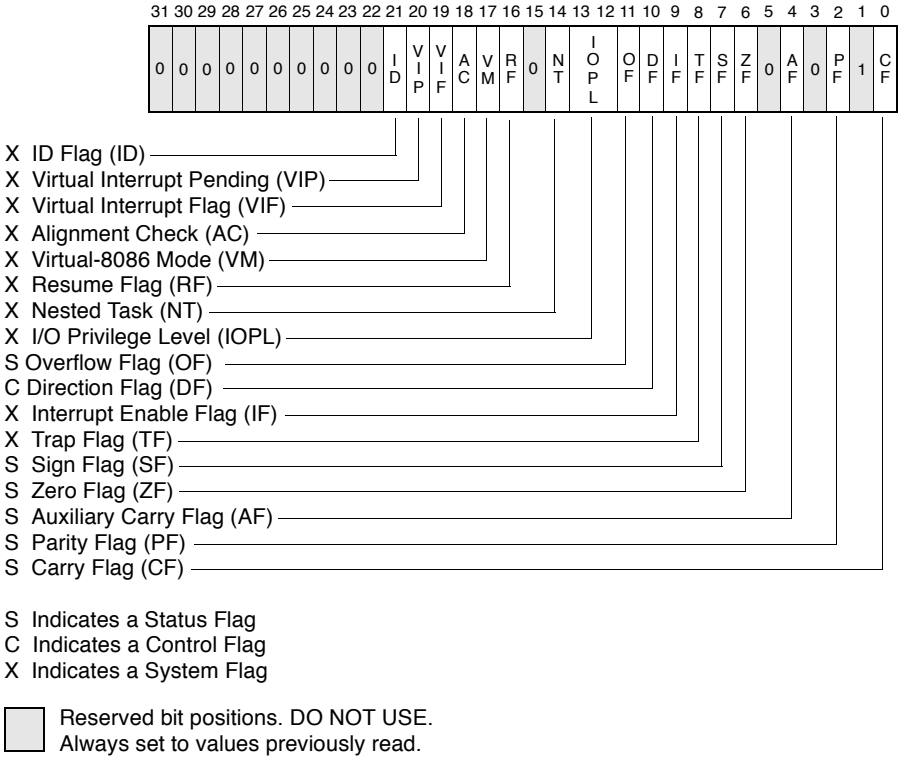


FIGURE 2.10 – Registre des indicateurs. Tiré de [4].

Le registre des indicateurs EFLAGS permet d'obtenir de l'information sur le

résultat de la dernière instruction effectuée par le processeur. Par exemple, après une instruction de soustraction dont le résultat donne zéro (0), le bit du *zero flag* (ZF) sera un (1). Le registre des indicateurs a 32 bits, certains n'étant pas utilisés ou réservés (voir figure 2.10) :

- CF : carry flag (Bit 0). Débordement des opérations d'addition et de soustraction sur des entiers non signés. Bit de retenue.
- PF : parity flag (Bit 2). 1 si le nombre de bits à 1 est pair dans l'octet le plus significatif, 0 sinon.
- AF : adjust flag (Bit 4). Débordement pour les nombres BCD (Binary Coded Decimal)
- ZF : zéro flag (Bit 6). 1 si le résultat est 0 après l'exécution d'une instruction.
- SF : signe flag (Bit 7). 1 si le résultat est négatif, sinon 0.
- DF : direction flag (Bit 10). Contrôle les instructions sur les chaînes de caractères : incrémentation si 0, décrémentation si 1.
- OF : overflow flag (Bit 11). Débordement des opérations d'addition et soustraction sur des entiers signés en complément de deux.

Les autres indicateurs ne doivent pas être modifiés par un programme. Ils sont utilisés par le système d'exploitation.

Seulement, certaines instructions modifient ce registre. Les annexes A et B du volume 1 de [4] précisent quels sont les indicateurs qui sont modifiés par les différentes instructions.

2.2.6 Les données

En mémoire, les données peuvent être représentées par les types (voir figure 2.11) :

- Byte = 1 octet
- Word = 2 octets
- Doubleword = 4 octets (long)
- Quadword = 8 octets

Comme les processeurs Intel sont des machines petit bout (*little endian*), le données sont placées en mémoire comme illustré à la figure 2.12.

Les entiers représentés en mémoire pourront être considérés signés ou non signés (voir figure 2.13). Vous pouvez consulter l'annexe A pour un rappel de la représentation des nombres.

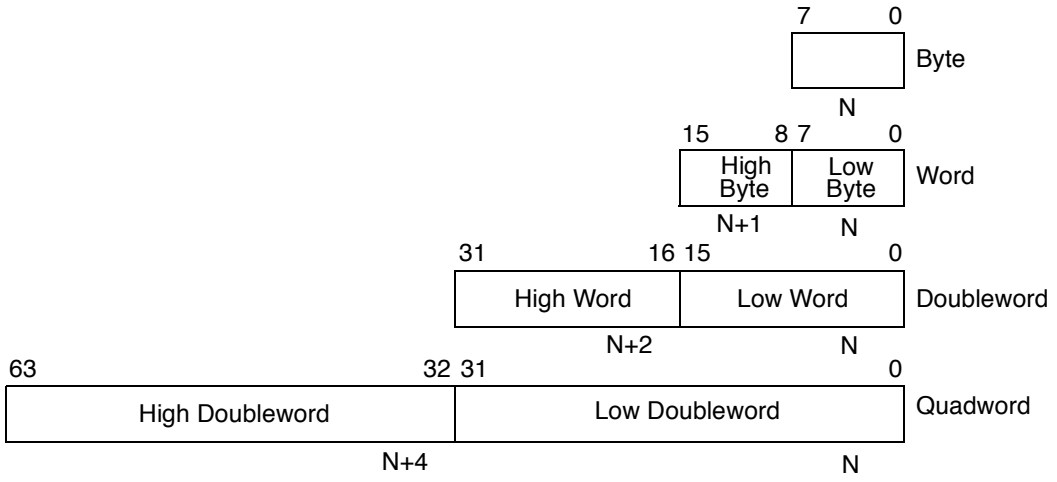


FIGURE 2.11 – Types de données. Tiré de [4].

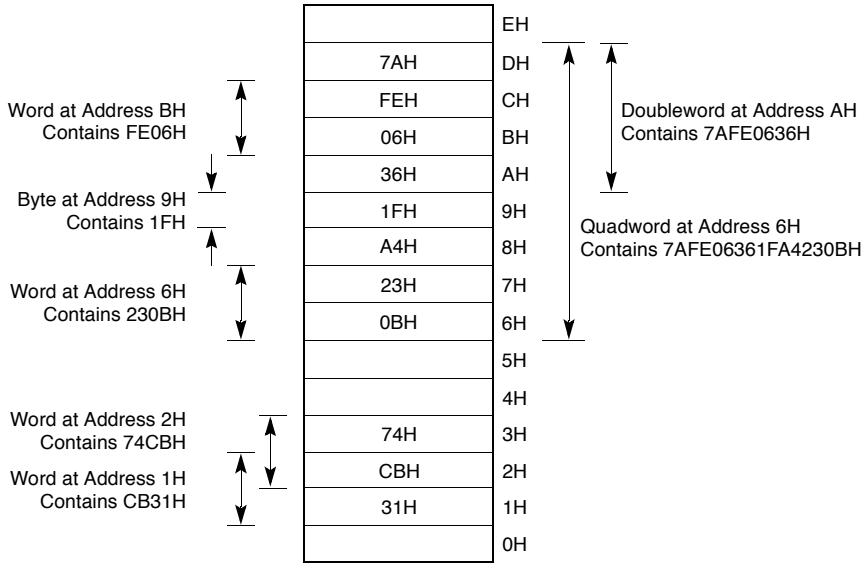


FIGURE 2.12 – Données en mémoire. Tiré de [4].

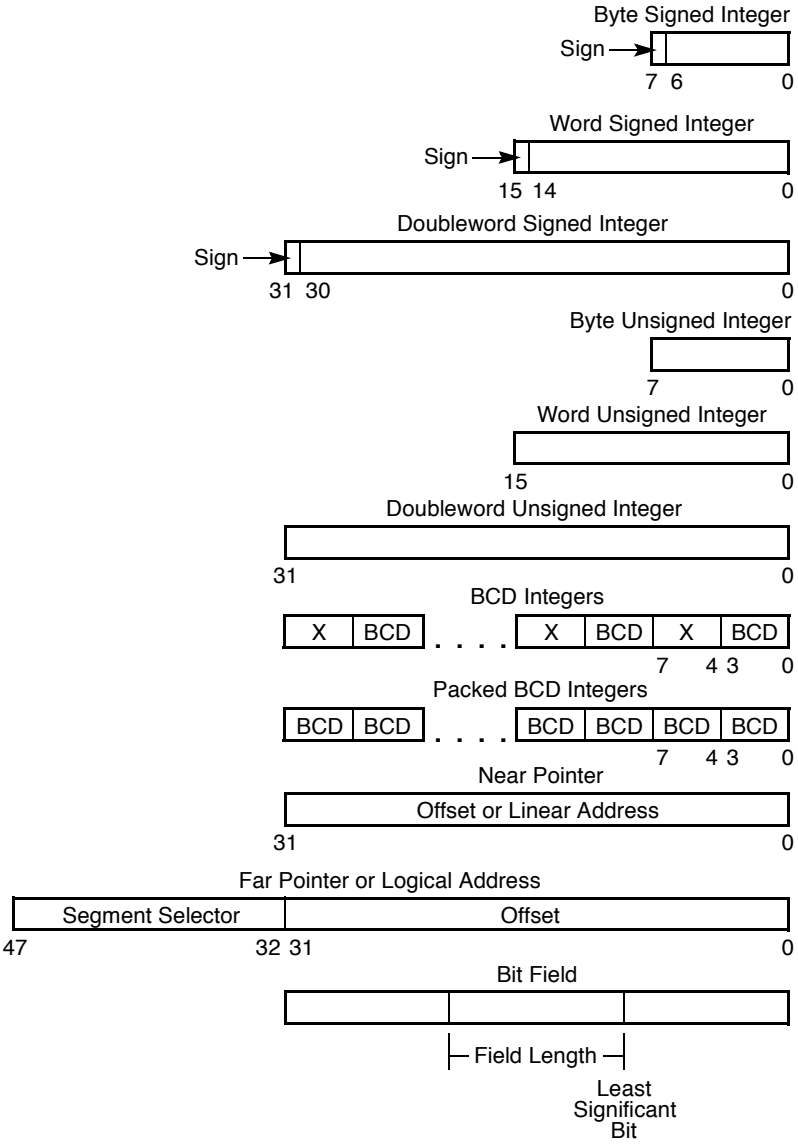


FIGURE 2.13 – Types de données. Tiré de [4].

Entiers signés en complément à 2 $[-2^{N-1}, 2^{N-1} - 1]$

où N = Nombre de bits.
— 1 octet = $[-128, 127]$
— 2 octets = $[-32768, 32767]$
— 4 octets = $[-2^{31}, 2^{31} - 1]$
— 8 octets = $[-2^{63}, 2^{63} - 1]$
— 16 octets =$[-2^{127}, 2^{127} - 1]$

Entiers non signés $[0, 2^N - 1]$

où N = Nombre de bits
— 1 octet = $[0, 255]$
— 2 octets = $[0, 65535]$
— 4 octets = $[0, 2^{32} - 1]$
— 8 octets = $[0, 2^{64} - 1]$
— 16 octets =$[0, 2^{128} - 1]$

2.2.7 Les modes d'adressage

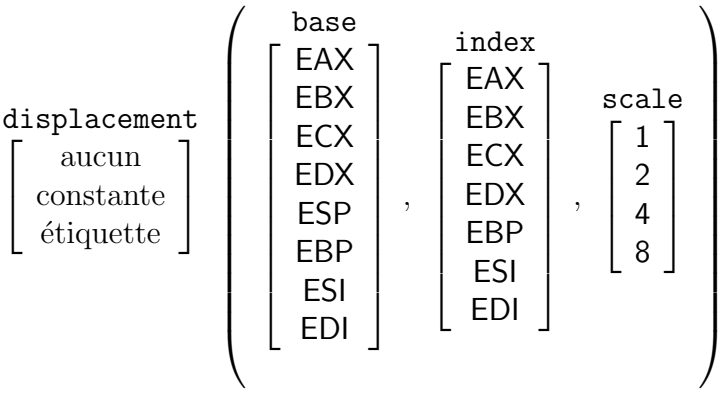


FIGURE 2.14 – Pointeur ou adresse mémoire.

Les processeurs i386 d'Intel permet d'utiliser différents modes d'adressage. Les valeurs immédiates (valeur connue à la compilation) : constante précédé d'un $, ou adresse d'une étiquette peuvent être utilisées pour spécifier une adresse

(par exemple $0x8049080 ou `$unentier`). La figure 2.14 vous montre les registres autorisés dans la spécification d'une adresse. Dans la syntaxe AT&T, une adresse peut avoir la forme suivante : `displacement(base,index,scale)`, qui résulte en l'adresse : `displacement+base+(index*scale)`.

Les registres EAX, EBX, ECX, EDX, ESI, EDI réfèrent à une adresse dans l'espace des données.

Les registres ESP, EBP réfèrent à une adresse dans l'espace de la pile.

Le registre ESP ne peut pas servir d'index.

Rappelons que le signe `$` est utilisé pour une valeur immédiate. `$etiquette` est l'adresse mémoire de `etiquette` alors que `etiquette` est la valeur pointée par son adresse.

2.2.8 La pile

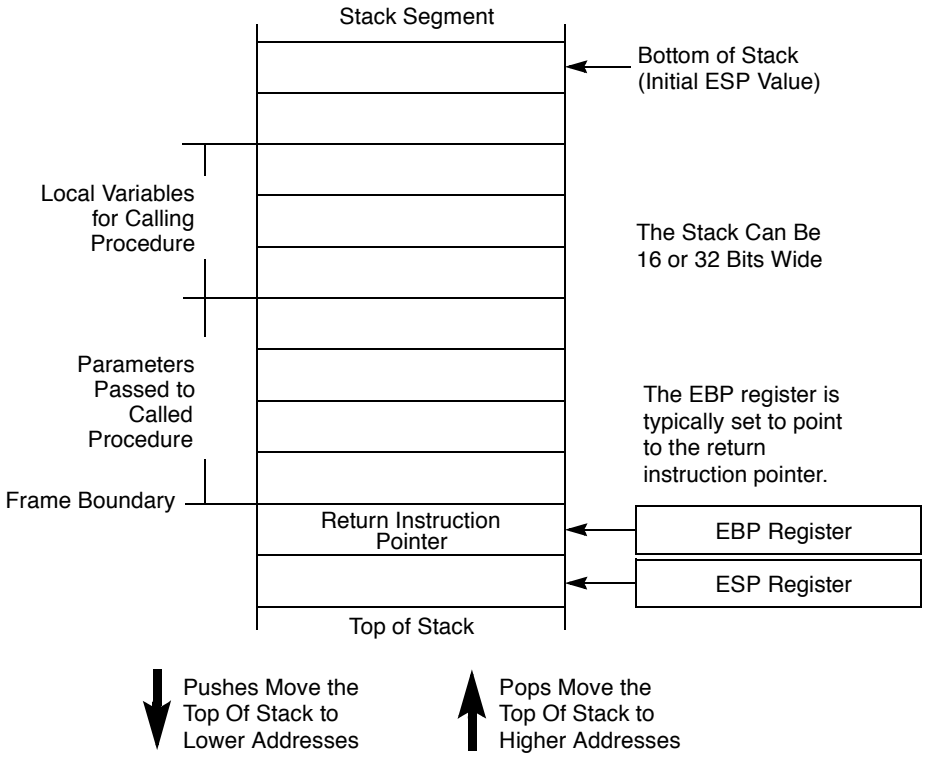


FIGURE 2.15 – La pile. Tiré de [4].

La pile est un espace mémoire qui est géré principalement à l'aide du registre ESP qui pointe au sommet de cette pile. La figure 2.15 présente la structure de la pile qui sera 32 bits de large (4 octets). On utilise aussi le registre EBP pour accéder à des éléments de la pile. La pile sert :

- à conserver l'adresse de retour lors de l'appel d'une fonction ;
- à passer des arguments à une fonction ;
- à réserver temporairement de l'espace mémoire pour les variables locales des fonctions ;
- à sauver temporairement le contenu des registres ;
- à sauver le contenu des indicateurs lors d'une interruption.

Les opérations de la pile sont `push` ou `pop`. Par opération, on ne peut empiler que 4 octets et dépiler 4 octets. Lors d'un `push`, le pointeur ESP décroît de 4 octets, et lors d'un `pop`, le pointeur ESP augmente de 4 octets.

2.2.9 Les interruptions

Les interruptions sont des situations durant lesquelles l'ordre normal d'exécution des instructions est modifié. Parmi les différents types d'interruption que l'on retrouve sous le 386, nous allons décrire très brièvement les interruptions générées par une requête d'un composant matériel d'entrée-sortie et celles qui résultent d'un appel système fait par un programme.

Les interruptions matérielles (IRQ – interrupt request)

Un composant matériel d'entrée-sortie génère une interruption lorsqu'il doit signifier au processeur qu'un « événement » vient de se produire. Les IRQ sont numérotés et un des ces numéros est associé à chaque composant. De plus, une fonction d'interruption (ISR – interrupt service routine) est associée à chacun des IRQ.

Ainsi, lors d'une interruption, le processeur interrompt l'exécution du programme en cours, et débute l'exécution de l'ISR associé à IRQ qui a été généré. Lorsque la fonction d'interruption a terminé son exécution, le processeur retourne continuer l'exécution du programme interrompu.

Les interruptions logicielles (INT)

Les interruptions logicielles sont des instructions incluses dans le programme qui permettent de transférer le contrôle de l'exécution à un autre processus (au

système d'exploitation par exemple). Ainsi, l'instruction :

```
int n
```

où `n` est le numéro de l'interruption, transfère l'exécution à l'interruption logicielle `n`. Lorsque la fonction d'interruption a terminé son exécution, le processeur retourne continuer l'exécution du programme interrompu. Sous Linux 32 bits, les appels systèmes sont exécutés via l'interruption :

```
int $0x80
```

Registres de contrôle disponibles sous le kernel

En plus des registres généraux, le noyau (*kernel*) d'un système d'exploitation en mode protégé a accès à des registres de contrôle pour la gestion de la mémoire, des processus, des interruptions, etc (voir figure 2.16).

2.3 L'architecture 64 bits Intel® 64

Développée initialement par AMD, l'architecture Intel® 64 (aussi nommé AMD64 et x86-64) est une extension de l'architecture 32 bits. La grande majorité des registres deviennent 64 bits et prennent le préfixe R : par exemple EAX devient RAX. Ici, nous allons ne présenter que les principales différences avec l'IA-32.

2.3.1 Environnement d'exécution d'un programme

L'environnement de d'exécution d'un programme (figure 2.2) est composé des éléments suivants :

- L'espace adressable contenant l'espace mémoire pour le code du programme, les données et la pile. La pile est très importante lors de l'exécution d'un programme, car lors de l'appel de fonctions, le programme utilise cet espace de façon temporaire pour les arguments de la fonction, l'adresse de retour à l'appelant, les variables locales et la sauvegarde de certains registres.
- Seize registres généraux de 64 bits pouvant être utilisés avec diverses instructions.
- Six registres de segments de 16 bits (qui ne seront pas utilisés dans le cadre du cours).
- Un registre de 64 bits contenant l'état des indicateurs : RFLAGS.
- Un registre pointeur d'instruction RIP.

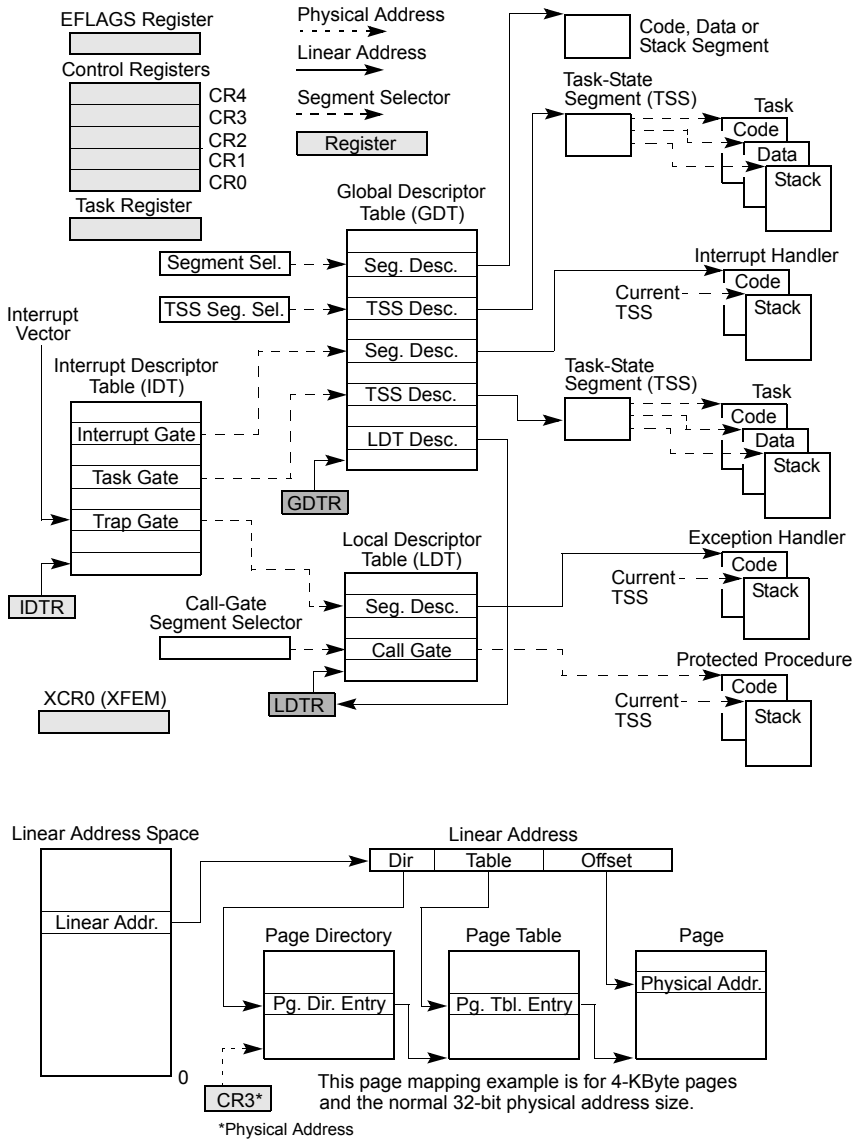


FIGURE 2.16 – Registres de contrôle disponibles au niveau 0. Tiré de [4].

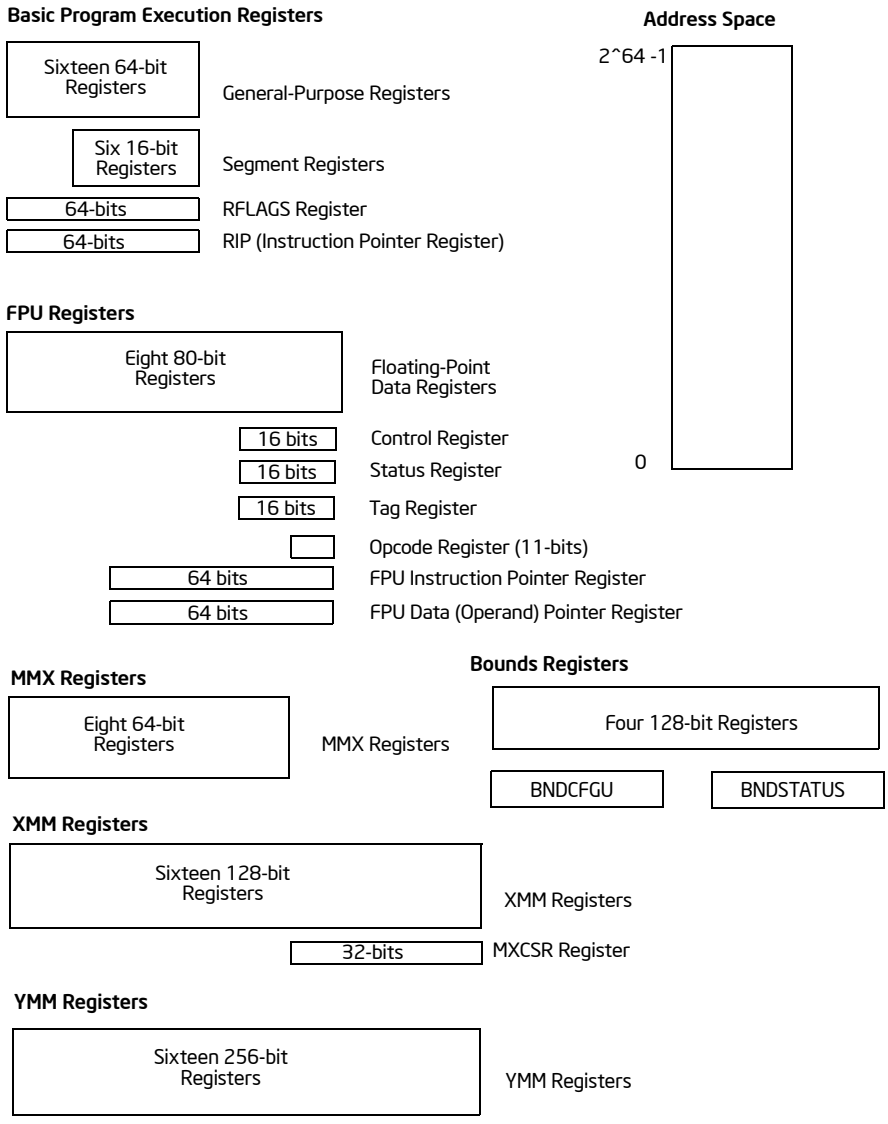


FIGURE 2.17 – Environnement d'exécution d'un programme 64 bits. Tiré de [4].

2.3.2 Les registres

Les registres généraux peuvent être utilisés comme
— opérandes dans les opérations logiques et arithmétiques ;
— opérandes dans les calculs d'adresses ;
— pointeurs mémoires.
Les seize registres généraux sont :
— RAX : Accumulateur. Pour les instructions arithmétiques (multiplication et division).
— RBX : Base. Pointeur à des données.
— RCX : Compteur : Compteur pour les instructions sur les chaînes de caractères. Compteur pour les boucles.
— RDX : Data. Pour les instructions arithmétiques (multiplication et division) et comme pointeur d'entrée/sortie.
— RSI et EDI : Source Index et Destination Index. Utiles pour les instructions des chaînes de caractères.
— RSP : Stack Pointer. Adresse du sommet de la pile lors de l'exécution d'un programme. À modifier avec beaucoup de soin !
— RBP : Base Pointer. Registre à utiliser avec la pile d'un programme. Ce registre est très important lorsque l'on abordera l'interface avec le C.
— les huit registres R8 à R15

2.3.3 La pile

La pile est 8 octets (64 bits) de large en mode x86-64 (voir figure 2.15).

2.3.4 Les interruptions

Les interruptions sont traitées de façon très similaire sous 64 bits.

Les interruptions logicielles (INTQ)

Les interruptions logicielles sont des instructions incluses dans le programme qui permettent de transférer le contrôle de l'exécution à un autre processus. Ainsi, l'instruction :

```
intq n
```

où `n` est le numéro de l'interruption, transfère l'exécution à l'interruption logicielle `n` (le suffixe `q` est pour quad). Lorsque la fonction d'interruption a terminé son

exécution, le processeur retourne continuer l'exécution du programme interrompu. Sous Linux 64 bits, les appels systèmes sont exécutés via l'instruction :

```
syscall
```

2.4 Architecture ARM

Dans cette section, nous allons décrire brièvement l'architecture du ARMv7-A, plus précisément celle du ARM Cortex™–A8.

Ce processeur comprend 16 registres de 32 bits : R0 à R12, et les registres SP, LR et PC (voir figure 2.18). Les registres R0 à R7 sont communs à tous les modes d'opération du processeur. Les registres R8 à R12 sont communs à tous les modes sauf le mode fiq pour lesquels une banque à part est prévue. SP (stack pointer : pour la pile), LR (link register : peut contenir le lien pour le retour d'une fonction) et PC (program counter : pour le flot d'exécution) ont aussi leurs banques prevues dans plusieurs modes.

Ce processeur peut passer d'un mode d'opération à l'autre lors de l'exécution :

User mode : le seul mode d'opération non-privilégié.

FIQ mode : mode privilégié qui est activé lorsque le processeur accepte une interruption rapide (*Fast Interrupt request*).

IRQ mode : mode privilégié qui est activé lorsque le processeur accepte une interruption.

Supervisor (svc) mode : mode privilégié qui est activé quand le CPU subit un *reset* ou que l'instruction `SVC` est exécutée.

Abort mode : mode privilégié qui est activé sur l'occurence d'un *prefetch abort* ou d'un *data abort exception*.

Undefined mode : mode privilégié qui est activé quand une exception d'instruction non-définie se produit.

System mode (ARMv4 and above) : le seul mode privilégié qui n'est pas activé sur une exception. On entre dans ce mode sur une instruction qui écrit explicitement dans le bit de mode du regitre CPSR.

Monitor mode (ARMv6 and ARMv7 Security Extensions : mode moniteur introduit pour le support des extensions *TrustZone*.

Hyp mode (ARMv7 Virtualization Extensions, ARMv8 EL2) : mode *hypervisor* pour la virtualisation.

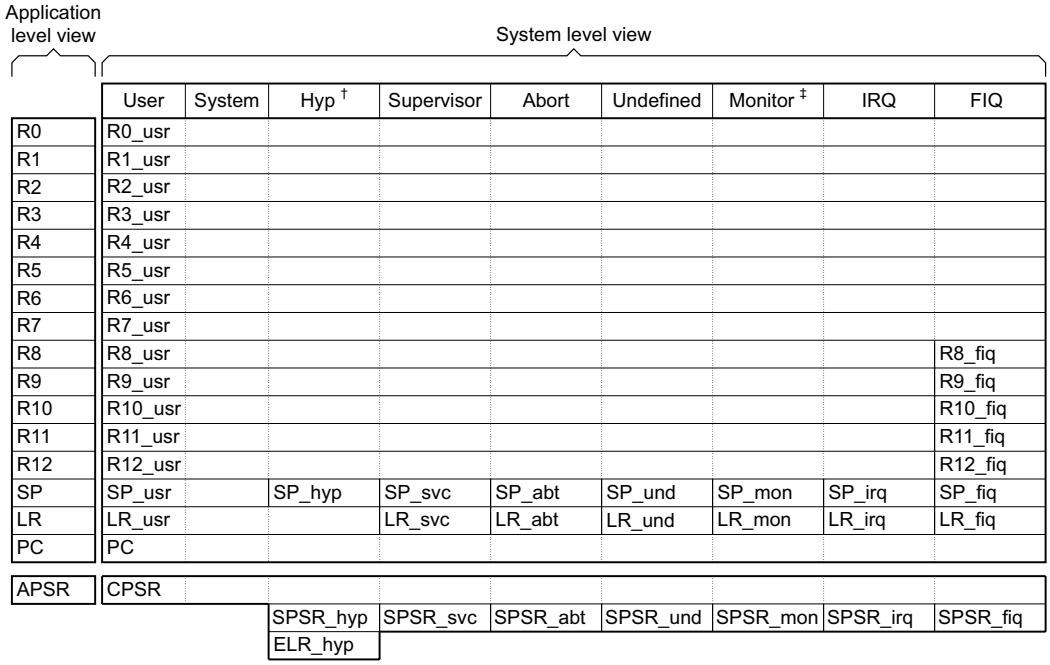


FIGURE 2.18 – Les principaux registres sous ARM. Tiré de [1].

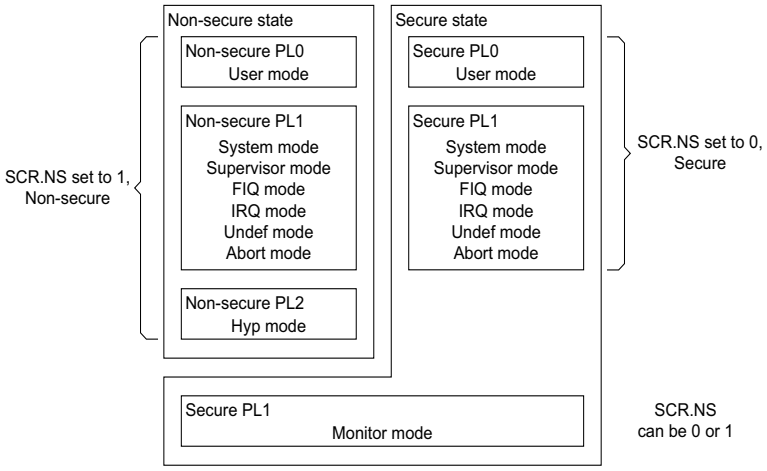


FIGURE 2.19 – Les modes sous ARM. Tiré de [1].

Les niveaux de privilèges sont présentés à la figure 2.19. Ici PL0 est de privilège inférieur à PL1. Du point de vue d'une application, dans le *user mode*, le registre APSR (*Application Program Status Register*) contient les indicateurs similaires à ceux sous architecture Intel (voir figure 2.20).

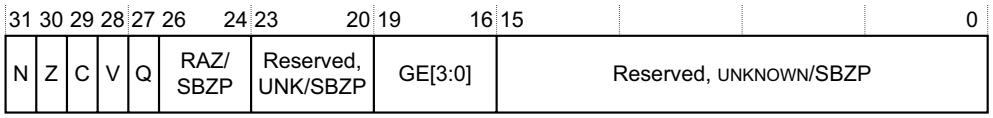


FIGURE 2.20 – Le registre des indicateurs sous ARM. Tiré de [1].

La mémoire est par défaut petit-bout bien qu'elle configuré gros-bout, alors que les instructions sont toujours encodés petit-bout. Un des modes de mémoire virtuelle (*Short-descriptor format*) permet d'avoir des regions mémoires de 1 Mo, 16 Mo, et des pages de 64 Ko et 4 Ko (voir figure 2.21).

Les types de données disponibles sont les mêmes que sous IA-32 à l'exception des BCD (*Binary Coded Decimals*) et des *Far Pointers*.

La pile est de 4 octets de large et part des adresses hautes et croit vers le bas. Son usage est similaire à celle sous Intel bien qu'elle ne soit utilisée qu'occasionnellement pour le passage de paramètres.

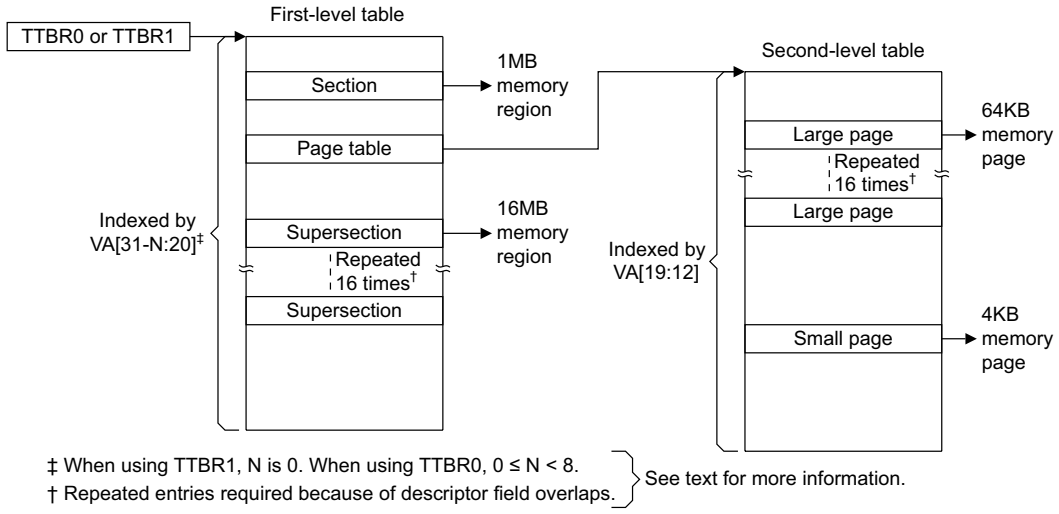


FIGURE 2.21 – Résolution d'adresse virtuelle (*Short-descriptor format*) sous ARM. Tiré de [1].

Les extensions NEON du processeur permettent le SIMD (32 registres de 64 bits supportant entiers et point-flottant simple précision).

2.5 Architecture ARM64 (AArch64)

Dans cette section, nous allons décrire brièvement l'architecture du ARMv8-A, plus précisément celle du ARM Cortex™–A53.

Ce processeur comprend 31 registres de 64 bits : R0 à R30 (X0 à X30 pour 64 bits et X0 à X30 pour 32 bits), et les registres FP (X39) et LR (X30) (voir figure 2.22). Les registres R0 à R30 sont communs à tous les modes d'opération du processeur. Il y a aussi le SP (stack pointer : pour la pile avec des versions pour les différentes modes), le LR (link register : peut contenir le lien pour le retour d'une fonction aec de version pour les exceptions) et le PC (program counter : pour le flot d'exécution).

Le registre XZR/WZR donne 0 (zéro) en lecture et est ignoré en écriture (poubelle).

Ce processeur utilise des *Exception Level* pourpasser d'un mode d'opération à l'autre lors de l'exécution. Les niveaux de privilèges sont présentés à la figure 2.23.

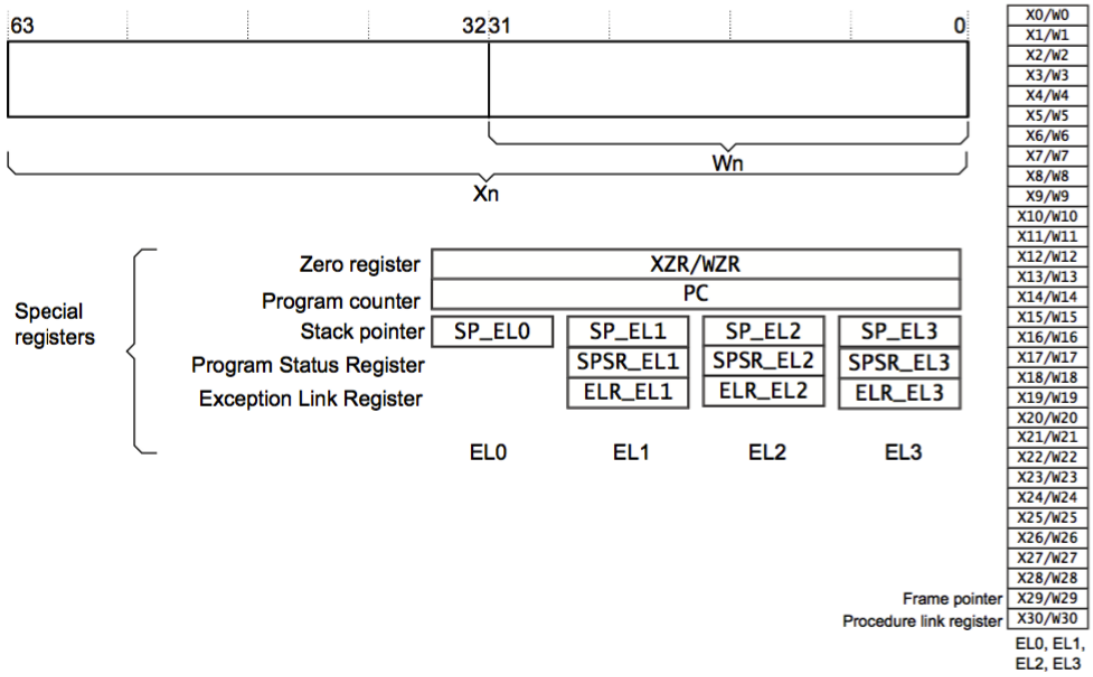


FIGURE 2.22 – Les principaux registres sous ARM64. Tiré de [2].

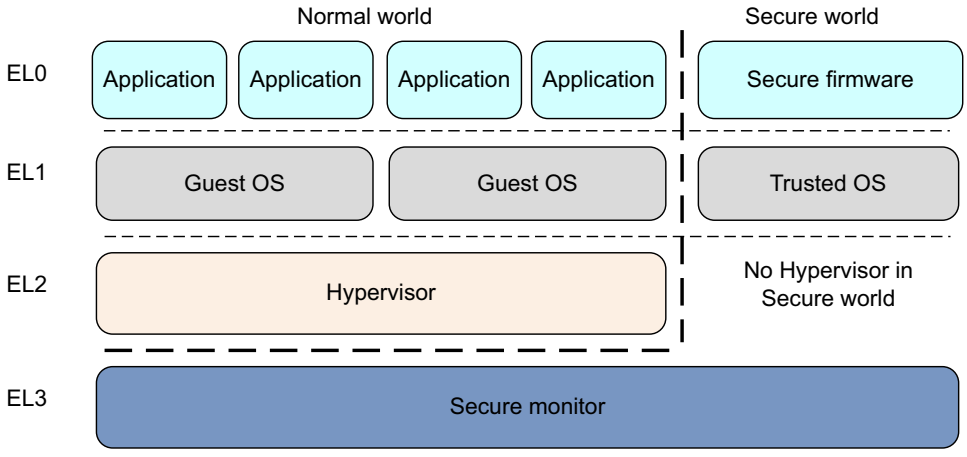


FIGURE 2.23 – Les *Exception Level* sous ARM64. Tiré de [2].

Ici EL0 est de privilège inférieur à EL1, etc.

La mémoire est par défaut petit-bout bien qu'elle peut être configuré gros-bout, alors que les instructions sont toujours encodés petit-bout. La mémoire virtuelle (voir figure 2.24) peut être configurée dans plusieurs modes avec des pages de dimension différentes (voir [2]).

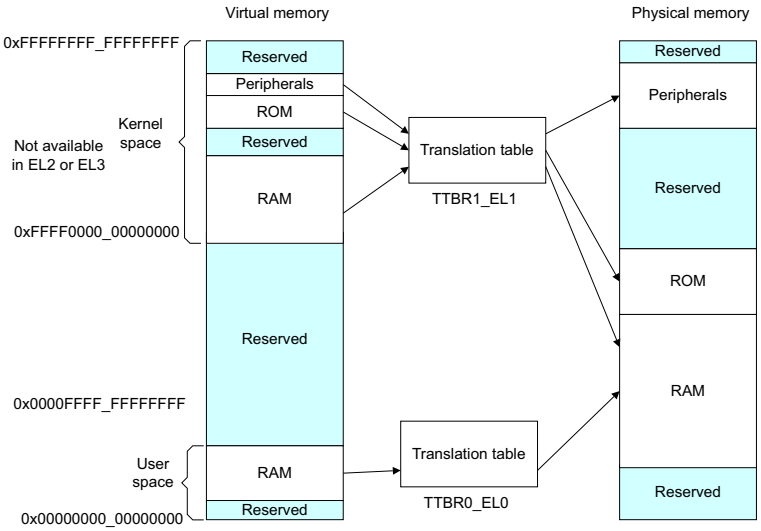


FIGURE 2.24 – Résolution d'adresse virtuelle (*Short-descriptor format*) sous ARM64. Tiré de [2].

Les types de données disponibles sont les mêmes que sous IA-64 à l'exception des BCD (*Binary Coded Decimals*) et des *Far Pointers*.

La pile est de 8 octets de large et part des adresses hautes et croit vers le bas. Son usage est similaire à celle sous Intel bien qu'elle ne soit utilisée qu'occasionnellement pour le passage de paramètres.

Les extensions NEON du processeur permettent le SIMD (32 registres de 64 bits supportant entiers et point-flottant simple précision).

2.6 Les ABI (Application Binary Interface)

Les concepteurs de processeurs suggèrent dans leur documentation des façons de structurer les appels de fonctions, l'alignement des données, l'allocation des variables locales, etc. De manière générale, les compilateurs suivent ces directives

car elles mènent souvent à une utilisation efficiente des ressources matérielles (registres, mémoire cache, pile, ...). Par exemple, on pourra constater de grandes similitudes entre les codes assembleur générés par différents compilateurs sous la même architecture mais sous différents systèmes d'exploitation. Le langage C, étant le langage de programmation des systèmes d'exploitation, permet d'illustrer cela. Nous allons présenter un programme simple pour lequel le code assembleur resultant sera analysé sous différentes « architectures ».

Voici un programe simple qui calcule des factoriels. Ce programme utilise des variables globales, des variables locales et des appels de fonctions.

```
#include <stdio.h>

/* variable globale */
int vecteurglob[8] = {1,2,3,4,5,6,7,8};

int factorial(int n){
  if(n==0){
    return 1;
  }
  return n*factorial(n-1);
}

int main(void){
  /* variables locales */
  int i;
  int vecteurloc[8] = {11,12,13,14,15,16,17,18};

  for(i = 0; i < 8; i++) {
    vecteurloc[i] = factorial(vecteurglob[i]);
    printf("la factoriel de %d est %d\n",vecteurglob[i],vecteurloc[i]);
  }
  return 0;
}
```

En utilisant l'option `-S` de gcc, nous pouvons générer le code assembleur pour ce programme. Nous l'avons fait sous différentes « architectures ». Nous vous présentons les résultats auxquels nous avons ajouté des commentaires pour faire le lien avec le code original.

IA-32

Voici ce que l'on obtient sous Linux-x86 (32 bits) :

```
    .file   "architecture.c"
```

```
        .globl  vecteurglob   # l'etiquette est visible de l'exterieur
        .data                 # segment de donnees read/write
        .align 32
        .type   vecteurglob, @object
        .size   vecteurglob, 32
vecteurglob:                  # variable initialisee vecteurglob[8]
        .long   1
        .long   2
        .long   3
        .long   4
        .long   5
        .long   6
        .long   7
        .long   8
        .text                 # segment de code read/execute
        .globl  factorial
        .type   factorial, @function
factorial:
.LFB0:
        .cfi_startproc
        pushl   %ebp          # sauvegarde du base pointer (ABI)
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp    # le base pointer est mis au top de la pile
        .cfi_def_cfa_register 5
        subl    $24, %esp     # on prend 24 octets sur la pile (espace de travail local)
        cmpl    $0, 8(%ebp)   # l'argument de la fonction (int n) est compare a 0
                              # le top de la pile a l'entree contient le %eip de retour
                              # donc 4 octets, on a fait un pushl un autre 4 octet,
                              # l'argument passe dans la pile est a %ebp + 8
        jne     .L2           # jump si pas 0
        movl    $1, %eax      # !0 = 1 (valeur de retour dans %aex)
        jmp     .L3           # jump pour le return
.L2:
        movl    8(%ebp), %eax # n dans %eax
        subl    $1, %eax      # %eax = n-1
        movl    %eax, (%esp)  # n-1 au top de la pile
        call    factorial     # appel a factorial (argument place au top de la pile
                              # push du %eip de l'instrcution suivante)
        imull   8(%ebp), %eax # n*factorial(n-1) car la valeur de retout est dans %eax
                              # le resultat du calcul est dans %eax
.L3:                          # %eax est pret pour le retour
        leave                 # equivalent de movl %ebp, %esp puis popl %ebp
                              # on retabli la pile et le base pointer
        .cfi_restore 5
```

```
        .cfi_def_cfa 4, 4
        ret                     # retour a l'appelant (le %eip requis pour continuer
                                # l'execution a l'instruction suivant le call est
                                # au top de la pile) un peu comme un popd %eip
        .cfi_endproc
.LFE0:
        .size   factorial, .-factorial
        .section        .rodata # segment de donnees readonly
.LC0:
        .string "la factoriel de %d est %d\n"  # chaine static d'appel a printf
        .text                   # de retour au segment de code read/execute
        .globl  main
        .type   main, @function
main:
.LFB1:
        .cfi_startproc
        pushl   %ebp          # sauvegarde du base pointer (ABI)
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp    # le base pointer est mis au top de la pile
        .cfi_def_cfa_register 5
        andl    $-16, %esp    # alignement du stack frame a modulo 16 (ABI)
        subl    $64, %esp     # variable locales 8x4 + 4 + 3x4 + padding
        movl    $11, 28(%esp) # vecteurloc[0] = 11
        movl    $12, 32(%esp) # vecteurloc[1] = 12
        movl    $13, 36(%esp) # vecteurloc[2] = 13
        movl    $14, 40(%esp) # vecteurloc[3] = 14
        movl    $15, 44(%esp) # vecteurloc[4] = 15
        movl    $16, 48(%esp) # vecteurloc[5] = 16
        movl    $17, 52(%esp) # vecteurloc[6] = 17
        movl    $18, 56(%esp) # vecteurloc[7] = 18
        movl    $0, 60(%esp)  # i = 0
        jmp     .L5
.L6:
        movl    60(%esp), %eax             # %eax = i
        movl    vecteurglob(,%eax,4), %eax # copie de vecteurglob[i] dans %eax
        movl    %eax, (%esp)               # push de vecteurglob[i] sur la pile
        call    factorial                  # appel a factorial
        movl    60(%esp), %edx             # %edx  = i
        movl    %eax, 28(%esp,%edx,4)      # copie de la valeur de retour
                                           # %eax => vecteurloc[i]
        movl    60(%esp), %eax             # %eax = i
        movl    28(%esp,%eax,4), %ecx      # vecteurloc[i] => %ecx
        movl    60(%esp), %eax             # %eax = i
        movl    vecteurglob(,%eax,4), %edx # vecteurglob[i] => %edx
```

```
        movl    $.LC0, %eax               # adresse de la chaine du printf dans %eax
                                          # Les arguments sont places de droite a
                                          # gauche dans la pile
        movl    %ecx, 8(%esp)             # vecteurloc[i]
        movl    %edx, 4(%esp)             # vecteurglob[i]
        movl    %eax, (%esp)              # "la factoriel de %d est %d\n"
        call    printf
          # call printf("la factoriel de %d est %d\n",vecteurglob[i],vecteurloc[i]);
        addl    $1, 60(%esp)              # i++
.L5:
        cmpl    $7, 60(%esp)  # est-ce que la boucle est terminee i<=7
        jle     .L6           # non, on jump
        movl    $0, %eax      # return 0; dans %eax
        leave                 # equivalent de movl %ebp, %esp puis popl %ebp
                              # on retabli la pile et le base pointer
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret                   # retour a l'appelant (le %eip requis pour continuer
                              # l'execution a l'instruction suivant le call est
                              # au top de la pile) un peu comme un popd %ei
        .cfi_endproc
.LFE1:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section        .note.GNU-stack,"",@progbits
```

Voici ce que l'on obtient sous Win32 (32 bits) gcc :

```
        .file   "architecture.c"
        .globl  _vecteurglob  # l'etiquette est visible de l'exterieur
                              # (avec prefix _ ajoute sous Win32)
        .data                 # segment de donnees read/write
        .align 32
_vecteurglob:                 # variable initialisee vecteurglob[8]
        .long   1
        .long   2
        .long   3
        .long   4
        .long   5
        .long   6
        .long   7
        .long   8
        .text                 # segment de code read/execute
        .globl  _factorial
        .def    _factorial;     .scl    2;      .type   32;     .endef
_factorial:
```

```
LFB6:
        .cfi_startproc
        pushl   %ebp         # sauvegarde du base pointer (ABI)
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp   # le base pointer est mis au top de la pile
        .cfi_def_cfa_register 5
        subl    $24, %esp    # on prend 24 octets sur la pile (espace de travail local)
        cmpl    $0, 8(%ebp)  # l'argument de la fonction (int n) est compare a 0
                             # le top de la pile a l'entree contient le %eip de retour
                             # donc 4 octets, on a fait un pushl un autre 4 octet,
                             # l'argument passe dans la pile est a %ebp + 8
        jne     L2           # jump si pas 0
        movl    $1, %eax     # !0 = 1 (valeur de retour dans %aex)
        jmp     L3           # jump pour le return
L2:
        movl    8(%ebp), %eax # n dans %eax
        subl    $1, %eax      # %eax = n-1
        movl    %eax, (%esp)  # n-1 au top de la pile
        call    _factorial    # appel a _factorial (argument place au top de la pile
                              # push du %eip de l'instrcution suivante)
        imull   8(%ebp), %eax # n*factorial(n-1) car la valeur de retout est dans %eax
                              # le resultat du calcul est dans %eax
L3:                           # %eax est pret pour le retour
        leave                 # equivalent de movl %ebp, %esp puis popl %ebp
                              # on retabli la pile et le base pointer
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret                   # retour a l'appelant (le %eip requis pour continuer
                              # l'execution a l'instruction suivant le call est
                              # au top de la pile) un peu comme un popd %eip
        .cfi_endproc
LFE6:
        .def    ___main;        .scl    2;      .type   32;     .endef
        .section .rdata,"dr" # segment de donnees readonly
LC0:
        .ascii "la factoriel de %d est %d\12\0"  # chaine static d'appel a printf
        .text                  # de retour au segment de code read/execute
        .globl  _main
        .def    _main;  .scl    2;      .type   32;     .endef
_main:
LFB7:
        .cfi_startproc
        pushl   %ebp         # sauvegarde du base pointer (ABI)
        .cfi_def_cfa_offset 8
```

```
        .cfi_offset 5, -8
        movl    %esp, %ebp   # le base pointer est mis au top de la pile
        .cfi_def_cfa_register 5
        andl    $-16, %esp    # alignement du stack frame a modulo 16 (ABI)
        subl    $64, %esp     # variable locales 8x4 + 4 + 3x4 + padding
        call    ___main       # appel a _main (initialisation sous Win32)
        movl    $11, 28(%esp) # vecteurloc[0] = 11
        movl    $12, 32(%esp) # vecteurloc[1] = 12
        movl    $13, 36(%esp) # vecteurloc[2] = 13
        movl    $14, 40(%esp) # vecteurloc[3] = 14
        movl    $15, 44(%esp) # vecteurloc[4] = 15
        movl    $16, 48(%esp) # vecteurloc[5] = 16
        movl    $17, 52(%esp) # vecteurloc[6] = 17
        movl    $18, 56(%esp) # vecteurloc[7] = 18
        movl    $0, 60(%esp)  # i = 0
        jmp     L5
L6:
        movl    60(%esp), %eax              # %eax = i
        movl    _vecteurglob(,%eax,4), %eax # copie de vecteurglob[i] dans %eax
        movl    %eax, (%esp)                # push de vecteurglob[i] sur la pile
        call    _factorial                  # appel a factorial
        movl    60(%esp), %edx              # %edx  = i
        movl    %eax, 28(%esp,%edx,4)       # copie de la valeur de retour
                                            # %eax => vecteurloc[i]
        movl    60(%esp), %eax              # %eax = i
        movl    28(%esp,%eax,4), %edx       # vecteurloc[i] => %edx
        movl    60(%esp), %eax              # %eax = i
        movl    _vecteurglob(,%eax,4), %eax # vecteurglob[i] => %eax
                                            # Les arguments sont places de droite a
                                            # gauche dans la pile
        movl    %edx, 8(%esp)               # vecteurloc[i]
        movl    %eax, 4(%esp)               # vecteurglob[i]
        movl    $LC0, (%esp)                # "la factoriel de %d est %d\n"
        call    _printf
          # call printf("la factoriel de %d est %d\n",vecteurglob[i],vecteurloc[i]);
        addl    $1, 60(%esp)                # i++
L5:
        cmpl    $7, 60(%esp)  # est-ce que la boucle est terminee i<=7
        jle     L6            # non, on jump
        movl    $0, %eax      # return 0; dans %eax
        leave                 # equivalent de movl %ebp, %esp puis popl %ebp
                              # on retabli la pile et le base pointer
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret                   # retour a l'appelant (le %eip requis pour continuer
```

```
                                # l'execution a l'instruction suivant le call est
                                # au top de la pile) un peu comme un popd %eip
        .cfi_endproc
LFE7:
        .ident  "GCC: (GNU) 4.8.1"
        .def    _printf;        .scl    2;      .type   32;     .endef
```

Les deux programmes sous IA-32 sont presqu'identiques. On constate que les valeurs de retour sont dans `%eax`. Que la gestion du *stack frame* requiert de préserver `%ebp` et `%esp`. Que `%ecx` et `%edx` sont utilisées sans être préservés. Ceci est conforme à l'ABI. Comme il n'y que 4 registres de disponibles sur 8, il n'est pas surprenant de voir une utilisation « intense » de la pile. Comme le Langage C permet des fonctions variadiques, il faut que le premier argument soit le dernier placé dans la pile, d'où de passage des arguments de droite vers la gauche. Comme il y a peu de registres, la pile est aussi fortement utilisée pour les variables locales (même un compteur entier). Bien sûr, la compilation avec l'option `-O` (optimisation), générera du code très différent et parfois difficile à analyser.

Intel 64

Voici ce que l'on obtient sous Linux-x86-64 (64 bits) :

```
        .file   "architecture.c"
        .globl  vecteurglob   # l'etiquette est visible de l'exterieur
        .data                 # segment de donnees read/write
        .align 32
        .type   vecteurglob, @object
        .size   vecteurglob, 32
vecteurglob:                  # variable initialisee vecteurglob[8]
        .long   1
        .long   2
        .long   3
        .long   4
        .long   5
        .long   6
        .long   7
        .long   8
        .text               # segment de code read/execute
        .globl  factorial
        .type   factorial, @function
factorial:
.LFB0:
        .cfi_startproc
        pushq   %rbp        # sauvegarde du base pointer (ABI)
```

```
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp   # le base pointer est mis au top de la pile
        .cfi_def_cfa_register 6
        subq    $16, %rsp    # on prend 16 octets sur la pile (espace de travail local)
        movl    %edi, -4(%rbp) # on prend l'argument de la fonction => %rsp+12=%rbp-4
        cmpl    $0, -4(%rbp) # l'argument de la fonction (int n) est compare a 0
        jne     .L2          # jump si pas 0
        movl    $1, %eax     # !0 = 1 (valeur de retour dans %aex)
        jmp     .L3          # jump pour le return
.L2:
        movl    -4(%rbp), %eax # n dans %eax
        subl    $1, %eax       # %eax = n-1
        movl    %eax, %edi     # %edi est le premier parametre pour un appel
        call    factorial      # appel a factorial avec argument dans %edi
        imull   -4(%rbp), %eax # n*factorial(n-1) car la valeur de retout est dans %eax
.L3:
        leave                  # equivalent de movq %ebp, %esp puis popq %ebp
                               # on retabli la pile et le base pointer
        .cfi_def_cfa 7, 8
        ret                    # retour a l'appelant (le %eip requis pour continuer
                               # l'execution a l'instruction suivant le call est
                               # au top de la pile)
        .cfi_endproc
.LFE0:
        .size   factorial, .-factorial
        .section        .rodata # segment de donnees readonly
.LC0:
        .string "la factoriel de %d est %d\n" # chaine static d'appel a printf
        .text                  # de retour au segment de code read/execute
        .globl  main
        .type   main, @function
main:
.LFB1:
        .cfi_startproc
        pushq   %rbp         # sauvegarde du base pointer (ABI)
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp   # le base pointer est mis au top de la pile
        .cfi_def_cfa_register 6
        subq    $48, %rsp    # variable locales 8x4 + 4 + padding
        movl    $11, -32(%rbp) # vecteurloc[0] = 11
        movl    $12, -28(%rbp) # vecteurloc[1] = 12
        movl    $13, -24(%rbp) # vecteurloc[2] = 13
        movl    $14, -20(%rbp) # vecteurloc[3] = 14
```

```
        movl    $15, -16(%rbp) # vecteurloc[4] = 15
        movl    $16, -12(%rbp) # vecteurloc[5] = 16
        movl    $17, -8(%rbp)  # vecteurloc[6] = 17
        movl    $18, -4(%rbp)  # vecteurloc[7] = 18
        movl    $0, -36(%rbp)  # i = 0
        jmp     .L5
.L6:
        movl    -36(%rbp), %eax           # %eax = i
        cltq                              # convert long to quad %eax => %rax
        movl    vecteurglob(,%rax,4), %eax # copie de vecteurglob[i] dans %eax
        movl    %eax, %edi                # argument dans %edi
        call    factorial                 # appel a factorial
        movl    -36(%rbp), %edx           # %edx  = i
        movslq  %edx, %rdx                # convert long to quad %edx => %rdx
        movl    %eax, -32(%rbp,%rdx,4)    # copie de la valeur de retour
                                          # %eax => vecteurloc[i]
        movl    -36(%rbp), %eax           # %eax = i
        cltq                              # convert long to quad %eax => %rax
        movl    -32(%rbp,%rax,4), %edx    # vecteurloc[i] => %edx arg3
        movl    -36(%rbp), %eax           # %eax = i
        cltq                              # convert long to quad %eax => %rax
        movl    vecteurglob(,%rax,4), %eax # vecteurglob[i] => %edx
        movl    %eax, %esi                # %esi arg2
        movl    $.LC0, %edi               # %edi pointe sur
                                          # "la factoriel de %d est %d\n" arg1
        movl    $0, %eax                  # nombre d'arguments non-entiers (varargs)
        call    printf
          # call printf("la factoriel de %d est %d\n",vecteurglob[i],vecteurloc[i]);
        addl    $1, -36(%rbp)             # i++
.L5:
        cmpl    $7, -36(%rbp) # est-ce que la boucle est terminee i<=7
        jle     .L6           # non, on jump
        movl    $0, %eax      # return 0; dans %eax
        leave                 # equivalent de movq %ebp, %esp puis popq %ebp
                              # on retabli la pile et le base pointer
        .cfi_def_cfa 7, 8
        ret                   # retour a l'appelant (le %eip requis pour continuer
                              # l'execution a l'instruction suivant le call est
                              # au top de la pile)
        .cfi_endproc
.LFE1:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4"
        .section        .note.GNU-stack,"",@progbits
```

On remarque beaucoup de similarités avec IA-32 au niveau du *stack frame*

mais avec un pile de 8 octets de large. Comme huit registres supplémentaires sont disponibles, les arguments ne sont pas passés sur la pile mais dans les registres dans l'ordre suivant : %rdi, %rsi, %rdx, %rcx, %r8 et %r9. Ici, on ne souffre plus d'une pénurie de registres. Les valeurs de retour sont dans %rax (ou %eax). La fonction appelée doit préserver les registres %rbp, %rbx, and %r12-%r15.

ARMv7-A8

Avant d'analyser le code, il faut prendre connaissance de l'APCS *(ARM Procedure Call Standard)* et des noms alternatifs pour les registres qui y sont présentés et qui peuvent être utilisés dans différentes variantes de l'APCS. La figure 2.25 donnent les registres et les alternatives. Dans la mise en oeuvre présentée ci-dessous, on constate que les regitres r11-r12 sont utilisés sous leur nom alternatif (ip,fp). Les regitres r0-r3 sont utilisés sans être préservés alors que les regitres r4-r8 doivent être préservés. L'APCS stipule aussi que les registres r9-r15 peuvent être à usage particulier (r13-r15 sont toujours sp,lr,pc respectivement).

Le passage des paramètres se fait dans les registres r0-r3. La valeur de retour est dans r0 (et r1 si nécessaire). Les variables locales sont prises sur la pile qui est gérée via sp,fp.

Voici ce que l'on obtient sous Arm Cortex-A8 (32 bits) :

```
        .arch armv7-a
        .eabi_attribute 27, 3
        .eabi_attribute 28, 1
        .fpu neon
        .eabi_attribute 20, 1
        .eabi_attribute 21, 1
        .eabi_attribute 23, 3
        .eabi_attribute 24, 1
        .eabi_attribute 25, 1
        .eabi_attribute 26, 2
        .eabi_attribute 30, 6
        .eabi_attribute 34, 1
        .eabi_attribute 18, 4
        .file   "architecture.c"
        .global vecteurglob   @ l'etiquette est visible de l'exterieur
        .data                 @ segment de donnees read/write
        .align  2
        .type   vecteurglob, %object
        .size   vecteurglob, 32
vecteurglob:                  @ variable initialisee vecteurglob[8]
        .word   1
```

Register	APCS name	APCS role
r0	a1	argument 1/scratch register/result
r1	a2	argument 2/scratch register/result
r2	a3	argument 3/scratch register/result
r3	a4	argument 4/scratch register/result
r4	v1	register variable
r5	v2	register variable
r6	v3	register variable
r7	v4	register variable
r8	v5	register variable
r9	sb/v6	static base/register variable
r10	sl/v7	stack limit/stack chunk handle/register variable
r11	fp/v8	frame pointer/register variable
r12	ip	scratch register/new-sb in inter-link-unit calls
r13	sp	lower end of the current stack frame
r14	lr	link register/scratch register
r15	pc	program counter

FIGURE 2.25 – APCS de ARM. Tiré de [1].

```
        .word   2
        .word   3
        .word   4
        .word   5
        .word   6
        .word   7
        .word   8
        .text                @ segment de code read/execute
        .align  2
        .global factorial
        .type   factorial, %function
factorial:
        @ args = 0, pretend = 0, frame = 8
        @ frame_needed = 1, uses_anonymous_args = 0
        stmfd   sp!, {fp, lr}  @ sauvegarde de fp et lr sur la pile => push {fp, lr}
        add     fp, sp, #4     @ sauvegarde de sp
        sub     sp, sp, #8     @ 8 octets sur la pile
        str     r0, [fp, #-8]  @ on place n dans la pile
        ldr     r3, [fp, #-8]  @ on place n dans r3
        cmp     r3, #0         @ n==0 ?
        bne     .L2            @ jump si pas 0
        mov     r3, #1         @ r3 = !0 = 1
        b       .L3            @ jump pour le return
.L2:
        ldr     r3, [fp, #-8]  @ r3 = n
        sub     r3, r3, #1     @ r3 = n-1
        mov     r0, r3         @ r0 = n-1
        bl      factorial      @ appel a factorial argument dans r0
        mov     r2, r0         @ valeur de retour r0 => r2
        ldr     r3, [fp, #-8]  @ r3 = n (rappel r3 non preserve sur "function call")
        mul     r3, r3, r2     @ r3 = n*factorial(n-1)
.L3:
        mov     r0, r3         @ r3 => r0 pour la valeur de retour
        sub     sp, fp, #4     @ restore de sp
        @ sp needed
        ldmfd   sp!, {fp, pc}  @ restore de fp et (lr->pc) => pop {fp, pc}, return
        .size   factorial, .-factorial
        .section        .rodata @ segment de donnees readonly
        .align  2
.LC1:
        .ascii  "la factoriel de %d est %d\012\000"  @chaine static d'appel a printf
        .align  2
.LC0:               @ valeurs pour initialiser la variable locale vecteurloc[8]
        .word   11
        .word   12
```

```
        .word   13
        .word   14
        .word   15
        .word   16
        .word   17
        .word   18
        .text                   @ de retour au segment de code read/execute
        .align  2
        .global main
        .type   main, %function
main:
        @ args = 0, pretend = 0, frame = 40
        @ frame_needed = 1, uses_anonymous_args = 0
        stmfd   sp!, {fp, lr}  @ sauvegarde de fp et lr sur la pile => push {fp, lr}
        add     fp, sp, #4     @ sauvegarde de sp
        sub     sp, sp, #40    @ 40 octets sur la pile
        movw    r3, #:lower16:.LC0 @ chargement de l'adresse 32 bits en deux
        movt    r3, #:upper16:.LC0 @ blocs de 16 bits
        sub     ip, fp, #40    @ ip pointe sur le debut de vecteurloc
        mov     lr, r3         @ lr pointe a .LC0
        ldmia   lr!, {r0, r1, r2, r3} @ copie de quatre valeurs dans r0-r3
                                @ puis incremente lr
        stmia   ip!, {r0, r1, r2, r3} @ copie r0-r3 vers memoire a ip (&vecteurloc[0])
                                @ puis incremente ip
        ldmia   lr, {r0, r1, r2, r3}  @ copie des quatre valeurs suivantes dans r0-r3
        stmia   ip, {r0, r1, r2, r3}  @ copie r0-r3 vers memoire a ip (&vecteurloc[4])
        mov     r3, #0                @ r3 = 0
        str     r3, [fp, #-8]         @ i = 0
        b       .L5
.L6:
        movw    r3, #:lower16:vecteurglob @ adresse 32 bits de vecteurglob dans r3
        movt    r3, #:upper16:vecteurglob
        ldr     r2, [fp, #-8]         @ r2 = i
        ldr     r3, [r3, r2, asl #2]  @ r3 = vecteurglob[i]  r3+(r2*4)
        mov     r0, r3                @ r0 = vecteurglob[i]
        bl      factorial             @ call a factorial(vecteurglob[i])
        mov     r2, r0                @ r2 = valeur de retour
        ldr     r3, [fp, #-8]         @ r3 = i
        mov     r3, r3, asl #2        @ r3*4
        sub     r1, fp, #4            @ r1 = &vecteurloc[9]
        add     r3, r1, r3            @ r3 = &vecteurloc[9+i]
        str     r2, [r3, #-36]        @ r2 => vecteurloc[9+i-9] = vecteurloc[i]
        movw    r3, #:lower16:vecteurglob @ r3 adresse 32 bits vecteurglob[0]
        movt    r3, #:upper16:vecteurglob @ en deux blocs 16 bots
        ldr     r2, [fp, #-8]         @ r2 = i
```

```
        ldr     r2, [r3, r2, asl #2]  @ r2 = vecteurglob[i]  (r2*4)
        ldr     r3, [fp, #-8]         @ r3 = i
        mov     r3, r3, asl #2        @ r3*4
        sub     r1, fp, #4            @ r1 = &vecteurloc[9]
        add     r3, r1, r3            @ r3= &vecteurloc[9+i]
        ldr     r3, [r3, #-36]        @ r3 => vecteurloc[9+i-9] = vecteurloc[i]
        movw    r0, #:lower16:.LC1    @ adresse 32 bits de
        movt    r0, #:upper16:.LC1    @ "la factoriel de %d est %d\n" dans r0
        mov     r1, r2                @ r1 = vecteurglob[i]
        mov     r2, r3                @ r2 = vecteurloc[i]
        bl      printf
          @ call printf("la factoriel de %d est %d\n",vecteurglob[i],vecteurloc[i]);
        ldr     r3, [fp, #-8]         @ r3 = i
        add     r3, r3, #1            @ r3++
        str     r3, [fp, #-8]         @ r3 => i (i++)
.L5:
        ldr     r3, [fp, #-8]         @ r3 = i
        cmp     r3, #7                @ i = 7 ?
        ble     .L6                   @ i <= 7, on continue
        mov     r3, #0                @ r3 = 0
        mov     r0, r3                @ r3 => r0, return 0
        sub     sp, fp, #4            @ restore de sp
        @ sp needed
        ldmfd   sp!, {fp, pc}  @ restore de fp et (lr->pc) => pop {fp, pc}, return
        .size   main, .-main
        .ident  "GCC: (GNU) 4.9.2"
        .section        .note.GNU-stack,"",%progbits
```

ARMv8-A

La figure 2.26 illustre l'usage des registres de l'APCS64. Le registre X8 permet de retourner des valeurs (`struct`) qui ne peuvent être placées dans les registres X0 à X7. Les variables locales sont prises sur la pile qui est gérée via sp,fp.

Voici ce que l'on obtient sous Arm Cortex-A53 (64 bits) :

```
        .cpu generic+fp+simd
        .file   "architecture.c"
        .global vecteurglob
        .data
        .align  3
        .type   vecteurglob, %object
        .size   vecteurglob, 32
vecteurglob:                 @ variable initialisee vecteurglob[8]
        .word   1
        .word   2
```

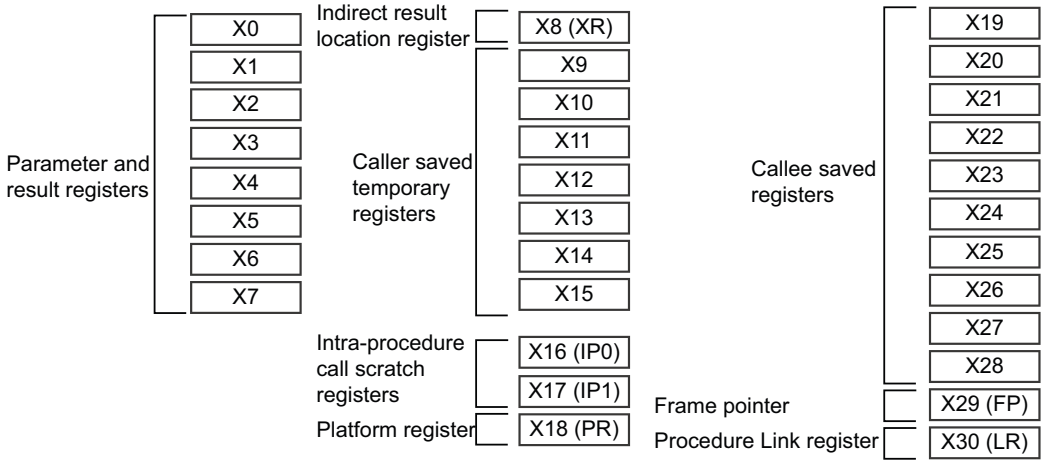


FIGURE 2.26 – Registres de l'ABI de ARM64. Tiré de [1].

```
        .word   3
        .word   4
        .word   5
        .word   6
        .word   7
        .word   8
        .text                   @ segment de code read/execute
        .align  2
        .global factorial
        .type   factorial, %function
factorial:
        stp     x29, x30, [sp, -32]! @ sauvegarde de fp et lr sur la pile et
                                     @ variables locales (16 bytes aligned)
        add     x29, sp, 0           @ sauvegarde de sp
        str     w0, [x29, 28]        @ on place n dans la pile
        ldr     w0, [x29, 28]        @ on place n dans w0
        cmp     w0, wzr              @ n==0 ?
        bne     .L2                  @ jump si pas 0
        mov     w0, 1                @ w0 = !0 = 1
        b       .L3                  @ jump pour le return
.L2:
        ldr     w0, [x29, 28]        @ w0 = n
        sub     w0, w0, #1           @ w0 = n-1
        bl      factorial            @ appel a factorial argument dans w0
        mov     w1, w0               @ valeur de retour w0 => w1
```

```
        ldr     w0, [x29, 28]        @ w0 = n (w0 non preserve sur "function call")
        mul     w0, w1, w0           @ w0 = n*factorial(n-1)
.L3:
        ldp     x29, x30, [sp], 32   @ restore de fp et lr et ajuste sp (pile)
        ret                          @ retour a l'appelant, w0 valeur de retour
        .size   factorial, .-factorial
        .section        .rodata      @ segment de donnees readonly
        .align  3
.LC1:
        .string "la factoriel de %d est %d\n"   @ chaine static d'appel a printf
        .text
        .align  2
        .global main
        .type   main, %function
main:
        stp     x29, x30, [sp, -64]! @ sauvegarde de fp et lr sur la pile et
                                     @ variables locales (16 bytes aligned)
        add     x29, sp, 0           @ sauvegarde de sp
        adrp    x0, .LC0             @ page de l'adresse des valeurs d'initialisation
        add     x0, x0, :lo12:.LC0   @ ajoute les 12bits pour completer l'adresse
        add     x2, x29, 24          @ x2 pointe sur la pile a &vecteurloc[0]
        mov     x3, x0               @ x3 pointe vers les valeurs d'initialisation
        ldp     x0, x1, [x3]         @ copie vecteurloc[0] et vecteurloc[1] dans x0, x1
        stp     x0, x1, [x2]         @ ecrit x0, x1 dans vecteurloc[0] et vecteurloc[1]
        ldp     x0, x1, [x3, 16]     @ copie vecteurloc[2] et vecteurloc[3] dans x0, x1
        stp     x0, x1, [x2, 16]     @ ecrit x0, x1 dans vecteurloc[2] et vecteurloc[3]
        str     wzr, [x29, 60]       @ i=0
        b       .L5
.L6:
        adrp    x0, vecteurglob           @ page de l'adresse vecteurglob
        add     x0, x0, :lo12:vecteurglob @ ajoute les 12bits pour completer l'adresse
        ldrsw   x1, [x29, 60]        @ x1 = i
        ldr     w0, [x0, x1, lsl 2]  @ w0 = vecteurglob[i]  x0+(x1*4)
        bl      factorial            @ call a factorial(vecteurglob[i])
        mov     w1, w0               @ w1 = valeur de retour
        ldrsw   x0, [x29, 60]        @ x0 = i
        lsl     x0, x0, 2            @ x0*4
        add     x2, x29, 64          @ x2 = &vecteurloc[10]
        add     x0, x2, x0           @ x0 = &vecteurloc[10+i]
        sub     x0, x0, #4096
        str     w1, [x0, 4056]       @ w1 = &vecteurloc[10+i-10]
        adrp    x0, vecteurglob           @ page de l'adresse vecteurglob
        add     x0, x0, :lo12:vecteurglob @ ajoute les 12bits pour completer l'adresse
        ldrsw   x1, [x29, 60]        @ x1 = i
        ldr     w1, [x0, x1, lsl 2]  @ w1 = vecteurglob[i]  x1+(x1*4)
```

```
        ldrsw   x0, [x29, 60]        @ x0 = i
        lsl     x0, x0, 2            @ x0*4
        add     x2, x29, 64          @ x2 = &vecteurloc[10]
        add     x0, x2, x0           @ x0 = &vecteurloc[10+i]
        sub     x0, x0, #4096
        ldr     w2, [x0, 4056]       @ w2 = &vecteurloc[10+i-10]
        adrp    x0, .LC1             @ page de l'adresse de "la factoriel de %d est %d\n"
        add     x0, x0, :lo12:.LC1   @ ajoute les 12bits pour completer l'adresse
        bl      printf
          @ call printf("la factoriel de %d est %d\n",vecteurglob[i],vecteurloc[i]);
        ldr     w0, [x29, 60]        @ w0 = i
        add     w0, w0, 1            @ w0++
        str     w0, [x29, 60]        @ w0 => i (i++)
.L5:
        ldr     w0, [x29, 60]        @ w0 = i
        cmp     w0, 7                @ i = 7 ?
        ble     .L6                  @ i <= 7, on continue
        mov     w0, 0                @ w0 = 0
        ldp     x29, x30, [sp], 64   @ restore de fp et lr et ajuste sp (pile)
        ret                          @ return(0) w0
        .size   main, .-main
        .section        .rodata
        .align  3
.LC0:                @ valeurs pour initialiser la variable locale vecteurloc[8]
        .word   11
        .word   12
        .word   13
        .word   14
        .word   15
        .word   16
        .word   17
        .word   18
        .text
        .ident  "GCC: (GNU) 5.2.0"
        .section        .note.GNU-stack,"",%progbits
```

2.7 Mémoire, stockage et I/O

Afin d'utiliser au mieux les ressources d'un système, différents éléments de « mémoire » sont utilisés. Les registres du processeur sont certainement les éléments les plus rapides pour ce qui est du temps d'accès. Vient ensuite la mémoire cache (il peut y avoir deux niveaux de cache : une à même le processeur et une autre externe mais à accès plus rapide que la mémoire standard) qui est un miroir

de la mémoire RAM qui, elle, est au niveau suivant. La mémoire cache est souvent divisée en deux « blocs » : un pour le code et un autre pour les données ; permettant ainsi un accès rapide et simultané au code (le *fetch*) et au *data*. L'espace mémoire *RAM* est partitionné en regions. Certaines sont attribuées à des périphériques permettant ainsi des entrées/sorties par des écriture en mémoire. Cela explique que l'espace adressable n'est pas toujours entièrement disponible pour y associé de la mémoire RAM (limite maximale sur la RAM). Sous certaines architectures (IA-32 par exemple) des ports existent avec des adresses accessibles via des instructions `inport` et `output` permettant aussi des accès a des périphériques.

Comme la cache du processeur peut être utilisée pour accélérer les opérations par rapport à la mémoire RAM, une partie de la mémoire RAM peut être utilisée comme cache pour les unités de stockages (disques dur, ...). Si une portion de fichier est lue fréquemment, elle sera fort probablement dans la cache de disque du système d'exploitation en mémoire RAM. En écriture, les opérations seront regroupées afin d'écrire des « blocs » entiers de la façon la plus efficace possible (d'où les fonctionnalités *eject media*, *file flush*, ...). Ce sont ces stratégies de gestion de ces mémoires caches à différents niveaux qui permettent d'aller chercher des gains de performance très intéressants par rapport à un même matériel sans cache. Notons que pour le programmeur d'applications, ceci est transparent. De plus, les entrées/sorties sont gérées par des appels système dans les systèmes d'exploitation.

Chapitre 3

Systèmes d'exploitation

Le choix d'un système d'exploitation pour le développement d'un système va dépendre fortement du contexte d'application. Pour mieux comprendre comment faire ce choix certaines notions relatives aux systèmes d'exploitation doivent être couvertes (voir [5] pour plus de détails).

3.1 Éléments d'un système d'exploitation

Un système d'exploitation est un logiciel spécialisé qui introduit une couche d'abstraction entre le matériel et l'usager (le logiciel) afin de gérer efficacement l'exécution des programmes [1]. Il contrôle l'allocation des ressources et s'assure d'éviter les interférences entres les applications lors de l'accès à des ressources matérielles communes. Les principales fonctions d'un système d'exploitation sont

La gestion de la mémoire : La mémoire physique d'un ordinateur doit être partagée entre les différents programmes (incluant le système d'exploitation). Il faut donc faire le suivi de l'allocation des différents blocs alloués aux programmes en cours d'exécution, de répondre au demande d'allocation faites par les programmes et de récupérer ces blocs de mémoire lors de la fin d'exécution des programmes. Dans les systèmes avec protection mémoire, il faut s'assurer de compartimenter les applications pour la stabilité et la sécurité du système.

La gestion des processus : Dans un environnement multi-programmes/multi-usagers, le système d'exploitation gère les tranches de temps du(des) pro-

1. Un recherche avec les mots clés :*os architecture* vous permettra de voir comment différents systèmes d'exploitation mettent en oeuvre les différentes couches.

cesseur(s) qui sont allouées aux différents processus en cours d'exécution. Diffrentes stratégies d'allocation (l'ordonnanceur) peuvent être mises en oeuvre.

La gestion des périphériques : Les différents périphériques présents au démarrage doivent être initialisés au démarrage via leur pilotes respectifs et doivent être rendus disponibles au travers d'une interface usager. Le système doit aussi pouvoir allouer/libérer les ressources nécessaires pour un périphérique qui est ajouté/retiré en cours d'opération.

La gestion des systèmes de fichiers : Le système de fichiers, sa hiérarchie (i.e. la structure de sous-répertoires), les droits d'accès, la gestions des accès simultanés doivent être présentés de façon uniforme sans égard au matériel de support (on parle dans certains cas de VFS — *virtual file system*, une couche d'abstraction entre les unités physiques et les programmes).

La gestion des usagers et de la sécurité : L'authentification, la gestion des droits d'accès aux fichiers et périphériques, la gestion des accès réseau, les quotas de ressources disponibles, entre autres, doivent être gérés de façon efficace et sécuritaire par le système d'exploitation.

3.2 La gestion des processus

Dans un système multi-programmes/multi-usagers, il faut mettre en oeuvre une stratégie de répartition des ressources dans le temps. Imaginons un programme qui entre dans dans un boucle infinie (suite à une erreur de programmation), le système d'exploitation doit permettre aux autres processus d'obtenir du temps CPU, sinon, le système deviendra non-fonctionnel (le système est « gelé »). Une des techniques utilisées pour ce faire est la préemption. On associe une interruption à un *timer* (par exemple à toutes le 20ms comme sous Windows). L'ordonnanceur qui gère les processus lance l'exécution du programme, après le délai, si le programme est encore en exécution, le *timer* génère l'interruption et le programme est interrompu et le système d'exploitation reprend le contrôle. L'ordonnanceur vérifie l'état des tâches (processus) en attentes et en fonction de la stratégie d'ordonnancement donnera l'exécution à un nouveau processus ou relancera le processus interrompu. Différents types d'ordonnancement sont mis en oeuvre en focntion des besoins du système : serveur, desktop, système temps-réel, etc.

En plus de l'échéance du *timer*, l'ordonnanceur peut reprendre le contrôle lors d'un appel système mettant le processus en attente. Par exemple, pour un

programme en attente d'une entrée au clavier, comme les humains sont très lents par rapport à l'échelle de temps du système, un autre processus sera exécuté en attendant le retour de l'appel système.

Si plusieurs programmes sont en exécution, ils pourront être placés dans une file d'attente de type *first-in/first-out* pour être exécuté chacun leur tour. On peut aussi leur associer un état (*Ready to run*, *Waiting for I/O*, ...) et un niveau de priorité afin de mettre en oeuvre une stratégie avec plus d'une file d'attente. Si l'intervalle de temps est assez rapide entre les préemptions, l'usager aura l'impression que les processus s'exécutent de façon « simultanée ».

Chaque processus possède son contexte d'exécution : état des registres, tables de gestion de la mémoire virtuelle, etc. Lors de la préemption, la routine d'interruption doit sauvegarder ce contexte avant de relancer l'exécution d'un autre processus pour lequel il faudra restaurer le contexte d'exécution. Ces changements de contexte prennent un certain temps en plus des changements de mode d'exécution (*user->kernel->user*), c'est pour cela qu'il ne faut pas choisir l'intervalle de préemption trop court sinon l'ordinateur va passer une trop grande partie du temps en changement de contexte.

Comme plusieurs processus peuvent être exécutés, le système système d'exploitation doit maintenir une liste de structures de données contenant l'information pertinente à chacun des processus : l'état (*Ready*, *Waiting*, *Running*), privilèges, identificateur (*PID*), pointeur vers le processus parent, pointeur d'instruction (*Program Counter*), les registres du CPU, information d'ordonnancement (priorité, ...), les tables de pages pour l'adressage de la mémoire virtuelle, etc.

3.3 La gestion de la mémoire

Lors du chargement d'un programme, le code et les données sont chargés en mémoire et le processus est démarré. L'image du programme en mémoire ressemble à la figure 3.1 où

Text : contient les instructions du programme (le code) ;

Data : contient les variables globales et statiques ;

Heap : contient la mémoire allouée dynamiquement :

Stack : la pile qui contient les variables locales, les paramètres d'entrée des fonctions ou méthodes et les adresses de retour des appels de fonctions.

Comme plusieurs processus sont exécutés à tour de rôle, il faut un mécanisme pour le partage de la mémoire pendant leur exécution. De plus, le noyau (*kernel*)

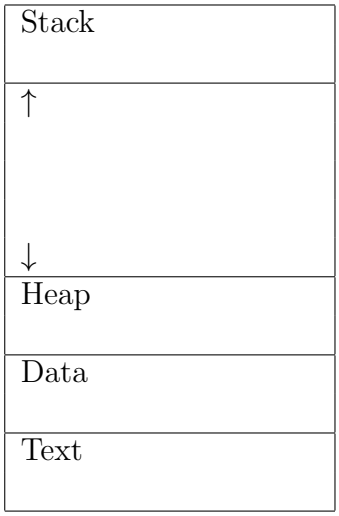


FIGURE 3.1 – Image mémoire d'un processus

du système d'exploitation doit avoir son propre espace mémoire. Une façon de faire, si l'architecture le permet, est d'utiliser des adresses logiques pour l'espace mémoire des programmes. Le système de gestion de mémoire du processeur (*memory management unit, MMU*) va transformer ces adresses logiques en adresses physiques. Ainsi, deux processus pourront avoir les mêmes adresses logiques pour les différentes sections (*Text, Data, Heap, Stack*) mais seront à des adresses physiques distinctes en mémoire.

L'espace mémoire adressable par le processeur est séparé en deux partitions : l'espace du noyau (*kernel space*) qui utilise la mémoire physique basse et l'espace mémoire usager pour les programmes (*user space*) qui utilise la mémoire physique haute. Dans le cas d'un système avec mémoire protégée, les processus usager n'ont pas accès à l'espace mémoire du noyau, ni aux espaces mémoires des autres processus. L'espace mémoire physique est divisé en blocs (*Pages*) qui seront assignés aux différents processus en exécution. Si la mémoire physique devient complètement utilisée, certains des blocs du *user space* seront écrits sur disque dans un espace tampon (*Swap File or Partition*) et pourront être rechargés par la suite lorsque certains programmes auront terminé leur exécution ou seront en état d'attente (*Waiting*). C'est le principe de la mémoire virtuelle (si vous êtes curieux, vous pouvez consulter les détails).

3.4 La gestion des périphériques

Pour un système multi-tâches, la gestion des périphériques par le système d'exploitation permet d'éviter les conflits d'accès aux ressources. Par exemple, un système avec plusieurs consoles (*Terminals*) ouvertes doit pouvoir gérer les entrées clavier et les sorties écran en fonction des fenêtres actives. Il en est de même pour toutes les ressources partagées.

Il existe trois approches pour la communication avec les périphériques :

Instructions spéciales : des instructions spéciales du processeur permettent d'interagir avec le périphérique. Un exemple, les instructions `inport` et `outport` sous l'architecture Intel.

Memory-mapped : il y a un espace mémoire commun entre le CPU et le périphérique. Il y a communication entre le CPU et le périphérique pour le transfert des données coordonné au moyen d'interruptions.

Direct Memory Access (DMA) : le périphérique a un accès direct à la mémoire via un contrôleur pour les transferts et il y a interruptions seulement au début et à la fin de transfert de blocs complets de données.

Quelque soit le type de communication, les périphériques sont gérés par des pilotes d'interface (*Device Drivers*) qui s'occupent de les présenter de façon uniforme au moyen d'appels système. Ces pilotes sont en charge de répondre rapidement aux interruptions matérielles des périphériques tout en évitant de monopoliser trop de ressources CPU.

3.5 La gestion des systèmes de fichiers

Le système d'exploitation doit présenter aux processus usager les différents systèmes de fichiers et leur support de façon unifirme et transparente. Au plus bas niveau, il y a un pilote d'interface qui peut lire et écrire des blocs de données sur l'unité de stockage (*Block Device Driver*). Le pilote peut interroger l'unité pour connaitre le nombre et la taille des blocs. Les premiers blocs de données contiennent des méta-données (*bootrecords* et tables de partitions : leurs types, positions de début et de fin). Chacune des partitions débute par des méta-données (*superblocks*) donnant le type de système de fichier présent (*fat32*, *NTFS*, *ext4*, etc). Le système d'exploitation au second niveau possède des pilotes qui interfacent ces types de systèmes de fichiers afin de les présenter de façon uniforme la hiérarchie : répertoire racine, sous-répertoires, fichiers, etc ; et permettent les

opérations de base : ouverture, lecture, écriture, fermeture, etc. Selon le système de fichiers, des droits d'accès fonction des usagers pourront être disponibles pour valider et autoriser les opérations.

3.6 La gestion des usagers et de la sécurité

Finalement, le système d'exploitation doit aussi prendre la gestion des usagers et la sécurité. Cela comprend l'authentification des usagers par différentes méthodes : usager/mot de passe, carte à puce/clé, caractéristique individuelle (empreinte digitale, retine, signature). Une fois l'usager authentifié, il faut limiter les accès disques et réseaux en fonction de ses droits d'accès.

De plus, il faut assurer la sécurité du système vis-à-vis les programmes malveillants que l'usager peut exécuter par mégarde ou intentionnellement.

Si l'ordinateur est en réseaux, il faut aussi que le système d'exploitation soit robuste face à des attaques extérieures (*firewall*).

Pour plus de détails concernant les points précédents, vous pouvez vous référons au tutoriel de Tutorials Point sur les systèmes d'exploitation (pdf).

3.7 Les appels système

Comme le noyau du système d'exploitation met en oeuvre une couche d'abstraction entre le matériel et les programmes, une procédure est mise en place pour l'accès à ces ressources : les appels système. Les compilateurs en langage C sous la très grande majorité des plateformes font appel à une libraire C qui contient les instructions nécessaires au passage de paramètres lors d'un appel système. Par exemple, l'ouverture d'un fichier en lecture nécessite de passer en arguments le nom du fichier et le mode d'ouverture (ici en lecture) et en cas de succès, on aura au retour un descripteur de fichier (un entier à valeur positive en cas de succès, un entier négatif en cas d'erreur). Tous les compilateurs utilisent « leur » librairie C par défaut lors de la génération d'un exécutable. Ces appels sont réalisés par des instructions en assembleur qui permettent de passer du monde *user* au mode superviseur.

3.8 Rôle de l'assembleur

Comme nous avons vu au chapitre 2, un programme en langage C utilise les registres généraux, la pile et la mémoire. Cependant, il n'existe pas d'instruction dans ce langage pour émettre des directives qui utilisent les registres de contrôle ou des instructions du mode superviseur. Il faut donc utiliser l'assembleur dans la mise en oeuvre. Ceci peut être fait de deux façons :

— par un programme écrit directement en assembleur. Par exemple sous Linux, le point d'entrée d'un appel système est traité par le fichier `entry.S` sous plateforme x86 et par le fichier `entry-common.S` sous ARM.

— par de l'assembleur en-ligne et des macros inclus à même un programme en langage C.

Dans la conception, les macros servent à fournir un appel similaire qui dépend le moins possible de l'architecture. Par exemple, le fichier `syscalls.h` est le même quelque soit l'architecture mais le fichier `syscall.h` dépend de l'architecture (ARM ou x86 par exemple).

3.9 Choix d'un système d'exploitation

3.9.1 Contraintes temporelles

Les contraintes temporelles imposées par l'application auront une influence importante sur le choix. Certaines applications demandent seulement d'être réactives (temps de réponse court, interface « rapide », ...). Pour d'autres, des tâches devront être réalisées à intervalles réguliers pour lesquelles deux cas sont possibles. La tâche est effectuée selon une fréquence prescrite mais un léger retard ou le saut d'un échantillon de façon occasionnelle n'entraine pas de conséquences importantes sur la bonne marche du système — *soft real-time* (exemples : mesure de la température ambiante par un thermostat qui communique avec un système central de chauffage-climatisation industriel, affichage de données graphiques pour un instrument de mesure, encodage d'un flux vidéo pour diffusion, ...). La tâche doit être effectuée selon une fréquence précise et tout retard ou saut d'échantillon entraine des problèmes importants (ou catastrophiques) dans le système — *hard real-time* (exemples : commande de vol en boucle-fermée d'un aéronef, module de contrôle de traction dans une voiture, traitement audio numérique pour performance *live*, durée d'une « dose » d'un traitement de radio-thérapie, ...).

3.9.2 L'empreinte mémoire

La quantité de mémoire requise pour l'exécution et pour le stockage de l'image du système sont des facteurs importants. Le coût supplémentaire de quelques Mo de mémoire pour système vendu à des millons d'exemplaires n'est pas négligeable. La possibilité de configurer une image minimale ne contenant que les fonctionnalités désirées est un facteur important si l'on veux pouvoir contrôler les requis en mémoire vive et espace de stockage. Par exemple, sous Linux, on peut configurer notre système pour utiliser GLibC qui est très complète mais plus volumineuse que uCLibC qui a une empreinte mémoire plus petite et qui peut-être configurée pour retirer certaines fonctionnalités afin de réduire encore plus la mémoire requise (voir GLibC vs uCLibC).

3.9.3 Les services requis

En fonction de l'application à développer, le système d'exploitation devra pouvoir offrir certains services : communication réseau, affichage graphique, système de fichiers locaux ou distants, interfaces sonores, *timers*, tâches multiples, l'ordonnancement des tâches, la signalisation, les interruptions, la synchronisation, les exclusions mutuelles, ... Le tout dans le respect de certains standards (POSIX, TCPIP, ...).

3.9.4 Disponibilité des pilotes d'interface matérielle

Le support des périphériques requis pour le système où la possibilité de développer un pilote d'interface pour le système d'exploitation sont aussi des critères importants lors de la sélection.

3.9.5 Choix d'ordonnancement

Certains systèmes permettent de choisir ou de configurer l'ordonnancement des tâches avec plus ou moins de flexibilité. En fonction des contraintes temporelles et du type de tâches à effectuer, cet aspect peut être un critère de premier plan dans le choix d'un système d'exploitation. Par exemple, un serveur web à haut volume de transactions aura un ordonnanceur configuré différemment d'un Desktop Graphique.

3.9.6 Modèle de protection mémoire

Si l'architecture retenue possède une MMU : *Memory Management Unit* (ou une MPU : *Memory Protection Unit*), le modèle de gestion mémoire choisi aura un impact sur la sécurité du système et sa robustesse. Par exemple, sous QNX (un système temps-réel), on a un micro-kernel au plus haut niveau de protection, les pilotes d'interfaces sont au niveau suivant, suivi des applications. Le noyau est donc protégé d'une mauvaise mise en oeuvre d'un pilote d'interface, ce qui n'est pas le cas pour Linux par exemple. En l'absence de MMU, certains systèmes d'exploitations seront alors de mise : FreeRTOS ou Linux MMU-less kernel par exemple. Une MPU peut ajouter de la robustesse mais ne permet pas d'adressage virtuel.

3.10 Systèmes d'exploitation les plus utilisés

Dans cette section, nous allons décrire brièvement certains systèmes d'exploitation les plus courants.

3.10.1 Systèmes de type Unix

Les systèmes dérivés ou inspirés de Unix sont nombreux et sont majoritairement caractérisés par un noyau monolithique. La majorité de ces variantes cherche à maintenir une compliance avec le standard POSIX, s'assurant ainsi d'une portabilité accrue des applications entres les différentes variantes.

Les dérivés BSD : Il existe plusieurs descendants du Berkeley Software Distribution :
- FreeBSD et ses dérivés (DragonFlyBSD, Darwin (OS X, iOS) de Apple, ...)
- NetBSD (OpenBSD)

Linux et dérivés :
- Linux toutes distributions
- Android et dérivés
- Kindle Fire OS

Les Unix : les systèmes, commerciaux, qui sont dérivés du code de base du Unix de la compagnie AT&T.

3.10.2 Systèmes de la famille Windows

Bien sûr, il y a l'incontournable logiciel propriétaire de Microsoft. Le système d'exploitation aux multiples variantes, allant des versions serveurs jusqu'à l'embarqué Windows CE.

3.10.3 Systèmes pour le temps réel

Il a aussi les systèmes spécialisés pour le temps réel (qui sont souvent des systèmes embarqués). En voici quelques uns des plus utilisés parmi le grand nombre disponible :

FreeRTOS : empreinte mémoire très petite, une librairie pour les *threads* à laquelle on peut ajouter une interface ligne de commande et d'entrées/sorties de type POSIX.

QNX : un micro-noyau avec un système de messages, des tâches serveurs et des pilotes d'interfaces hors noyau.

Real Time Linux : via une configuration spécifique du noyau qui permet la préemption de l'exécution du noyau avec des ordonnanceurs spécifiques au temps réel.

VxWorks : conçu pour les systèmes embarqués. C'est le un peu le *top end* des systèmes temps réel avec ce que cela implique ($$).

Windows Embedded Compact : avec un noyau hybride avec des composantes *open source* (plateformes spécifiques) et des composants binaires seulement (composantes dépendantes du CPU seulement), très faible empreinte mémoire.

Chapitre 4

Introduction à Linux

Maintenant que nous avons fait un survol des caractéristiques générales d'un système d'exploitation, nous allons décrire plus en détails le fonctionnement de Linux. Les systèmes Linux [1] sont composés d'un noyau monolithique permentant le chargement dynamique de modules (*drivers*). Ce qui permet d'obtenir une image de noyau de dimension raisonnable, tout en supportant un très grand nombre de périphériques qui pourront être chargés à partir des modules disponibles sur un système de fichiers et non inclus directement dans l'image du noyau. La figure 4.1 illustre son architecture.

Au démarrage, l'initialisation des différents éléments du noyau est effectuée. Puis, un premier programme *user space* est exécuté, habituellement appelé `init`. Ce programme en chargera d'autres via un `fork()` et tous ces programmes seront exécutés en multi-tâches par l'ordonnanceur. La prochaine section décrit plus en détails le processus de démarrage.

4.1 Le démarrage

Le processus de démarrage d'un système d'exploitation est tributaire d'un chargeur (*bootloader*) qui dépend de la plateforme et de l'architecture choisie. Cependant, le processus est très similaire d'une architecture à une autre (par exemple voir le processus pour x86 et pour AM335x ARM).

Au démarrage (ou sur un *reset*), le pointeur d'instruction est initialisé à une valeur prédéfinie. Celui-ci pointe dans la mémoire ROM où est situé le BIOS (ou

1. Plus précisément, GNU/Linux car ceux-ci sont composés d'un noyau Linux et d'un environnement de programmation et d'exécution GNU (binutils, gcc, etc).

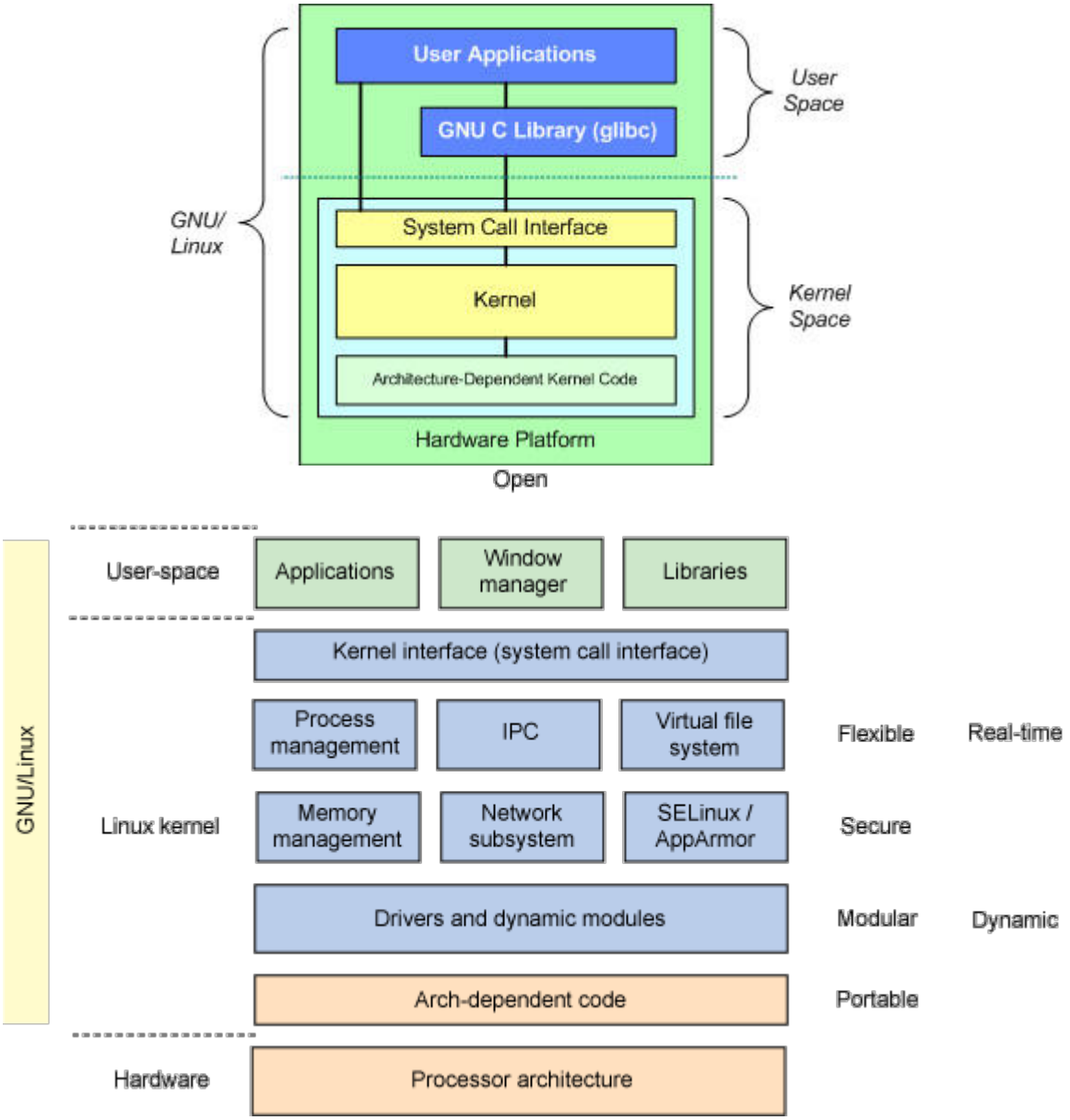


FIGURE 4.1 – Architecture de Linux (source IBM)

UEFI) sous certaines architectures ou un chargeur primaire (*primary bootloader*) sous d'autres architectures. Ce programme, inclus par le fournisseur du matériel, initie une séquence d'initialisation puis scrute différents supports à la recherche d'un chargeur secondaire (dans certains cas on parle du premier stage du chargeur externe). S'il en trouve un, celui-ci est chargé et prend le relai. Celui-ci examine les partitions actives pour un *boot* (systèmes de fichiers) disponibles et appelle le deuxième chargeur externe avec l'information trouvée. Ce deuxième chargeur doit charger en mémoire l'image du noyau (les images de noyau sont souvent compressées et le chargeur doit les décompresser lors du chargement) et appeler celui-ci avec les paramètres appropriés spécifiés dans un fichier de configuration : emplacement du système de fichiers racine (*rootfs*) et son type, emplacement du système de fichier transitoire (optionnel, `initrd` décrit plus loin), console sérielle pour l'affichage des messages de *boot* (optionnel), premier programme à exécuter (habituellement `init`) et autres paramètres spécifiques à configuration matérielle (options graphiques, paramètres pour certains périphériques, *blacklist*, ...).

4.1.1 Pourquoi un `initrd` ?

Lorsque l'on bâtit un système embarquée, on connait à l'avance le système de fichier utilisé pour le système de fichiers racine, par exemple *ext4*. On peut donc inclure le pilote d'interface à même l'image du noyau. Dans ce cas, l'utilisation d'un `initrd` n'est pas nécessaire.

Considérons maintenant le cas d'un système à usage général (exemples : un serveur ou un *desktop*). Comme il existe un grand nombres de systèmes de fichiers et de supports physiques pouvant servir pour le *rootfs*, il n'est pas envisageable de tous les inclurent dans l'image du noyau. Cependant, si le noyau n'a pas le pilote pour le système de fichiers racine, il ne pourra pas le charger et le démarrage ne sera pas possible. C'est ici que le `initrd` entre en jeu. On inclus dans notre noyau le pilote pour un système de fichiers de base (par exemple : ext3) et on construit un système de fichiers racine temporaire dans le même format qui contient les éléments d'un système minimal auxquels on ajoute le(s) module(s) requis pour charger notre système de fichiers racine « permanent » qui lui contient tous les autres modules d'un Linux complet. Ainsi, notre image minimale « standard » combinée avec notre image de disk `initrd` créée spécifiquement pour notre installation sera fonctionnelle (lors d'une mise à jour du noyau sous Ubuntu, par exemple, on remarque la création d'un `initrd` qui est toujours spécifique à notre installation). Ainsi, un fournisseur d'une distribution Linux peut, avec un noyau unique de faible dimension, adapter l'installation à une grande variété de systèmes.

Au démarrage, `initrd` (une image de disque compressée) est décompressée en mémoire (le `rd` est pour *RAM disk* qui contient le `init`) et est utilisé comme *rootfs* en mémoire, puis on charge à partir de ce « disque » les pilotes spécifiques à notre système. Une fois cela réalisé, on peut faire basculer notre noyau vers l'utilisation du *rootfs* « permanent » et récupérer la mémoire utilisée par le `initrd`.

On utilisera aussi `initrd` pour les systèmes sans disque. Au démarrage, le noyau et le système de fichiers seront chargés en mémoire RAM (à partir d'une mémoire permanente ou à travers un réseau). Dans ce cas `initrd` est le système de fichier « permanent » qui résidera en mémoire tout au long de l'opération du système. Au redémarrage, le système est toujours dans le même « état » et on peut y inclure des options de configuration qui occupent un faible espace dans une mémoire permanente (en lecture/écriture). Les mises à jour de tels systèmes sont souvent appelées *firmware upgrade*. Comme les deux images sont compressées (noyau et `initrd`), l'empreinte mémoire pour le ROM (ou autre variante) sera minimale. IBM a produit un bon tutoriel sur le `initrd`.

4.2 Le programme `init`

Comme nous l'avons vu précédemment, le premier programme appelé par le noyau est habituellement `init`. En pratique, ça peut être n'importe quel programme en *user space*. Pour un système à usage général, `init` est un *deamon* (programme qui va s'exécuter en arrière plan) qui sera en charge de démarrer plusieurs « services » (configuration réseau, serveur d'impression, serveur de fichiers, serveur X, gestionnaire des fenêtres, serveur audio, etc ...). Le choix des applications à démarrer dépendra du *runlevel* qui peut être de 0 à 6 dont voici les usages les plus communs

- *0 : Shut down (or halt) the system,*
- *1 : Single-user mode ; usually aliased as s or S,*
- *6 : Reboot the system,*
- *2 : Multiuser mode without networking,*
- *3 : Multiuser mode with networking,*
- *5 : Multiuser mode with networking and the X Window System*

et sont spécifiés dans le fichier `/etc/inittab`. On pourra consulter ce tutoriel pour plus de détails.

Pour un système embarqué, il peut s'agir de notre application spécifique qui sera en charge de démarrer les applications secondaires requises pour l'opération des fonctionnalités requises par notre produit.

4.3 Le système de fichiers

Sous Linux, il n'y a pas de notions de disque (`C:`, etc) comme sous, par exemple, Windows. Le *rootfs* est chargé à la racine `/`. Si d'autres partitions sont utilisées, elles sont chargées (*mounted*) dans un sous-répertoire. Ainsi, tous les fichiers sont visibles à travers une même arborescence. Par exemple, au laboratoire, la commande suivante :

```
% df -h
Filesystem                              Size  Used Avail Use% Mounted on
/dev/sda2                                46G  7.9G   36G  19% /
devtmpfs                                5.9G     0  5.9G   0% /dev
tmpfs                                   5.9G   15M  5.9G   1% /dev/shm
tmpfs                                   5.9G   66M  5.8G   2% /run
tmpfs                                   5.9G     0  5.9G   0% /sys/fs/cgroup
/dev/sda1                               976M  206M  704M  23% /boot
/dev/sda3                               870G  279G  547G  34% /export
srvgrm07:/mypool/locals/local_linux7    100G   12G   89G  12% /usr/local_linux7
nfs:/mypool/users/benacer                15G   23M   15G   1% /users/benacer
tmpfs                                   1.2G  280K  1.2G   1% /run/user/2902
tmpfs                                   1.2G     0  1.2G   0% /run/user/4710
srvgrm02:/mypool/CMC                    4.3T  822G  3.5T  19% /CMC
srvgrm02:/mypool/locals/local_linux7    3.5T   12G  3.5T   1% /usr/local
nfs:/mypool/users/Cours/ele4205/1       2.0G  143M  1.9G   7% /users/Cours/ele4205/1
nfs:/mypool/users/support               250G  138G  113G  55% /users/support
tmpfs                                   1.2G     0  1.2G   0% /run/user/2901
```

permet de constater que les fichiers contenus dans les répertoires

- `/export`,
- `/users/Cours/ele4205/1`,
- `/usr/local` et
- `/users/support`

proviennent respectivement

- de la troisième partition du premier disque local (`/dev/sda3`) et
- de disques réseau :
  - `nfs:/mypool/users/Cours/ele4205/1`,
  - `srvgrm07:/mypool/locals/local_linux7` et
  - `nfs:/mypool/users/support`.

Bien qu'il existe un grand nombre de distributions Linux, elles utilisent presque toutes la hiérarchie standard pour le système de fichiers.

L'accès à tous ces répertoires qu'il soient locaux, des disques différents (SATA, USB, ...), des disques réseau se fait du point de vue de l'usager au travers du VFS qui présente une interface uniforme sans égard aux aspects technologiques

du support. Le VFS « traduits » les droits d'accès possibles du système de fichiers vers les droits Unix (User, Group, World).

Pour terminer, sous Linux, presque tout est un fichier. Il y a les fichiers « standards ». Les sockets TCI/IP, une fois ouverts, sont représentés par un descripteur de fichier dans lequel on peut lire et écrire. Les périphériques sont des fichiers dans le répertoire `/dev` de deux types possibles : `c` — *character device* (carte de son, console `tty`, carte d'acquisition et autres *streams*) ou `b` — *block device* (disques et autres périphériques similaires). Les « cartes réseaux » sont aussi des périphériques mais qui ne sont pas utlisés au tranvert de « fichiers » dans dans le répertoire `/dev`.

4.4 La librairie libc

Comme illustré à la figure 4.1, les programmes usager utilisent la librairie GLibC (ou une variante) pour les appels système au noyau. Les mêmes fonctionnalités sont disponibles d'une architecture à l'autre et permettent aux programmes d'être portables. Illustrons ceci avec la fonction `write` qui permet d'écrire dans un fichier (ici `stdout`). Voici le *Hello World* par un appel système (12 caractères à écrire dans le descripteur de fichier 1=`stdout`) :

```
#include <unistd.h>

char message[] = "Hello World\n";

int main()
{
  write(1,message,sizeof(message)-1);

  return 0;
}
```

On peut exécuter ce programme avec `strace` pour constater l'appel système :

```
% gcc -o hellowrite hellowrite.c
% ./hellowrite
Hello World
% strace ./hellowrite > test.txt
...
...
munmap(0x7fb0ff695000, 184458)          = 0
```

```
write(1, "Hello World\n", 12)           = 12
exit_group(0)                           = ?
+++ exited with 0 +++
```

L'appel système a été effectué avec succès (valeur de retour 12 caractères écrits comme demandé) suivi d'un `exit_group(0)` -> `return 0;`

On pourrait faire la même chose en assembleur mais sans portabilité. Par exemple sous x86 on aurait :

```
.data                   # data

msg:
    .ascii    "Hello World\n"
len = . - msg           # length of string

.text                   # code

  .global _start        # entry point
_start:
                        # write to stdout
  movl    $len,%edx     # third argument: message length
  movl    $msg,%ecx     # second argument: pointer to message to write
  movl    $1,%ebx       # first argument: file handle (stdout)
  movl    $4,%eax       # system call number (sys_write)
  int     $0x80         # call kernel

                        # and exit
  movl    $0,%ebx       # first argument: exit code
  movl    $1,%eax       # system call number (sys_exit)
  int     $0x80         # call kernel
```

ce qui est loin d'être portable !

Une recherche sur le web vous permettra de trouver des versions en assembleur pour différentes architectures. La librairie GLibC développée pour les noyaux des différentes architectures permet la probabilité du code et une interface commune pour plusieurs architectures et systèmes d'exploitation qui implémentent le standard POSIX.

4.5 L'espace mémoire

L'espace mémoire virtuel (adresses linéaires) est séparé en deux sections : l'une en mémoire haute pour le noyau (*Kernel Space*) et l'autre en mémoire basse pour les programmes (*User Space*). Par exemple sous x86, 1 Go est réservé au noyau et 3 Go au programme usager. Les programmes usager ne peuvent ni lire ni écrire en mémoire haute (réservée au *Kernel*) alors que le noyau a accès à tout l'espace mémoire.

Chapitre 5

Chaîne de développement

Dans ce chapitre, nous allons vous présenter les principaux éléments de la chaîne de développement :

GCC : la suite de compilation ;

Binutils : les outils de traitement et d'analyse des fichiers binaires ;

les librairies : statiques et dynamiques ;

la structure des répertoires d'un chaine de compilation : où sont placés les différents éléments par rapport au *sysroot* ;

l'utilisation de la compilation croisée : principes de base, configuration et utilisation efficiente.

5.1 La suite de compilation GCC

L'environnement de programmation sous les systèmes dérivés de Unix est basée sur la suite de compilation GGC. Bien que certaines informations soient spécifiques à Linux, la majorité des outils présentés sont disponibles sous différentes *variantes* de Unix et même sous MS Windows. Ce chapitre se veut un aperçu et ne constitue pas une référence complète en soit. Le lecteur pourra se référer à documentation en ligne disponible. Cette suite est le résultat d'un développement sur près de trois décennies. Le nombre de processeurs et d'architectures supportés est très impressionnant. GCC s'appuie sur la suite Binutils pour plusieurs aspects comme nous le verrons plus bas.

La figure 5.1 présente l'environnement de programmation sous Linux. Le programme Xemacs pour l'édition du code source est montré à titre d'exemple. Le

contrôle de la compilation est fait par l'analyseur de dépendances make. as est disponible comme assembleur et gcc (g++) comme compilateur C (C++). L'édition des liens est la tâche de ld (qui peut être appelé automatiquement par gcc). Les librairies statiques et dynamiques sont créées à l'aide de ar et gcc (g++) respectivement. Le débogage se fait à l'aide de gdb. Ces différents outils sont souvent appelés implicitement par les environnements de développement intégrés (IDE).

Dans la figure 5.1, les cercles numérotés représentent les opérations :

1. d'édition des fichiers contenant le code source ;
2. d'analyse des dépendances entre les sources et les cibles ;
3. de traduction en langage machine ;
4. de création d'une librairie statique (si nécessaire) ;
5. de création d'une librairie dynamique (si nécessaire) ;
6. d'édition des liens pour la création du programme exécutable ;
7. de débogage du programme.

5.2 Le compilateur C GNU gcc

Le compilateur gcc (et g++) permet la compilation de programmes en langage C (fichiers `*.c`), en langage C++ (fichiers `*.cpp`) et en langage assembleur (fichiers `*.s`, par un appel à as). Après la compilation, et à moins que l'option `-c` soit utilisée, l'éditeur de liens ld est appelé automatiquement avec le module de « startup » et les librairies standards (plus de détails à la prochaine section). Notons que gcc et g++ diffèrent, entre autres, en ce qui concerne les librairies standards qui sont utilisées lors de la création de l'exécutable.

Considérons le programme `hello.c` : la commande

```
gcc -o hello hello.c
```

crée l'exécutable `hello` alors que la commande

```
gcc -g -o hello hello.c
```

crée l'exécutable `hello` contenant l'information de débogage. La commande

```
gcc -g -c -o hello.o hello.c
```

ou l'équivalent

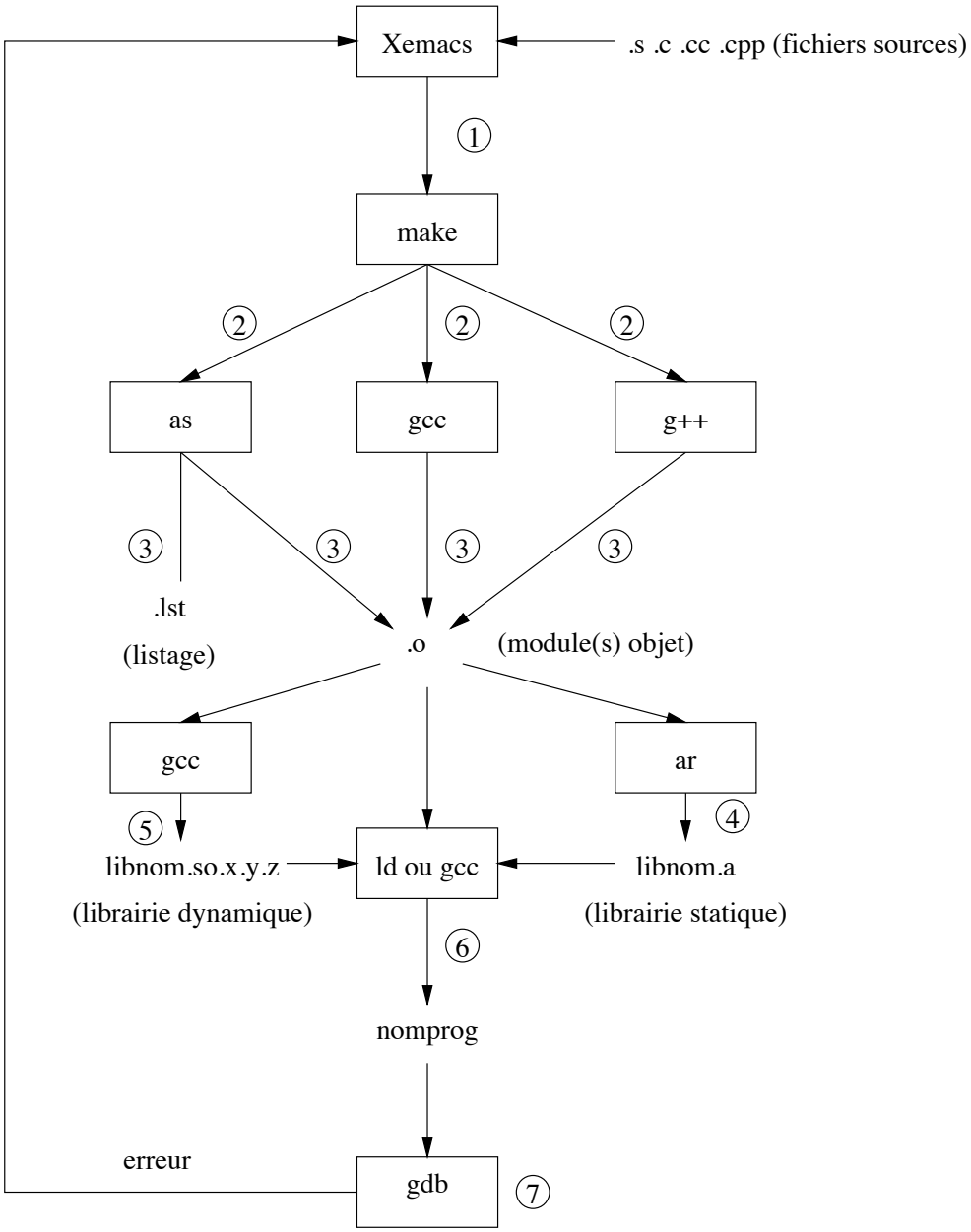


FIGURE 5.1 – Environnement de programmation sous Linux

```
gcc -g -c hello.c
```

permet de créer le module objet `hello.o` contenant l'information de débogage. Enfin,

```
gcc -o hello hello.o
```

appelle l'éditeur de liens pour créer l'exécutable `hello` en utilisant les modules et librairies standards du langage C. Plusieurs fichiers peuvent être spécifiés sur la ligne de commande. Par exemple,

```
gcc -o prog prog.c module.o
```

compile le fichier source `prog.c` dans un module objet et appelle l'éditeur de liens avec ce module, `module.o` et les librairies et modules standards. L'option `-v` permet de connaître les différentes librairies utilisées.

TABLE 5.1 – Principales options utilisées avec gcc

`--help`	affiche le message d'aide
`-c`	compile mais n'appelle pas l'éditeur de liens
`-g`	génère les informations pour le débogage du code source
`-IDIRECTORY`	ajoute le répertoire `DIRECTORY` au chemin de recherche des inclusions
`-lLIBNAME`	recherche la librairie `LIBNAME`
`-LDIRECTORY`	ajoute le répertoire `DIRECTORY` au chemin de recherche des librairies
`-o FILE`	nomme le fichier de sortie `FILE`
`-Wall`	affiche tous les messages d'avertissement
`-Wl,<options>`	passe les `<options>` (séparées par virgules mais sans espace) à l'éditeur de liens

Le tableau 5.1 présente les principales options utilisées avec gcc. Utiliser la commande `gcc --help` ou `man gcc` pour obtenir plus de détails. On pourra aussi se référer à la documentation en-ligne.

Par défaut, gcc recherche les fichiers d'entêtes (*header files*, `*.h` par exemple) dans le répertoire `/usr/include`. Pour générer un fichier `.o` à partir d'un fichier `.c`, gcc appelle le pré-processeur `cpp` qui traitera toutes les entêtes (inclusions et directives), retirera les commentaires et interprétera les macros. La commande

```
gcc -E hello.c -o hello.i
```

permet d'arrêter la « compilation » après le pré-processeur. Ensuite, la sortie de cpp est compilée pour générer du code assembleur pour l'architecture cible. La commande

```
gcc -S hello.c
```

génère le fichier assembleur `hello.s`. La sortie du compilateur est ensuite dirigée vers l'assembleur `as` qui finalement génère le fichier objet `.o`. On applique ce processus à tous nos fichiers `.c`.

Tous ces programmes sont situés dans le répertoire `/usr/bin`.

5.3 L'éditeur de liens GNU ld

L'éditeur de liens ld permet de combiner des modules objets et librairies pour créer un programme exécutable. Pour y parvenir, l'éditeur de liens ld tente de résoudre les références aux symboles non-définis entre les différents modules qui lui sont passés sur la ligne de commande. Après cette première étape et s'il reste des références encore non résolues, ld recherchera ceux-ci dans les différentes librairies qui auront été spécifiées sur la ligne de commande.

L'éditeur de liens est appelé automatiquement lorsque le fichier de sortie est un exécutable lors d'un appel à gcc (ou g++). Dans le cas du C/C++, les librairies standards de même que les modules d'initialisation (*startup*) et de terminaison (*cleanup*) nécessaires sont inclus à l'édition des liens en plus de arguments passés lors de l'appel sans intervention de l'utilisateur. Parmi les modules d'initialisation, le module objet `crt1.o` contient un appel à la fonction `int main()`.

Par exemple, la commande :

```
gcc -o hello hello.o
```

est équivalent d'une très longue commande à `ld` appelée indirectement par le programme `collect2`. Elle créera l'exécutable `hello` en utilisant le module `hello.o`, le module de « startup » et les librairies standards. La commande

```
gcc -v -o hello hello.o
```

permet de visualiser les commandes complètes exécutée pour créer l'exécutable `hello`.

Lors de l'édition des liens, les librairies sont recherchées dans les répertoires `/lib` et `/usr/lib`.

5.4 Les librairies

Lorsque l'on développe un programme, un ensemble de fonctions ayant un caractère assez général peut être regroupé dans une librairie pour une réutilisation dans d'autres projets (i.e. une librairie de fonctions de conversion entre différents formats graphiques). Il est possible de créer deux types de librairie [1] :

— une librairie statique ;
— une librairie dynamique.

La librairie statique est une archive de modules objets qui pourront être extraits et inclus dans le programme final par l'éditeur de liens. Ainsi, chaque programme utilisant une même fonction aura sa propre copie de la fonction. Une librairie dynamique, par contre, est un « exécutable » auquel est lié le programme utilisant certaines des fonctions. Les programmes utilisant une même fonction n'ont pas leur copie de cette fonction, mais contiennent un lien vers la librairie dynamique contenant cette fonction. Ceci permet un usage plus efficient de l'espace disque et de l'espace mémoire surtout lorsqu'un grand nombre de programmes utilise la même librairie.

5.4.1 Les librairies statiques

Les librairies statiques regroupent plusieurs modules objets dans un seul fichier qui pourra être utilisé avec ld. On peut créer une librairie statique à l'aide de ar qui s'utilise comme suit (voir le tableau 5.2 pour les options de la ligne de commande) :

```
ar [-]{dmpqrstx}[abcfilNoPsSuvV] [member-name] fichier-archive fichier...
```

Ainsi, la commande

```
ar rc libproj.a proj1.o proj2.o proj3.o
```

créera ou mettra à jour le fichier `libproj.a` en remplaçant ou ajoutant les fichiers `proj1.o`, `proj2.o` et `proj3.o`. Ce fichier pourra être utilisé comme librairie avec l'éditeur de liens. Par exemple, la commande

```
gcc -o hello hello.c -L./ -lproj
```

1. Pour plus de détails voir le document : Static, Shared Dynamic and Loadable Linux Libraries.

permet de compiler le programme `hello.c` et d'ajouter la librairie `libproj.a` à la liste des librairies lors de l'appel de l'éditeur de liens. Sous Linux, les librairies statiques ont des noms du type : `libnom.a` où `nom` est utilisé comme nom de librairie lors de l'appel de gcc lors de l'édition des liens. Voir la documentation en-ligne de ar pour plus de détails.

TABLE 5.2 – Principales opérations et options utilisées avec ar

`d`	enlève le(s) fichier(s) de l'archive
`m[abi]`	déplace le(s) fichier(s) dans l'archive
`p`	affiche les fichiers trouvés dans l'archive
`q[f]`	ajout rapide du(des) fichier(s) dans l'archive à la fin
`r[abi][f][u]`	remplace ou ajoute le(s) fichier(s) dans l'archive
`t`	affiche le contenu de l'archive
`x[o]`	extrait le(s) fichier(s) de l'archive
`[a]`	place le(s) fichier(s) après `[member-name]`
`[b]` ou `[i]`	place le(s) fichier(s) avant `[member-name]`
`[f]`	tronque les noms de fichiers
`[o]`	conserve les dates des fichiers
`[u]`	remplace seulement les fichiers les plus récents que ceux contenus dans l'archive
`[c]`	pas d'avertissement si l'archive doit être créée
`[s]`	crée un index des symboles d'une archive (voir ranlib)
`[S]`	ne crée pas de table des symboles
`[v]`	affiche le détail de la commande

5.4.2 Les librairies dynamiques

Dans bon nombre de systèmes Linux, on retrouve les librairies dynamiques dans, entre autres, les répertoires `/lib`, `/usr/lib` et `/usr/local/lib`. Lorsqu'un programme est chargé et qu'il requiert une fonction contenue dans une librairie dynamique, cette librairie doit être chargée. L'information sur le nom de la librairie a été incluse dans le programme par ld. Sous Linux, les librairies dynamiques de format ELF ont des noms du type : `libnom.so.x.y.z` où `x.y.z` identifie la version de la librairie. Comme nous le verrons plus loin, il existent souvent des liens

symboliques vers les librairies dynamiques et ces liens sont importants pour le bon fonctionnement des programmes utilisant la librairie. Pour connaître le nom des librairies dynamiques utilisées par un programme, on peut utiliser ldd (*List Dynamic Dependencies*). Par exemple,

```
$ ldd hello
        linux-vdso.so.1 =>  (0x00007fff085ff000)
        libc.so.6 => /lib64/libc.so.6 (0x0000003b5e000000)
        /lib64/ld-linux-x86-64.so.2 (0x0000003b5d800000)
```

On voit donc que le programme `hello` dépend entre autres de la librairie `libc.so.6`. Le programme nm permet d'obtenir une liste des symboles utilisés et/ou définis dans une librairie, par exemple

```
nm /lib64/libc.so.6
```

Lors de l'appel de ld, la recherche du fichier d'une librairie dynamique se fera dans les répertoires par défaut (`/usr/lib` par exemple) et dans les répertoires spécifiés par l'option `-L`. Pour le format ELF, l'option `-lnom` initie une recherche, dans l'ordre, de `libnom.so` puis `libnom.a`. Très souvent, `libnom.so` est un lien symbolique vers `libnom.so.x`.

Comme plusieurs programmes peuvent se référer à une librairie dynamique, et que cette librairie sera fort probablement modifiée pour régler des « bogues », un mécanisme pour le contrôle des versions doit être mis en place. Certaines révisions introduisent des changements dans le comportement des fonctions existantes alors que d'autres ne changent pas le comportement mais règlent des problèmes. Il existent des révisions « majeures » et des révisions « mineures ». Une révision mineure permet à un ancien programme, qui utilisait la librairie avant la révision, d'utiliser la librairie après la révision. Une révision majeure introduit une incompatibilité de fonctionnement entre les anciens programmes et la nouvelle librairie (changements dans l'API). Par exemple, `libexemple.so.5.2.18` a comme révision « majeure » 5 et comme révision « mineure » 2.18 (plus précisément : révision 2, « patch-level » 18).

Voici les étapes pour bâtir une librairie dynamique de type ELF avec les fichiers `proj1.c`, `proj2.c` et `proj3.c`.

```
gcc -fPIC -c proj1.c proj2.c proj3.c
```

qui permet d'obtenir les modules `proj1.o`, `proj2.o` et `proj3.o` (l'option `-fPIC` : *position-independent code*).

Puis la commande

```
gcc -shared -Wl,-soname,libproj.so.1 -o libproj.so.1.0 proj1.o proj2.o proj3.o
```

Génère la librairie dynamique en utilisant les options :
- `-shared` pour une librairie dynamique ;
- `-Wl,-soname,libproj.so.1` options pour ld : nom de librairie dynamique pour le « dynamic loader » ;
- `-o libproj.so.1.0` le nom du fichier contenant la librairie.

Comme ce sont les révisions majeures qui distinguent les incompatibilités entre les librairies, on doit créer le lien symbolique (voir `man ln` pour plus d'informations) :

```
ln -s libproj.so.1.0 libproj.so.1
```

pour permettre au « dynamic loader » qui tente de charger `libproj.so.1` de charger la dernière version (ici `libproj.so.1.0` car le `soname` est `libproj.so.1`). Maintenant que la librairie existe, il faut permettre à ld de retrouver la librairie lors de l'inclusion de `proj` dans les librairies. Par exemple

```
gcc -o hello hello.c -L./ -lproj
```

Il faut donc créer un autre lien symbolique

```
ln -s libproj.so.1 libproj.so
```

Lors du développement de la librairie, il est souvent utile d'utiliser

```
LD_LIBRARY_PATH=‘pwd‘:$LD_LIBRARY_PATH ; export LD_LIBRARY_PATH
```

sous bash (attention c'est `‘pwd‘` et non `’pwd’`). Sous tcsh, on utilisera :

```
setenv LD_LIBRARY_PATH ‘pwd‘:$LD_LIBRARY_PATH
```

pour ajouter le répertoire courant au chemin de recherche des librairies. Quand tout fonctionne, on peut déplacer la librairie dans, par exemple, `/usr/local/lib`, et on doit recréer les liens symboliques :

```
su
cp libproj.so.1.0 /usr/local/lib
/sbin/ldconfig
( cd /usr/local/lib ; ln -s libproj.so.1 libproj.so )
```

Chaque librairie a un `soname` et c'est ce nom que l'éditeur de liens ld va inclure dans l'exécutable, et c'est ce fichier que le « dynamic loader » va rechercher, d'où :

```
libproj.so => libproj.so.1 => libproj.so.1.0
```

5.5 Les outils de Binutils

Les outils les plus utilisés de GNU/Binutils sont :

ld : l'éditeur de liens ;

as : l'assembleur de GNU ;

ar : création, modification et gestion de archives de librairies statiques.

Il y a aussi

addr2line : convertit une adresse mémoire d'un binaire en un nom de fichier et un numéro de ligne.

c++filt : un filtre de conversion des symboles *mangled* du C++ vers les noms et prototypes originaux des fonctions ou méthodes.

dlltool : création des fichiers requis pour les DLL (librairies dynamiques sous MS Windows).

gprof : affiche les informations de profilage d'un programme.

nm : liste les symboles d'un fichier objet ou executable.

objcopy : copie et traduit des fichiers objets.

objdump : affiche les informations concernant un fichier objet.

ranlib : génère un index de contenu d'une archive `ar` (accélère l'édition des liens).

readelf : affiche les informations pour un fichier de format ELF.

size : affiche les sections (leurs dimensions) d'une archive ou d'un fichier objet.

strings : affiche les chaînes de caractères imprimable d'un fichier.

strip : retire les symboles d'un fichier.

windmc : compilateur de messages (MS Windows).

windres : compilateur de fichiers de ressources (MS Windows).

Vous allez utiliser certains de ces outils au laboratoire. Référez vous à la documentation pour plus de détails.

Tous ces programmes sont situés dans le répertoire `/usr/bin`.

5.6 La compilation croisée

La compilation croisée consiste à compiler une application pour une plateforme de type B (cible, *target*) sur un ordinateur de type A (hôte, *host*). Les deux systèmes peuvent se différencier par leurs architectures et/ou leurs systèmes d'exploitation. Ainsi, une station de travail très « puissante » peut être utilisée pour compiler rapidement une application pour un système embarqué de capacité très modeste. Dans ce chapitre, nous allons décrire la structure d'installation d'une chaîne de compilation croisée ainsi que différentes façons de l'utiliser. La génération d'une telle chaîne de compilation est présentée en annexe C.

5.6.1 Éléments de base

Un chaîne de compilation croisée comprend les éléments suivants :

les outils de compilation : ce sont des exécutables au format type A (Binutils et la suite GGC) qui génèrent du code pour la plaforme type B. Par exemple, le programme `arm-poky-linux-gnueabi-gcc` est une version de `gcc` qui va produire du code pour une architecture `arm` et un système `linux` utilisant l'ABI `gnueabi`. Pour cette suite croisée, tous les outils présentés dans les sections précédentes seront disponibles grâce à l'utilisation du préfixe `arm-poky-linux-gnueabi-`. Donc, chaque suite de compilation croisée aura son préfixe.

les fichiers d'inclusion et les librairies de la cible : pour compiler avec succès un programme, il faut avoir accès aux fichiers d'entête (*headers*) de la cible pour générer les `.o` ainsi qu'aux modules standards et aux librairies (statiques et dynamiques) spécifiques à la cible pour l'édition des liens. Il faudra donc indiquer au compilateur où se trouvent ces fichiers car il ne peut pas utiliser les versions natives qui sont dans `/usr/include`, `/lib` et `/usr/lib` car celles-ci seront généralement incompatibles avec la cible.

Comme les outils de compilation seront dans un répertoire spécifique, il faudra ajouter ce répertoire au chemin de recherche des exécutables afin que l'on puisse exécuter directement, par exemple, `arm-poky-linux-gnueabi-gcc` sans avoir à inclure le chemin complet où se trouve l'exécutable.

Cependant, par défaut, `gcc` recherche les entêtes dans `/usr/include` et les librairies dans `/lib` et `/usr/lib`, ce qui n'est pas adéquat mais un mécanisme simple et astucieux permet de régler de problème : l'option `--sysroot=<dir>`. Si, par exemple, les copies de nos fichiers d'entêtes et librairies sont dans des sous-

répertoires de `/opt/cross/arm-bbb` et respectent l'arborescence « standard », alors l'option `--sysroot=/opt/cross/arm-bbb` fera que les entêtes, les modules standards et les librairies seront recherchées dans `/opt/cross/arm-bbb/usr/include`, `/opt/cross/arm-bbb/lib` et `/opt/cross/arm-bbb/usr/lib`

Il est d'usage d'écrire un fichier de configuration qui défini un environnement de travail spécifique à notre compilation croisée. Par exemple, au laboratoire, vous avez utilisé la commande (créée automatiquement sous Yocto) :

```
$ source /export/tmp/4205_nn/opt/poky/environment-setup-aarch64-poky-linux
```

Le contenu de ce fichier pour une installation dans `/opt/poky` est présenté la page suivante.

On constate que cet environnement modifie le `PATH`, définit un sysroot et les variables standards de compilation utilisée par GNU Make (cet outil sera présenté au prochain chapitre). L'utilisation d'un tel fichier permet de passer à un environnement de compilation croisée en une simple commande qui, si le script a été adéquatement établi, sera compatible avec les outils de développement GNU Make, Autoconf, CMake, Eclipse, etc.

```
export SDKTARGETSYSROOT=/opt/poky/sysroots/aarch64-poky-linux
export PATH=/opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin:/opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin/../x86_64-pokysdk-linux/bin:
   /opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux:/opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux-uclibc:
   /opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux-musl:$PATH
export CCACHE_PATH=/opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin:/opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin/../x86_64-pokysdk-linux/bin:
   /opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux:/opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux-uclibc:
   /opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux-musl:$CCACHE_PATH
export PKG_CONFIG_SYSROOT_DIR=$SDKTARGETSYSROOT
export PKG_CONFIG_PATH=$SDKTARGETSYSROOT/usr/lib/pkgconfig
export CONFIG_SITE=/opt/poky/site-config-aarch64-poky-linux
export OECORE_NATIVE_SYSROOT="/opt/poky/sysroots/x86_64-pokysdk-linux"
export OECORE_TARGET_SYSROOT="$SDKTARGETSYSROOT"
export OECORE_ACLOCAL_OPTS="-I /opt/poky/sysroots/x86_64-pokysdk-linux/usr/share/aclocal"
export PYTHONHOME=/opt/poky/sysroots/x86_64-pokysdk-linux/usr
unset command_not_found_handle
export CC="aarch64-poky-linux-gcc  --sysroot=$SDKTARGETSYSROOT"
export CXX="aarch64-poky-linux-g++  --sysroot=$SDKTARGETSYSROOT"
export CPP="aarch64-poky-linux-gcc -E  --sysroot=$SDKTARGETSYSROOT"
export AS="aarch64-poky-linux-as "
export LD="aarch64-poky-linux-ld  --sysroot=$SDKTARGETSYSROOT"
export GDB=aarch64-poky-linux-gdb
export STRIP=aarch64-poky-linux-strip
export RANLIB=aarch64-poky-linux-ranlib
export OBJCOPY=aarch64-poky-linux-objcopy
export OBJDUMP=aarch64-poky-linux-objdump
export AR=aarch64-poky-linux-ar
export NM=aarch64-poky-linux-nm
export M4=m4
export TARGET_PREFIX=aarch64-poky-linux-
export CONFIGURE_FLAGS="--target=aarch64-poky-linux --host=aarch64-poky-linux --build=x86_64-linux --with-libtool-sysroot=$SDKTARGETSYSROOT"
export CFLAGS=" -O2 -pipe -g -feliminate-unused-debug-types"
export CXXFLAGS=" -O2 -pipe -g -feliminate-unused-debug-types"
export LDFLAGS="-Wl,-O1 -Wl,--hash-style=gnu -Wl,--as-needed"
export CPPFLAGS=""
export KCFLAGS="--sysroot=$SDKTARGETSYSROOT"
export OECORE_DISTRO_VERSION="2.0.2"
export OECORE_SDK_VERSION="2.0.2"
export ARCH=arm64
export CROSS_COMPILE=aarch64-poky-linux-

# Append environment subscripts
if [ -d "$OECORE_TARGET_SYSROOT/environment-setup.d" ]; then
    for envfile in $OECORE_TARGET_SYSROOT/environment-setup.d/*.sh; do
    source $envfile
    done
fi
if [ -d "$OECORE_NATIVE_SYSROOT/environment-setup.d" ]; then
    for envfile in $OECORE_NATIVE_SYSROOT/environment-setup.d/*.sh; do
    source $envfile
    done
fi
```

Chapitre 6

Outils de développement

Maintenant que nous savons comment compiler nos applications, nous allons présenter des outils de développement qui vont nous permettre d'être plus productif dans la création de nos applications :

- L'analyseur de dépendances GNU make ;
- La suite Autotools ;
- CMake qui combine GNU make et Autotools d'une façon très portable ;
- Les gestionnaires distribués de code source, plus précisément GIT ;
- Le système de documentation automatisée Doxygen (très similaire à JavaDoc) ;
- Le Markdown pour les fichiers textes (les README, etc).

Tous ces outils peuvent être utilisés directement ou indirectement dans l'interface de l'IDE Eclipse présenté au prochain chapitre.

6.1 L'analyseur de dépendances GNU make

Pour éviter d'avoir à entrer de façon répétitive les commandes nécessaires à la génération de l'exécutable et pour éviter la compilation de modules qui n'ont pas été modifiés depuis la dernière compilation, on utilise l'analyseur de dépendances make. C'est un outil extrêmement puissant qui utilise, entre autres, les dates et heures des fichiers constituant les programmes sources, les modules objets, les librairies pour effectuer seulement les opérations nécessaires pour la mise à jour des composantes du projet de programmation. Il utilise un fichier décrivant les dépendances des différentes cibles à construire. On retrouve l'équivalent dans les outils intégrés de développement (IDE) qui présentent l'arborescence des projets

à compiler. Avant d'écrire un `makefile`, il faut définir les dépendances entre les différents fichiers.

Considérons l'exemple de la figure 6.1. On remarque qu'un changement dans le fichier d'entête `figures.h` nécessite la (re)compilation des quatre fichiers sources `.c` suivie d'une édition des liens :

```
$ gcc -c figures.c
$ gcc -c ligne.c
$ gcc -c cercle.c
$ gcc -c carre.c
$ gcc -o figures figures.o ligne.o cercle.o carre.o
```

On réalise rapidement que, pour un projet réel contenant des dizaines (voir des centaines) de fichiers, une gestion manuelle de la compilation serait très fastidieuse. De plus, lorsqu'un seul fichier est modifié, il est beaucoup rapide de recompiler seulement ce qui est requis.

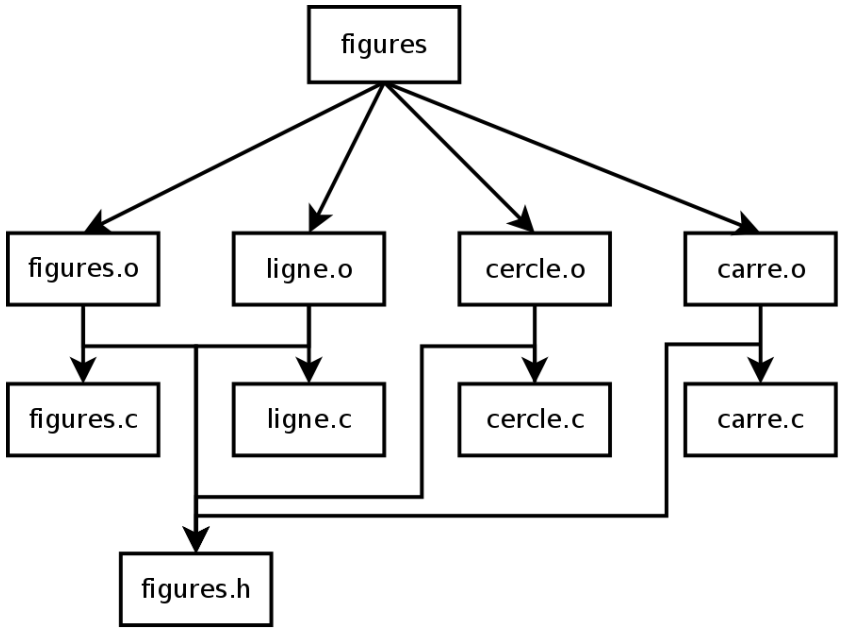


FIGURE 6.1 – Dépendances entre différents fichiers

GNU make permet d'automatiser la compilation en utilisant un fichier `makefile` contenant des cibles et des règles de construction.

Le tableau 6.1 présentent les principales options de make qu'on exécute comme suit :

```
make [options] [cibles] ...
```

Avec `cible` (target) : le nom d'une ou plusieurs cibles à construire. Si aucune cible n'est spécifiée, la première cible présente dans le fichier sera construite.

TABLE 6.1 – Principales options utilisées avec make

`-h, --help`	affiche le message d'aide
`-B, -always-make`	compilation inconditionnelle des cibles
`-C DIRECTORY, --directory=DIRECTORY`	changement de répertoire avant l'exécution du makefile
`-d, --debug[=FLAGS]`	affichage d'informations pour déboguer le makefile
`-f FILE, --file=FILE, --makefile=FILE`	lit le fichier `FILE` comme le `makefile`
`-i, --ignore-errors`	ignore toutes les erreurs des commandes
`-j [N], -jobs[=N]`	permet N tâches simultanées
`-k, --keep-going`	effectue toutes les cibles possibles
`-n, --just-print, --dry-run, --recon`	les règles sont interprétées, mais les commandes ne sont pas exécutées
`-o FILE, --old-file=FILE, --assume-old=FILE`	considère le fichier `FILE` comme plus vieux que les cibles
`-p, --print-data-base`	vue d'ensemble sur les règles et les dépendances
`-r, --no-builtin-rules`	désactive les règles implicites
`-s, --silent, --quiet`	n'affiche pas les commandes
`-t, --touch`	les fichiers dépendants ne sont pas créés à partir des règles, mais leur date est mise à jour
`-w, --print-directory`	affiche le répertoire

Lors de l'appel de make, le programme cherche les fichiers suivants en ordre d'apparition : `GNUmakefile`, `makefile`, `Makefile` ou spécifié par l'option `-f FILE`. Ces fichiers contiennent les règles de construction ou les commandes. Un fichier makefile pourra contenir les éléments suivants :

— des commentaires ;

— des macros ;
— de règles explicites ;
— de règles implicites ;
— des directives.

Le début d'un commentaire est marqué par le caractère `#` et se termine par une fin de ligne. Le format général d'un fichier `makefile` est le suivant :

```
[définition de macros]
...
règle
        commande
        ...
règle
        commande
        ...
```

Les commandes associées à une règle doivent être déplacées d'au moins **une tabulation** vers la droite. Les commandes sont dans un format similaire à celles utilisées dans l'interpréteur (le *shell*).

6.1.1 Règles explicites

Les règles explicites ont la forme suivante :

```
cible [cible ] : [dépendant] [dépendant...]
        [commande]
```

ou

```
cible [cible ] : [dépendant] [dépendant...];    [commande]
        [commande]
```

avec
`cible` : le(s) fichier(s) à mettre à jour ;
`dépendant` : le(s) fichier(s) cible(s) dépend(ent) du(des) dépendants ;
`commande` : la(les) commande(s) à effectuer pour la mise à jour de la cible.
La règle sera considérée comme VRAI si au moins un des fichiers dépendants est plus récent (date) que la cible. La règle doit commencer à la première colonne à gauche et les commandes doivent être précédées d'au moins une tabulation (**attention** : pas le nombre d'espaces équivalent).

Par exemple, le fichier ci-dessous permet de mettre en oeuvre les dépendances de la figure 6.1.

```
# utilisation des règles explicites
# make1.mk

figures : ligne.o cercle.o carre.o figures.o
        gcc -o figures -g ligne.o cercle.o carre.o figures.o

ligne.o : ligne.c figures.h
        gcc -c ligne.c

cercle.o : cercle.c figures.h
        gcc -c cercle.c

carre.o : carre.c figures.h
        gcc -c carre.c

figures.o : figures.c figures.h
        gcc -c figures.c
```

Si la liste des fichiers dépendants est vide, cela implique que les commandes seront toujours effectuées. La cible `.PHONY` est une pseudo-cible, c'est-à-dire elle ne correspond pas à un fichier et qu'il n'est pas nécessaire de la créer.

Dans l'exemple suivant,

```
# utilisation des règles explicites
# make2.mk

figures : ligne.o cercle.o carre.o figures.o
        gcc -o figures -g ligne.o cercle.o carre.o figures.o

ligne.o : ligne.c figures.h
        gcc -c ligne.c

cercle.o : cercle.c figures.h
        gcc -c cercle.c
```

```
carre.o : carre.c figures.h
        gcc -c carre.c

figures.o : figures.c figures.h
        gcc -c figures.c

.PHONY : clean

clean :
        rm *.o
        rm figures
```

l'appel de make pourra se faire de deux manières différentes :

— `make` pour créer la cible `figure`

— `make clean` pour la cible `clean`

On peut dans un `makefile` avoir la possibilité de construire plusieurs cibles ou une seule cible. Dans l'exemple précédent, on peut exécuter le `makefile` selon les choix suivants :

— `make` ou `make figures` on crée la cible `figures`.

— `make ligne.o` la cible `ligne.o` est créée.

— `make ligne.o carre.o` les cibles `ligne.o` et `carre.o` sont crées.

6.1.2 Règles implicites

Les règles implicites ont la forme suivante :

```
%.ExtensionCible : %.ExtensionDépendant
        [commande]
```

avec

`%.ExtensionCible` : l'extension du fichier à mettre à jour ;

`%.ExtensionDépendant` : l'extension du fichier dépendant ;

`commande` : la(les) commande(s) à effectuer pour mettre à jour.

Par exemple, les règles :

```
%.o : %.c
        gcc -c -o $@ $<
hello : hello.o
        gcc -o $@ $<
```

feront que lors de la sauvegarde du fichier `hello.c` sa date sera plus récente que `hello.o` s'il existe. Comme `hello` dépend de `hello.o` et qu'il existe une règle implicite pour générer un fichier `.o` à partir d'un fichier `.c`, cette règle implicite sera utilisée (ici la macro `$@` signifie la cible et `$<` le dépendant), pour produire les commandes

```
gcc -c -o hello.o hello.c
gcc -o hello hello.o
```

Reprenons notre exemple avec des règles implicites (la macro `$+` représente tous les dépendants) :

```
# utilisation des règles implicites
# make3.mk

%.o : %.c
        gcc -c -o $@ $<

figures : ligne.o cercle.o carre.o figures.o
        gcc -o $@  $+

ligne.o : ligne.c figures.h

cercle.o : cercle.c figures.h

carre.o : carre.c figures.h

figures.o : figures.c figures.h

.PHONY : clean

clean :
        rm *.o
        rm figures
```

Cette façon de faire a un net avantage, si l'on desire changer les options de compilation, un seul changement est requis.

Make possède des règles implicites prédéfinies. Pour désactiver l'utilisation de ces règles, on exécutera la commande :

```
make -r
```

L'exemple suivant nous montre qu'il n'est pas nécessaire d'énoncer toutes les dépendances dans le `makefile`.

```
# utilisation des règles implicites
# make4.mk

%.o : %.c figures.h
        gcc -c -o $@ $<

figures : ligne.o cercle.o carre.o figures.o
        gcc -o $@  $+

.PHONY : clean

clean :
        rm *.o
        rm figures
```

.

6.1.3 Macros

Une macro est un nom qui représente une chaîne de caractères. Quand make rencontre une macro, il la remplace par la chaîne de caractères prédéfinie par la macro et traite les macros (s'il y en a) qui sont contenues dans la chaîne de caractères de remplacement. Ce qui permet d'inclure une macro dans une macro. Attention, la récursivité n'est pas permise. Une macro est définie de la façon suivante :

```
nom_macro = texte
```

est utilisée par la forme suivante

```
$(nom_macro)
```

Par exemple, le `makefile`

```
LIBPROJ = proj1.o proj2.o proj3.o
%.o : %.c
        gcc -c -o $@ $<
libproj.a : $(LIBPROJ)
        ar rc $@ $(LIBPROJ)
```

résulte dans les opérations suivantes :

```
gcc -c -o proj1.o proj1.c
gcc -c -o proj2.o proj2.c
gcc -c -o proj3.o proj3.c
ar rc libproj.a proj1.o proj2.o proj3.o
```

Il est aussi possible d'effectuer un remplacement de chaînes de caractères dans une macro :

```
$(nom_macro:orig=rempl)
```

ainsi le `makefile` précédent peut être reformulé comme suit :

```
LIBPROJ = proj1.c proj2.c proj3.c
%.o : %.c
        gcc -c -o $@ $<
libproj.a : $(LIBPROJ:.c=.o)
        ar rc $@ $(LIBPROJ:.c=.o)
```

et génère les mêmes commandes.

Afin de faciliter et de simplifier son utilisation, make a un ensemble de macros prédéfinies (voir le tableau 6.2)

6.1.4 Catalogue des règles implicites

Make utilise une série de règles implicites (voir le tableau 6.3) à l'aide de macros prédéfinies (voir tableau 6.4). La commande `make -p` permet d'obtenir la liste des règles implicites prédéfinies.

6.1.5 Cibles multiples

Dans certains cas, il sera utile d'utiliser des cibles multiples de pair avec des règles implicites. Par exemple :

TABLE 6.2 – Macros prédéfinies sous make (liste partielle)

`$@`	le chemin et nom de fichier de la cible
`$<`	le chemin et nom de fichier du premier dépendant
`$?`	le chemin et nom de tous les dépendants plus récents que la cible séparés par un espace
`$^` `$+`	le chemin et nom de tous les dépendants séparés par un espace, `$^` supprime les dépendances multiples alors que `$+` les conserve et en préserve l'ordre
`$*`	le chemin et nom de fichier (sans extension) du radical d'une règle implicite
`$(@D)` `$(@F)`	la partie répertoire et la partie fichier de `$@`
`$(*D)` `$(*F)`	la partie répertoire et la partie fichier de `$*`
`$(<D)` `$(<F)`	la partie répertoire et la partie fichier de `$<`
`$(^D)` `$(^F)`	la partie répertoire et la partie fichier de `$^`
`$(+D)` `$(+F)`	la partie répertoire et la partie fichier de `$+`
`$(?D)` `$(?F)`	la partie répertoire et la partie fichier de `$?`

TABLE 6.3 – Règles implicites prédéfinies (liste partielle)

`.c` en `.o`	`$(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@`
`.cc`, `.C` ou `.cpp` en `.o`	`$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@`
`.s` en `.o`	`$(AS) $(ASFLAGS) $< -o $@`
`fichier.o` en `fichier`	`$(CC) $(LDFLAGS) $^ $(LOADLIBES) -o $@`

TABLE 6.4 – Macros prédéfinies des règles implicites (liste partielle)

CC	le compilateur gcc
CXX	le compilateur g++
CPP	le préprocesseur cpp
AS	l'assembleur as
ASFLAGS	les options de l'assembleur as
CFLAGS	les options du compilateur gcc
CXXFLAGS	les options du compilateur g++
CPPFLAGS	les options du préprocesseur cpp
LDFLAGS	les options de l'édition des liens
LOADLIBES	les librairies pour l'édition des liens

```
%.o : %.c
        gcc -g -c -o $@ $<

OBJETS = prog1.o prog2.o prog3.o

$(OBJETS) : prog.h

# etc..
```

permet de spécifier la dépendance des fichiers `prog1.o`, `prog3.o` et `prog3.o` par rapport au fichier `prog.h`. Comme il n'y a pas de commande associée à la règle, c'est la règle implicite qui sera utilisée pour créer les cibles à partir de fichiers `.c`.

6.1.6 Règles multiples

Il est aussi possible de spécifier plusieurs règles pour la même cible. Par exemple :

```
%.o : %.c
        gcc -g -c -o $@ $<

OBJETS = prog1.o prog2.o prog3.o

$(OBJETS) : prog.h
```

```
prog1.o : main.h

# etc..
```

permet de spécifier une dépendance supplémentaire pour `prog1.o` tout en utilisant la règle implicite.

6.1.7 Directives

```
VPATH =repertoire1:repertoire2:repertoire3........
```

Si les fichiers dépendants ne sont pas dans le répertoire courant, make va les chercher d'abord dans le `repertoire1`, ensuite le `repertoire2` et enfin le `repertoire3`

```
vpath =%.ext  repertoire1:repertoire2:repertoire3....
```

Idem que `VPATH` mais selon un critère de recherche.

```
vpath =%.c fichiers:fichierscpp
```

recherche les fichiers `.c` dans le répertoire courant, dans le répertoire `fichiers` et le répertoire `fichierscpp`.

6.1.8 Exécution conditionnelle

L'exécution conditionnelle, utilisée de pair avec les macros, ajoute beaucoup de flexibilité à make. Les principales directives conditionnelles : `ifdef`, `ifeq`, `ifndef` et `ifneq`, sont utilisées de pair avec `else` et `endif`. Par exemple, le fichier :

```
ifdef DEBUG
        GCCDEBUG = -g
else
        GCCDEBUG =
endif
LIBPROJ = proj1.o proj2.o proj3.o
%.o : %.c
        gcc -c $(GCCDEBUG) -o $@ $<
libproj.a : $(LIBPROJ)
        ar rc $@ $(LIBPROJ)
```

utilisé avec la commande `make` permet de compiler sans les informations de débogage alors que la commande `make DEBUG=1` permet de compiler avec les informations de débogage.

Pour plus de détails sur make, on pourra se référer à la documentation en-ligne.

6.2 La suite Autotools

Si vous avez déjà installé une application ou une librairie à partir d'un code source sous un système de type Unix, vous avez probablement eu à faire des commande de ce type :

```
$ ./configure --without-X11
$ make
$ su
% make install
% exit
```

Le script `configure` utilise le *GNU Build Systems* : Autotools. Ce script, généralement, crée un fichier `config.h` contenant des directives et des définitions qui « permettent » au code source d'utiliser de la compilation conditionnelle (les `#ifdef`, etc) pour assurer la portabilité du code entre les différentes saveurs Unix. Il crée aussi un fichier `Makefile` qui va tenir compte des options (ici, par exemple, une version qui ne requiert pas X11). Vous pouvez vous référer à ce tutoriel si vous désirez apprendre à créer vos propres fichiers de configuration. Il va sans dire que Autotools n'est pas l'outil le plus convivial et le plus simple à utiliser.

Pour nos développements, nous allons privilégier l'utilisation de CMake qui combine Make et Autotools dans un outil hautement portable (pas seulement pour les systèmes basés sur Unix).

6.3 Introduction à CMake

6.3.1 Limitations de l'analyseur de dépendances GNU make

L'utilisation de make et des `makefile` permet d'automatiser la compilation. Cependant, il faut que système de développement soit pourvu de GNU make. De plus, le programme développé doit pouvoir être compilé sur de plateformes différentes, différents paramètres de compilation devront être utilisés et les outils de développement peuvent être différents (par exemple `NMake` sous MS Windows)

et la gestion entre des librairies peut aussi différer (`*.a` vs `*.lib` et `*.so` vs `*.dll`). L'utilisation d'un IDE conjointement avec un projet utilisant un `makefile` nécessite souvent de dédoubler le fichier de définition du projet : un `makefile` et un fichier pour le projet (par exemple un fichier `*.sln` sous MS Visual C++).

CMake permet de décrire les dépendances sans spécifier les outils utilisés pour générer le programme (ou la librairie) dans un fichier `CMakeLists.txt` et son utilisation permet de générer plusieurs types de fichier :

— un `makefile` pour GNU make ;
— un `makefile` pour NMake ;
— un fichier de projet pour plusieurs IDE (Eclipse, MS Visual C++, KDevelop, Xcode, etc.).
— et autres.

6.3.2 Exemple simple d'utilisation

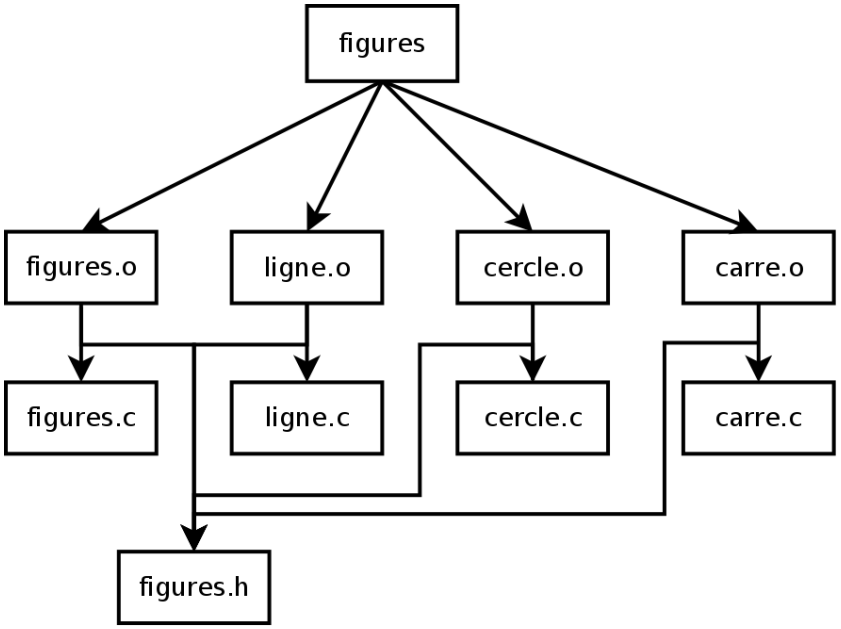


FIGURE 6.2 – Dépendances entre différents fichiers

Considérons de nouveau l'exemple de la figure 6.2 et le fichier `CMakeLists.txt`.

```
cmake_minimum_required (VERSION 2.6)
project (figures)
add_executable(figures carre.c cercle.c figures.c ligne.c)
```

La première ligne indique la version minimale de CMake requise pour le projet. La deuxième ligne nomme le projet et la troisième ligne indique que l'exécutable `figures` dépend des fichiers `carre.c`, `cercle.c`, `figures.c` et `ligne.c`.

Voyons maintenant comment utiliser ce fichier. La commande

```
$ cmake .
```

va générer, entre autres, un fichier `Makefile` qui tient compte du système et de la chaîne de compilation disponible. Ici, le `.` sert à aviser que le fichier `CMakeLists.txt` est dans le répertoire courant. Cette commande affichera de nombreux messages quant à la configuration du système. Nous sommes maintenant prêts pour la compilation

```
$ make
Scanning dependencies of target figures
[ 25%] Building C object CMakeFiles/figures.dir/carre.c.o
[ 50%] Building C object CMakeFiles/figures.dir/cercle.c.o
[ 75%] Building C object CMakeFiles/figures.dir/figures.c.o
[100%] Building C object CMakeFiles/figures.dir/ligne.c.o
Linking C executable figures
[100%] Built target figures
```

Mais comment inclure la dépendance du fichier `figures.h`. Ce n'est pas nécessaire, lors de la commande `cmake`, les fichiers en langage C sont scannés et les inclusions sont détectées automatiquement. Ce que l'on peut vérifier de la façon suivante :

```
$ touch *.h
$ make
Scanning dependencies of target figures
[ 25%] Building C object CMakeFiles/figures.dir/carre.c.o
[ 50%] Building C object CMakeFiles/figures.dir/cercle.c.o
[ 75%] Building C object CMakeFiles/figures.dir/figures.c.o
[100%] Building C object CMakeFiles/figures.dir/ligne.c.o
Linking C executable figures
[100%] Built target figures
```

Le problème avec l'approche précédente est que notre répertoire des fichiers sources devient rempli de fichiers et sous-répertoires générés par CMake. Une façon de remédier à ce problème est un *out of source built*.

```
$ mkdir build
$ cd build
$ cmake ..
$ make
Scanning dependencies of target figures
[ 25%] Building C object CMakeFiles/figures.dir/carre.c.o
[ 50%] Building C object CMakeFiles/figures.dir/cercle.c.o
[ 75%] Building C object CMakeFiles/figures.dir/figures.c.o
[100%] Building C object CMakeFiles/figures.dir/ligne.c.o
Linking C executable figures
[100%] Built target figures
```

Ici, le `..` sert à aviser que le fichier `CMakeLists.txt` est dans le répertoire parent. On peut donc effacer le répertoire `build` sans crainte de perdre des fichiers.

On peut utiliser la commande `make VERBOSE=1` qui affichera les commandes utilisées pour la compilation.

Dans le cas où nous voudrions utiliser l'environnement Eclipse pour la suite du développement de ce programme, la commande

```
cmake -G "Eclipse CDT4 - Unix Makefiles" .
```

va générer projet Eclipse et son `Makefile`. La figure 6.3 présente ce projet apres un `File->Import->General->Existing Projects into Workspace`.

On peut aussi générer un projet avec différents types de built avec l'option `CMAKE_BUILD_TYPE` (avec choix de `Debug`, `Release`, `RelWithDebInfo` ou `MinSizeRel`).

```
cmake -G "Eclipse CDT4 - Unix Makefiles" -DCMAKE_BUILD_TYPE=Debug .
```

permet de générer une version pour débogage. Pour un projet Eclipse avec contrôle de version et fonctions avancées d'édition, une configuration avec un *out of source built* sera complètement fonctionnel en ajoutant l'option
`-DCMAKE_ECLIPSE_GENERATE_SOURCE_PROJECT=TRUE`.
D'autres options sont aussi disponibles, par exemple
`-DCMAKE_ECLIPSE_MAKE_ARGUMENTS=-j4`
permet de spécifier à make d'utiliser 4 coeurs (*process*) lors de la compilation sous Eclipse.

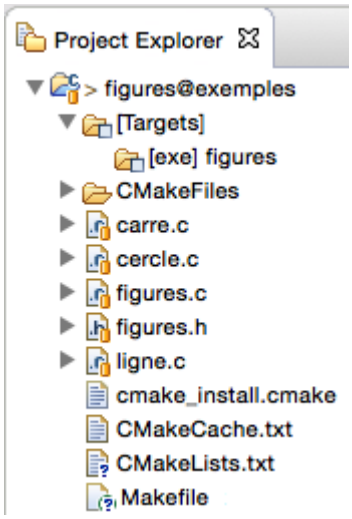


FIGURE 6.3 – Projet dans Eclipse

On utilise CMake de façon similaire pour d'autres IDE ou systèmes de compilation (liste pour la version 2.8.12, tous les générateurs ne sont pas disponibles sur toutes les plateformes) :

- Visual Studio 6, Visual Studio 7, Visual Studio 10, Visual Studio 11, Visual Studio 12, Visual Studio 7 .NET 2003, Visual Studio 8 2005, Visual Studio 9 2008,
- Borland Makefiles,
- NMake Makefiles, NMake Makefiles JOM,
- Watcom WMake,
- MSYS Makefiles, MinGW Makefiles,
- Unix Makefiles,
- Ninja,
- Xcode,
- CodeBlocks - MinGW Makefiles, CodeBlocks - NMake Makefiles, CodeBlocks - Ninja, CodeBlocks - Unix Makefiles,
- Eclipse CDT4 - MinGW Makefiles, Eclipse CDT4 - NMake Makefiles, Eclipse CDT4 - Ninja, Eclipse CDT4 - Unix Makefiles,
- KDevelop3, KDevelop3 - Unix Makefiles,
- Sublime Text 2 - MinGW Makefiles, Sublime Text 2 - NMake Makefiles, Sublime Text 2 - Ninja, Sublime Text 2 - Unix Makefiles.

6.3.3 Création d'une librairie

Pour créer sa propre librairie, il suffit d'utiliser la directive `add_library` comme suit

```
add_library(malibrairie source1.c source2.c)
```

Pour include cette librairie à la compilation d'un exécutable, il faut utiliser la directive `target_link_libraries`. Voici un exemple où le programme `monprog` requiert la librairie `malibrairie` :

```
add_executable(monprog monprog.c)
target_link_libraries(monprog  malibrairie)
```

6.3.4 Utilisation d'une librairie

Si votre projet nécessite une librairie « système » présente dans la configuration de votre environnement, la directive `find_package` après la définition de votre exécutable peut être très utile. Voici un exemple d'utilisation de la librairie libpng :

```
find_package(PNG)
if(PNG_FOUND)
  include_directories(${PNG_INCLUDE_DIRS})
  target_link_libraries(monprog ${PNG_LIBRARIES})
else(PNG_FOUND)
  message(FATAL_ERROR "Librairie PNG requise mais non disponible!")
endif(PNG_FOUND)
```

CMake contient un grand nombre de directives et d'options. Il est inutile de reproduire ici la documentation complète et les tutoriels que l'on retrouve en-ligne. Comme CMake est abondamment utilisé, il n'est pas difficile de trouver les informations qui vont nous dépanner en cas de problème. CMake permet aussi de « tester » le système cible pour créer un fichier config.h, une cible pour le `make install`, une procédure de tests et bien d'autres options.

6.3.5 Versions interactives de CMake

Selon la plateforme de développement, `ccmake` et/ou `cmake-gui` offrent la possibilités de configurer de façon interactive la génération de la « configuration » et des options utilisées.

6.4 Introduction à GIT

Cette section a pour but de vous présenter les principales fonctionnalités du système de contrôle de version distribué Git. Il n'est sûrement pas un substitut au manuel de l'utilisateur ou à l'excellent livre Pro Git.

GIT est parmi les plus récents systèmes de gestion concurrente de code source. Avec Mercurial et SVN, c'est, des systèmes non-propriétaires, un des plus utilisés.

6.4.1 Configuration

Git place son fichier de configuration global dans le fichier `~/.gitconfig` et la commande

```
$ git config --global --list
```

permet d'en afficher le contenu. Donc la première chose à faire est de configurer l'identité de l'utilisateur.

```
$ git config --global user.name "Prenom Nom"
$ git config --global user.email prenom.nom@polymtl.ca
```

De cette façon, on pourra identifier l'usager qui a modifié le code source d'un projet.

La commande

```
$ git config --list
user.name=Prenom Nom
user.email=prenom.nom@polymtl.ca
```

permet d'en afficher la configuration courante qui comprend les informations du fichier `~/.gitconfig` qui pourront être surchargées par les informations que l'on retrouve dans le fichier `.git/config` du dépôt en usage. On peut donc avoir une configuration spécifique à un projet.

6.4.2 Créer un dépot local

Pour créer notre premier dépot, allons créer un répertoire puis dans ce répertoire allons initialiser le dépot :

```
$ mkdir premiers
$ cd premiers/
$ git init
Initialized empty Git repository in /home/usager/Documents/premiers/.git/
```

Nous allons maintenant créer trois fichiers pour obtenir

```
$ ls
makefile  premiers.c  premiersprg.c
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

makefile
premiers.c
premiersprg.c

nothing added to commit but untracked files present (use "git add" to track)
```

Ok, notre dépot est vide mais trois fichiers sont présents mais pas sous le contrôle de révision. Nous ajouter ces trois fichiers.

```
$ git add *.c makefile
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

new file:   makefile
new file:   premiers.c
new file:   premiersprg.c
$ git commit -m"Version initiale"
```

```
[master (root-commit) 659c131] Version initiale
 3 files changed, 95 insertions(+)
 create mode 100644 makefile
 create mode 100644 premiers.c
 create mode 100644 premiersprg.c
$ git status
On branch master
nothing to commit, working directory clean
```

Il y a deux étapes : le `git add` et le `git commit`. La commande `git show` permet de voir les informations du dernier `commit`.

Compilons notre programme :

```
$ make
gcc -c -o premiersprg.o premiersprg.c
gcc -c -o premiers.o premiers.c
gcc -o premiersprg premiersprg.o premiers.o -lm
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

premiers.o
premiersprg
premiersprg.o

nothing added to commit but untracked files present (use "git add" to track)
```

Les fichiers résultats de la compilation apparaissent comme *Untracked*. On ne veut pas les mettre en suivi car ce sont des fichiers qui résultent de la compilation. Nous allons configurer notre dépot pour qu'il ignore ces fichiers.

```
$ echo "*.o" > .gitignore
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

.gitignore
```

```
premiersprg

nothing added to commit but untracked files present (use "git add" to track)
$ echo "premiersprg" >> .gitignore
$ more .gitignore
*.o
premiersprg
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

.gitignore

nothing added to commit but untracked files present (use "git add" to track)
```

Notre fichier `.gitignore` est maintenant complet et nous allons l'ajouter au dépot.

```
$ git add .gitignore
$ git commit -m"pas de *.o ni executable"
[master 7256e6b] pas de *.o ni executable
 1 file changed, 2 insertions(+)
 create mode 100644 .gitignore
$ git status
On branch master
nothing to commit, working directory clean
```

On peut consulter l'historique des `commits` :

```
$ git log
commit 7256e6bc1448c5c5bb7e53a078e6a26fa0f4a9c6
Author: Prenom Nom <prenom.nom@polymtl.ca>
Date:   Mon May 25 16:48:53 2015 -0400

    pas de *.o ni executable

commit 659c131ea4a0dd0c6fa1a69e0861314df8f9c2ee
Author: Prenom Nom <prenom.nom@polymtl.ca>
Date:   Mon May 25 16:32:56 2015 -0400
```

```
Version initiale
```

La section *Recording changes basics* du document Pro Git illustre bien les quatre statuts possibles pour un fichier.

La dernière opération que vous avez effectuée n'est pas ce que vous vouliez ? Selon la situation, vous pouvez faire un Undo.

6.4.3 Dépot distant

Il est possible de faire une copie d'un dépot distant hébergé sur un serveur. Dans le cas d'un Git publique, c'est très simple avec la commande `git clone [url]`. Par exemple, on veut utiliser la librairie libgd qui est hébergée chez GitHub. La commande

```
$ git clone https://github.com/libgd/libgd.git
$ cd libgd
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

va « cloner » une copie du dépot dans le répertoire `libgd` qui sera créé. On pourra faire toutes les modifications locales que l'on désire. Cependant, il ne sera pas possible de faire un `push` de nos `commit` locaux vers le serveur distant car ce dépot nous est accessible en lecture seulement. On pourra pour notre code modifié générer un *patch file* qui va contenir les différences entre le dépôt distant et notre version locale. Si les modifications apportées règlent un(des) bogue(s), nous pourrons alors soumettre le *patch file* au *maintainers* du dépôt pour inclusion dans le prochain commit.

On peut aussi spécifier le répertoire local (ici `monlibgd`) :

```
$ git clone https://github.com/libgd/libgd.git monlibdg
```

Pour un dépot distant avec des droits d'accès en écriture, la syntaxe serait plutôt la suivante :

```
$ git clone https://username@github.com/libgd/libgd.git
```

Lorsqu'on travaille avec un dépot distant, nos `commit` sont locaux. Pour mettre à jour la version sur le serveur, c'est avec un `git push` que l'on y parvient. Pour allez chercher la dernière version sur le serveur et en faire un `merge` avec notre version locale, on utilise un `git pull` (voir *Working with Remotes*).

Une recherche sur le web avec `git cheat sheet` vous mènera vers plusieurs aide-mémoires très pratiques.

6.4.4 Branches

Dans Git, votre développement n'est pas contraint à être linéaire. Vous pouvez maintenir plusieurs branches en parallèle (versions stables, de développement, expérimentales, etc.), ajouter une nouvelle branche, en fusionner deux. Le site d'aide d'Atlassian presente le concept de branches et la gestion de celles-ci.

6.4.5 GUI pour Git

Il existe plusieurs versions de GUI disponibles comme l'illustre la référence du site Git (SmartGit qui est multi-plateformes est installé au laboratoire, il est gratuit pour un usage non-commercial ou éducationnel).

6.5 Introduction à Doxygen

6.5.1 Pourquoi un générateur de documentation ?

Lors du développement d'une application, il faut commenter adéquatement notre code. Lorsque l'on travaille dans une équipe développement, c'est encore plus important. De plus, une documentation « interne » est souvent requise pour le développement et la maintenance de l'application. Un générateur de documentation est alors très utile et sauve beaucoup de temps, car le code documenté est utilisé pour créer la documentation. Ainsi, une seule étape de documentation est requise et la mise à jour du code entraine une mise à jour automatique de la documentation. Il existe plusieurs outils de documentation : Doxygen, Javadoc, QDoc, etc. Nous allons vous présenter Doxygen, un générateur multi-plateformes, multi-langages et code source ouvert. Les autres générateurs sont souvent similaires et un utilisateur de Doxygen pourra facilement s'adapter à un autre système. Ce document constitue un bref aperçu des possibilités offertes par Doxygen. La documentation complète est disponible en-ligne : www.doxygen.org.

6.5.2 Un exemple simple

Considérons le calcul d'un certain nombre de nombres premiers. On a deux fichiers sources :

```
/* premiersprg.c */
#include <stdio.h>
#include <stdlib.h>

void premiers(int n, int *pa); /* calcule n nombres premiers */

int main(int argc, char *argv[])
{
  int i, npremiers = 2000;
  int *premierstab;

  premierstab = malloc(npremiers*sizeof(int));
  premiers(npremiers, premierstab);
  if(argc >= 2) {
    /*  on imprime s'il y a un argument à l'appel */
    for(i = 0; i < npremiers; i++) {
      printf("%5d: %5d\n",i, premierstab[i]);
    }
  }
  free(premierstab);
  return 0;
}

/* premiers.c */
#include <math.h>
/* calcule n nombres premiers et
   les mémorise dans le tableau pointé par pa */
void premiers(int n, int *pa)
{
  int i = 1, j;
  int essai = 3, sqrte, estpremier;
  pa[0] = 2;
  while(i < n) {
    estpremier = 1;
    sqrte = sqrt(essai);
    for(j = 1; j < i; j++) {
      if(pa[j] > sqrte) {
        break; /* estpremier = 1 */
      } else {
        if((essai%pa[j]) == 0) {
          estpremier = 0;
```

```
            break;
          }
        }
      }
      if(estpremier) {
        pa[i++] = essai;
      }
      essai += 2;
    }
}
```

Voyons ce que Doxygen peut faire pour nous. Par défaut, Doxygen recherche un fichier `Doxyfile` dans le répertoire courant. On peut en créer un avec les paramètres standards avec la commande :

```
$ doxygen -g
```

et on peut ajuster la configuration avec l'interface graphique `doxywizard` :

```
$ doxywizard Doxyfile
```

On peut nommer notre projet : *Nombres premiers*. Choisir dans Mode (voit figure 6.4) : *All entities*, *Include cross-referenced source code in the output* et *Optimize for C* (comme nous n'avons pas encore documenté nos fichiers sources, il faut utiliser *All entities*, sinon aucune information sera documentée). Après avoir sauvé le fichier `Doxyfile`, on peut maintenant exécuter

```
$ doxygen
```

On aura un fichier `index.html` dans le répertoire `html`. Le documentation étant minimale car notre code source n'utilise pas les capacités de Doxygen (voir la figure 6.5).

Nous allons maintenant reprendre cet exemple en modifiant les sources afin d'utiliser certaines fonctionnalités de Doxygen. Choisir dans Mode (voir la figure 6.4) : *Documented entities only*

Premièrement, on utilise des commentaires Doxygen débutant avec `/*!` ou `/**` dans un des formats suivants :

```
/**
 * ... text ...
 */
```

ou

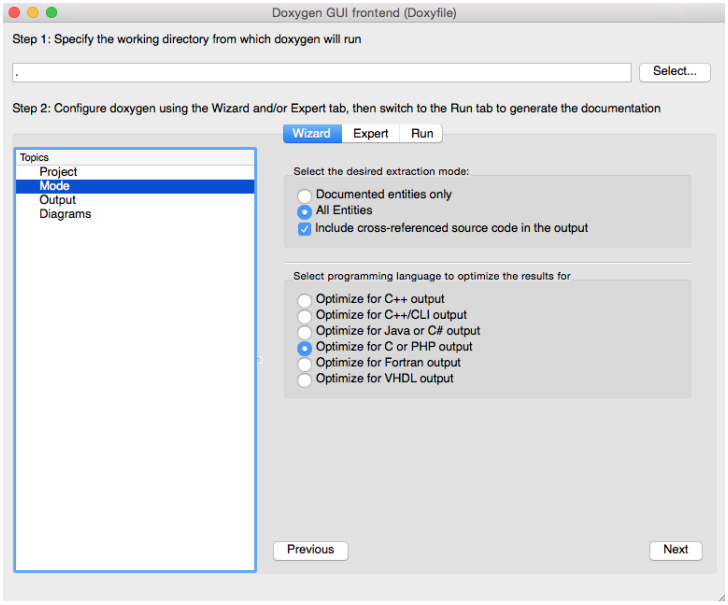


FIGURE 6.4 – Interface graphique pour Doxygen

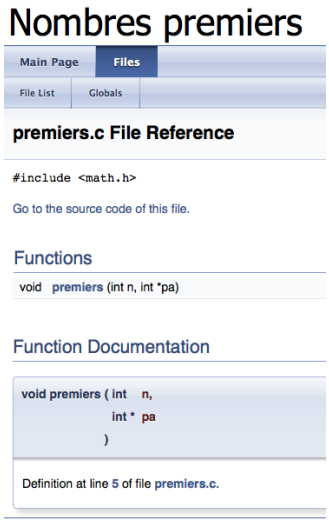


FIGURE 6.5 – Exemple de page HTML de Doxygen

```
/*!
 * ... text ...
 */
```

Le bloc contenant la « commande » `\mainpage` permet de créer la page principale (voir la figure 6.6) que l'on peut garnir dans une syntaxe similaire à LaTeX ou Markdown (en passant, toute la documentation de Doxygen est générée par Doxygen).

La commande `\file` permet de documenter les fonctions globales en C contenues dans le fichier.

La commande `\fn` permet de documenter une fonction et les `@param` après les entêtes des fonctions de spécifier les paramètres d'entrée des fonctions (voir la figure 6.7).

Voici maintenant les versions modifiées :

```
/*! \mainpage Calcul de nombres premiers
 *
 * \section algo_sec Algorithme de calcul
 *
 * Pour tester si un nombre \f$n\f$ est premier, il faut tester
 * les entiers premiers < \f$\sqrt(n)\f$ comme diviseurs.
 *
 * \section appel_sec Appel
 *
 * \subsection step1  premiersprg arg
 *
 * etc...
 */

/*! \file premiersprg.c */
#include <stdio.h>
#include <stdlib.h>

void premiers(int n, int *pa); /* calcule n nombres premiers */

/**
 * \fn main
 *
 * \brief Un appel sans argument (argc==1) calcule les nombres premiers
 *        alors qu'un appel avec un argument ou plus va imprimer les valeurs
 */
int main(int argc, char *argv[])
/**
 * @param argc le nombre d'arguments incluant le nom de l'exécutable
```

```
 * @param argv pointeur vers la liste des chaînes de caractères des arguments
 */
{
  int i, npremiers = 2000;
  int *premierstab;

  premierstab = malloc(npremiers*sizeof(int));
  premiers(npremiers, premierstab);
  if(argc >= 2) {
    /*  on imprime s'il y a un argument à l'appel */
    for(i = 0; i < npremiers; i++) {
      printf("%5d: %5d\n",i, premierstab[i]);
    }
  }
  free(premierstab);
  return 0;
}


/*! \file premiers.c */
#include <math.h>

/**
 * \fn premiers
 *
 * \brief calcule n nombres premiers et
 *        les mémorise dans le tableau pointé par pa
 */
void premiers(int n, int *pa)
/**
 * @param n le nombre de premiers à calculer.
 * @param pa pointeur vers une mémoire préalablement alouée
 *           pour n éléments int
 */
{
  int i = 1, j;
  int essai = 3, sqrte, estpremier;
  pa[0] = 2;
  while(i < n) {
    estpremier = 1;
    sqrte = sqrt(essai);
    for(j = 1; j < i; j++) {
      if(pa[j] > sqrte) {
        break; /* estpremier = 1 */
      } else {
        if((essai%pa[j]) == 0) {
```

```
            estpremier = 0;
            break;
          }
        }
      }
      if(estpremier) {
        pa[i++] = essai;
      }
      essai += 2;
    }
}
```

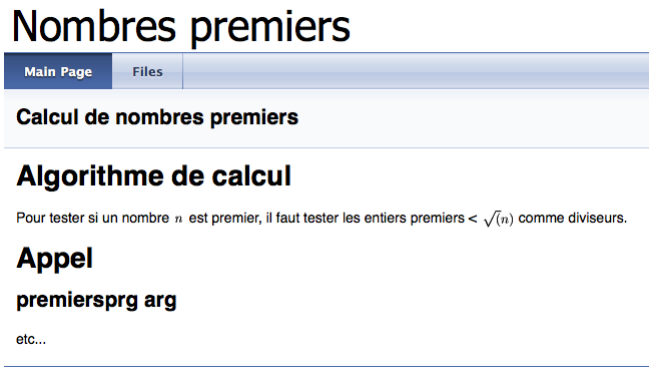


FIGURE 6.6 – Exemple de page principale HTML de Doxygen

Bien sûr, on peut documenter du langage C++, Java, etc. Voir la documentation complète www.doxygen.org pour plus de détails.

FIGURE 6.7 – Exemple de page de fonction documentée HTML de Doxygen

6.6 La syntaxe Markdown

Il est d'usage lors du développement d'une application de placer des fichiers d'information dans les répertoires du code source : `README`, `INSTALL`, ... Les sites d'hébergement de code source utilisant GIT, par exemple, afficherons les fichiers avec une extension `.md` (fichiers Markdown) avec un affichage enrichi. Les pages wiki du site SourceForge utilise aussi la syntaxe Markdown. Les versions récentes Doxygen (`>=1.8.0`) permettent aussi cette syntaxe.

La syntaxe Markdown permet d'avoir un fichier texte lisible par lui même mais qui sera trasformé en html pour meilleure lisibilité. Il suffira dans certains contextes d'utiliser un fichier avec l'extension `.md` pour obtenir une « page » html. Pour apprendre la syntaxe, on peut utiliser une page de pré-visualisation.

Chapitre 7

L'IDE Eclipse-CDT

Eclipse, développé au départ par IBM (sous le nom de VisualAge Micro Edition) puis devenu en code source ouvert, est un environnement intégré de développement (IDE) écrit en Java à l'origine pour la programmation en Java. Son système d'extensions (*plugins*) a permis de supporter de nombreux langages de programmation dont le C et le C++ dans un *toolkit* nommé CDT (développé à l"origine par QNX sous le nom de Momentics). On peut télécharger différentes « saveurs » de l'IDE dont Eclipse-CDT qui est portable (Linux, Mac OS X et MS Windows) étant écrit en Java.

En fonction de la plateforme utilisée, Eclipse-CDT pourra faire une compilation native, croisée (configuration manuelle et automatique par CMake), exécuter le programme sous le contrôle de GDB pour le débogage local ou distant (via GDB-remote). L'auto-indentation, l'auto-complétion des noms de variables ou de fonctions, l'aide à la documentation, la génération de `makefile`, la gestion des configurations (Release, Debug, ...) sont quelques unes des nombreuses fonctionnalités disponibles. Si on connait bien Eclipse, on a un seul IDE à apprendre, quelque soit la plateforme choisie. Nous allons présenter les principales fonctionnalités. La documentation en-ligne est très complète, le Wiki et la FAQ.

7.1 Projets de compilation native

L'IDE Eclipse ne comprend de compilateur C/C++, mais en fonction de la plateforme sur laquelle il est installé, il permettra de créer des projets en fonction de compilateurs disponibles. La fonction `File->New` (figure 7.1) permet de créer un nouveau projet, ce convertir un projet, ou d'ajouter des ressources, fichiers

sources, etc. En créant un nouveau projet, il pourra être natif en sélectionnant le compilateur disponible sur la plateforme et le type de programme. Eclipse pourra utiliser un `makefile` existant ou en crée un à partir des éléments inclus dans l'arborescence du projet.

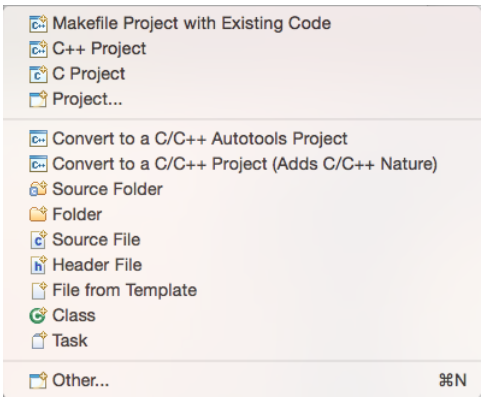


FIGURE 7.1 – `File->New` sous Eclipse

7.2 L'éditeur de code

Eclipse-CDT facilite l'édition des fichiers sources en langages C et C++. Nous allons illustrer quelques unes des fonctionnalités disponibles (il y en a beaucoup plus et la documentation disponible vous aidera à en découvrir d'autres). Nous allons démarrer avec un *Hello World* natif (figure 7.2).

Nous allons ajouter une fonction `test` au code (figure 7.3). On remarque que l'analyseur de code a détecté un problème. Si on place la souris au dessus du « bug », un message nous indique que la fonction n'est pas `void` mais doit retourner un `int`. On ajoute un `return` pour régler le problème.

Nous allons aussi ajouter une boucle `for`. Nous allons taper `for` suivi d'un `ctrl-space` pour obtenir les suggestions illustrées à la figure 7.4 et il ne reste qu'à compléter notre boucle. On peut aussi entrer un début de commentaire `/*` suivi d'un `return`. L'utilisation de `ctrl-space` permet de compléter les structures de contrôle du langage (`do`, `while`, ...), les noms de fonctions, de méthodes, de variables, etc.

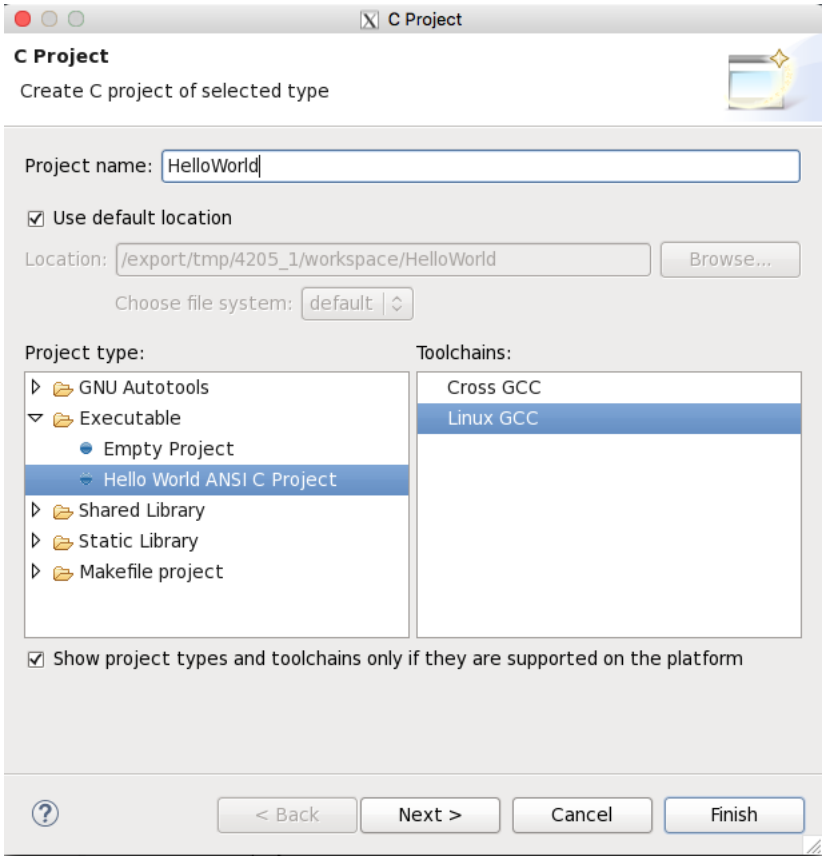


FIGURE 7.2 – `File->New->C Project` sous Eclipse

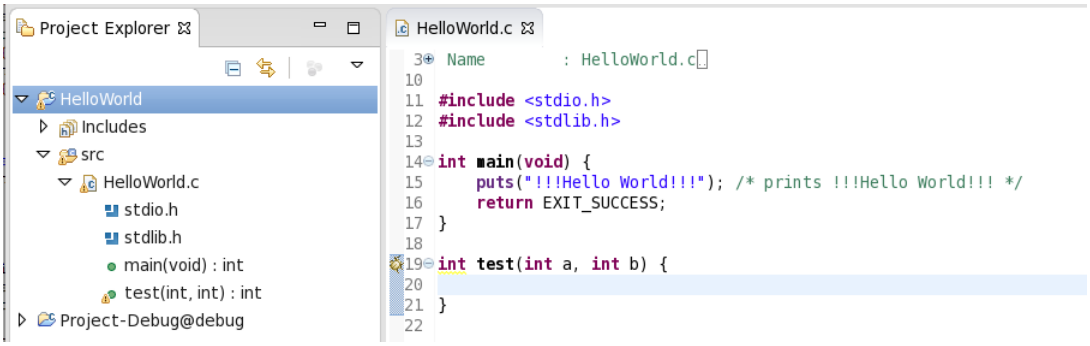


FIGURE 7.3 – Nouvelle fonction incomplète

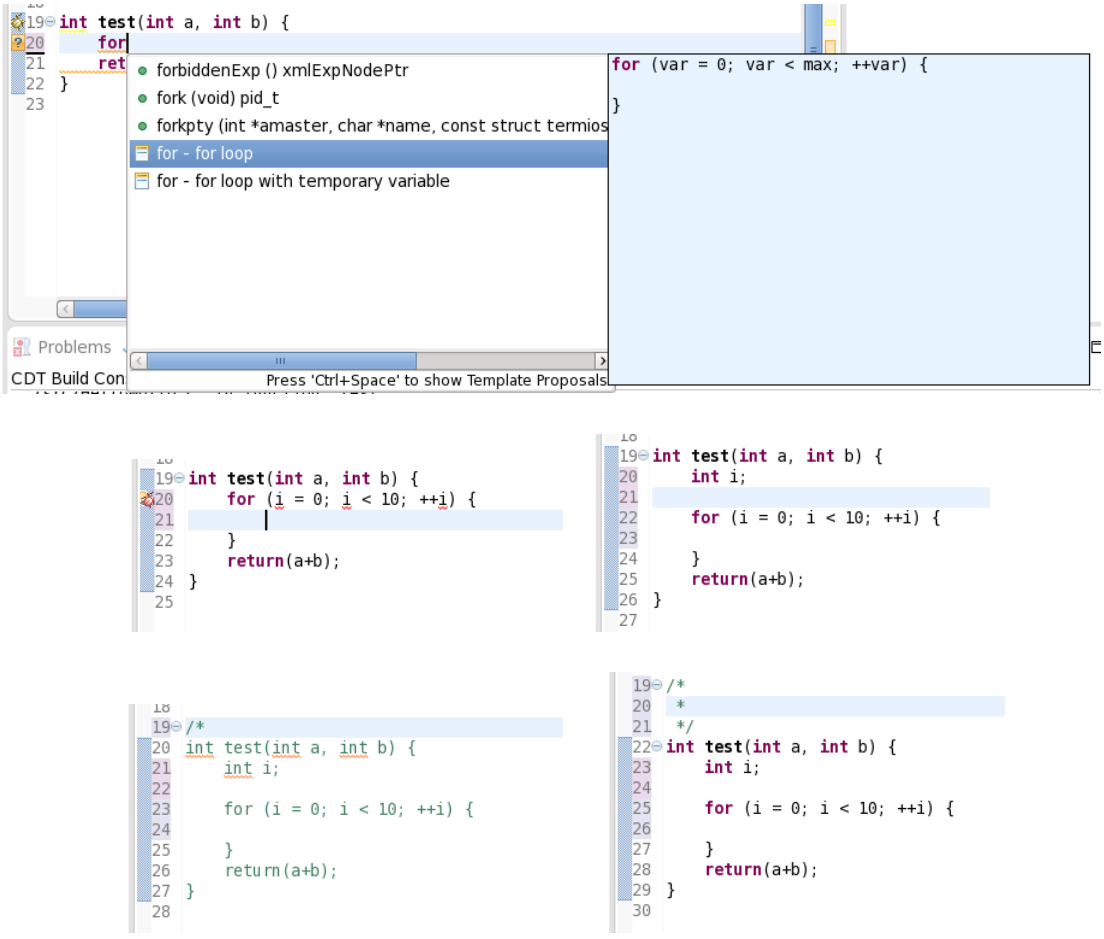


FIGURE 7.4 – Ajout d'un `for` et d'un commentaire

7.2.1 Intégration avec Doxygen

Comme Eclipse permet d'ajouter des commentaires facilement, il serait intéressant que ces commentaires soient « pré-formatés » pour Doxygen. Heureusement, Eclipse comprend déjà cette fonctionnalité.

Il faut activer les commentaires automatiques avec `Windows->Preferences->C/C++->Editor->Workspace default:` à `Doxygen` comme illustré à la figure 7.5.

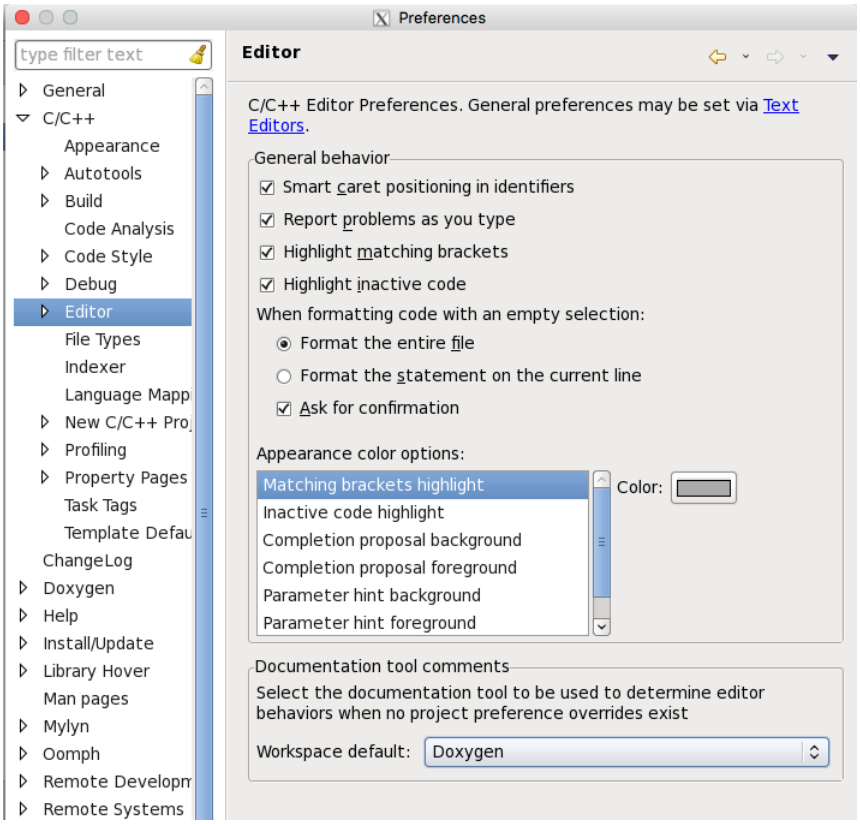
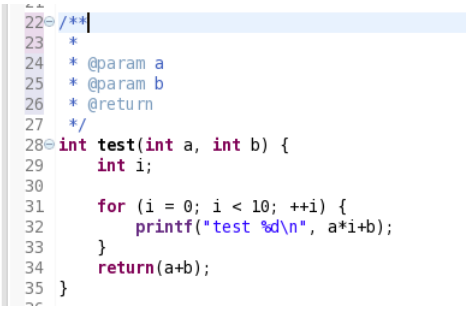


FIGURE 7.5 – Préférences pour *plugin* Eclox.

On peut maintenant utiliser `/**` suivi de `return` pour avoir des commentaires Doxygen comme illustré à la figure 7.6.



```
/**
 *
 * @param a
 * @param b
 * @return
 */
int test(int a, int b) {
    int i;

    for (i = 0; i < 10; ++i) {
        printf("test %d\n", a*i+b);
    }
    return(a+b);
}
```

FIGURE 7.6 – Commentaires avec fonctionnalités Doxygen dans Eclipse.

7.3 Projets de compilation croisée

Lors de la création d'un nouveau projet, on pourra sélectionner un projet de compilation croisée. Il faudra spécifier le préfixe du compilateur ainsi que le chemin (`PATH`) pour les exécutables de compilation croisé.

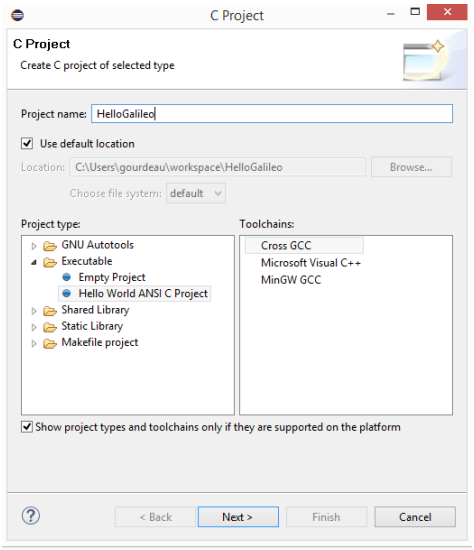
7.3.1 Exemple : compilation croisée pour le Galileo d'Intel

Voici un exemple avec un SDK pour la plateforme Galileo d'Intel pour une compilation sous Windows.

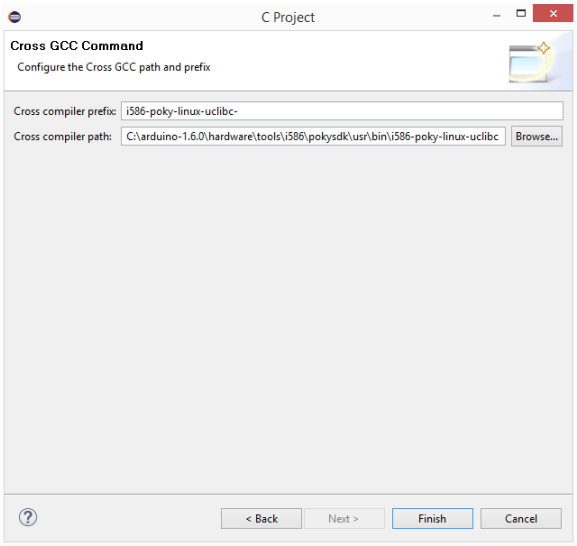
Pour utiliser Eclipse sous MS Windows, il faut d'abord installer MinGW afin d'avoir accès à Make. Après installation, il faut s'assurer que `mingw32-make` est dans le `PATH`.

Puis, il faut installer Intel Galileo Arduino SW 1.6.0 on Windows. Les instructions qui suivent considèrent que l'application a été installée dans le répertoire `C:\arduino-1.6.0\`.

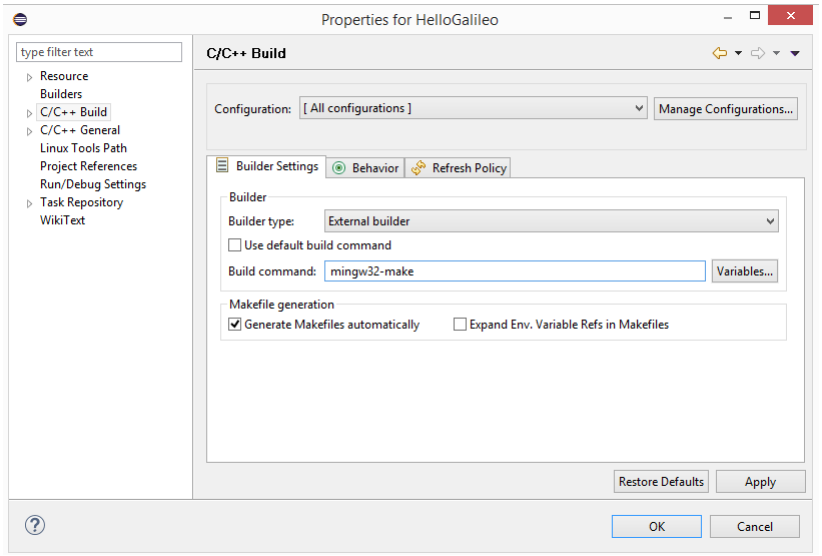
Après avoir démarrer Eclipse, nous allons créer un nouveau projet (`File->New->C Project`) *Cross GCC* :



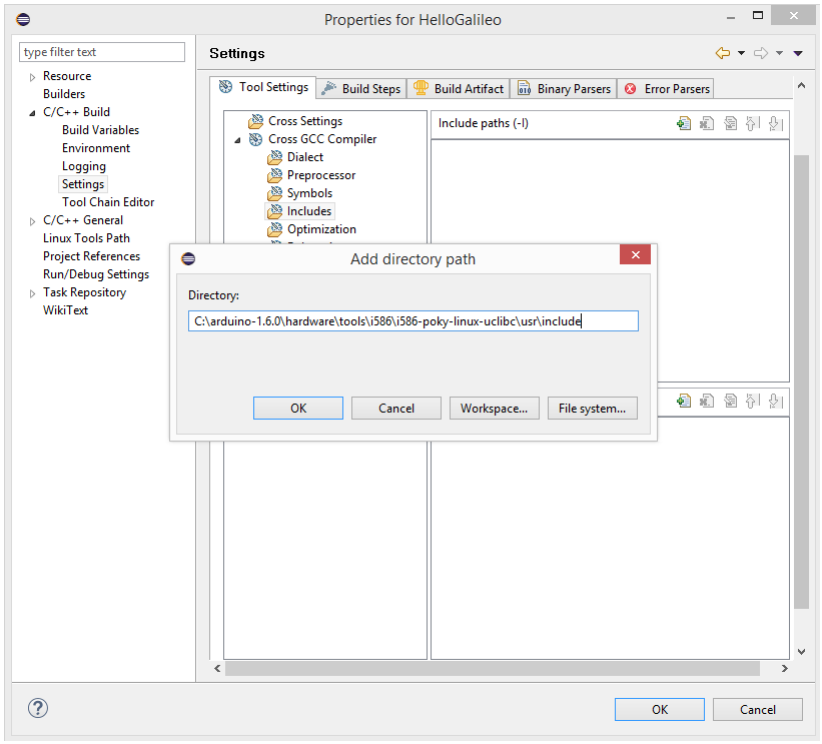
avec `i586-poky-linux-uclibc-` et `C:\arduino-1.6.0\hardware\tools\i586\pokysdk\usr\bin\i586-poky-linux-uclibc` comme *Cross compiler prefix* et *Cross compiler path* :



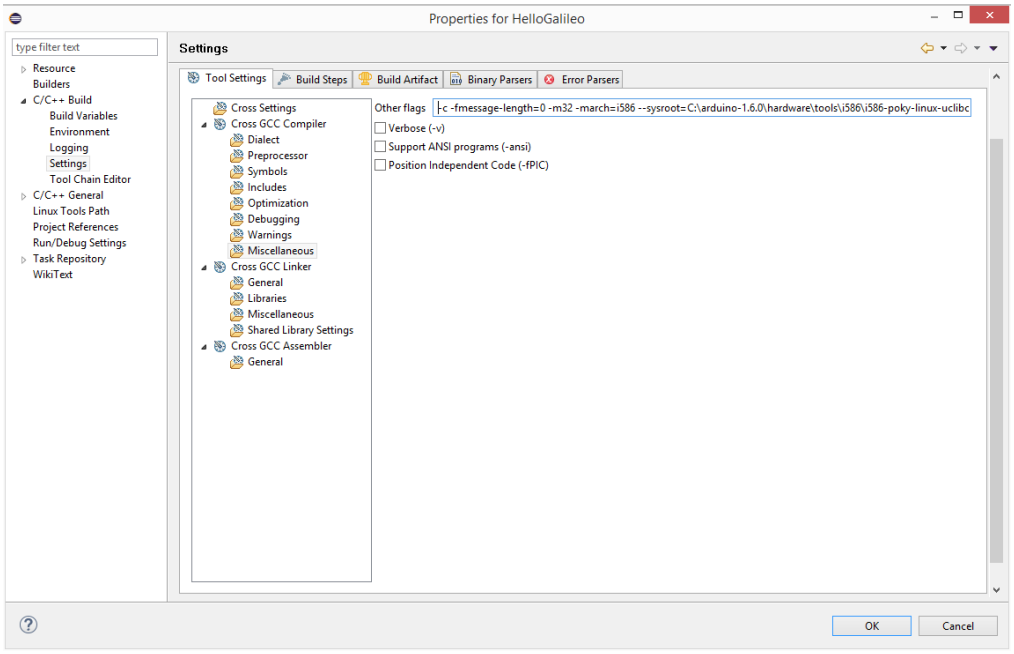
puis pour `[All configrations]` choisir le programme pour le makefile : `mingw32-make`

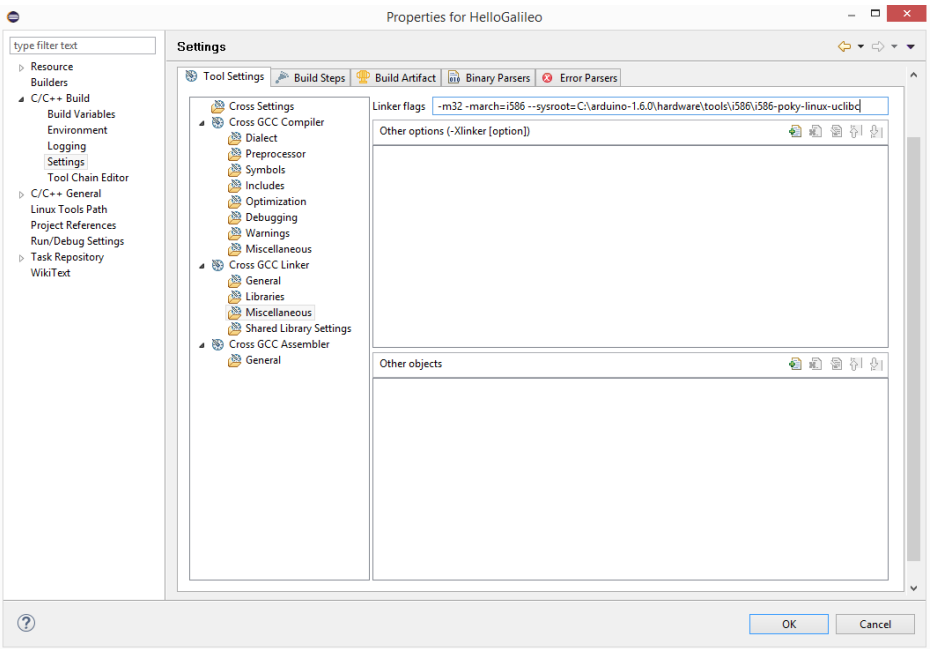


l'ajout dans *Cross GCC Compiler* - *Includes* de `C:\arduino-1.6.0\hardware\tools\i586\i586-poky-linux-uclibc\usr\include`



et l'ajout des options
`-m32 -march=i586 --sysroot=C:\arduino-1.6.0\hardware\tools\i586\i586-poky-linux-uclibc`
pour la compilation et l'édition des liens (configuration processeur et *sysroot*)





On peut maintenant compiler l'application qui pourrait être transférée vers la cible pour être exécutée. On peut suivre une procédure similaire pour d'autres plateformes cibles avec leur SDK.

Au laboratoire vous avez utilisé CMake pour configurer votre compilation croisée, mais nous aurions aussi pu le faire manuellement de façon similaire à ce qui a été présenté précédemment.

Intel fournit un Intel® System Studio IoT Edition qui est un Eclipse préconfiguré pour le Galileo avec le débogage distant. Vous savez comment faire la même chose « manuellement ».

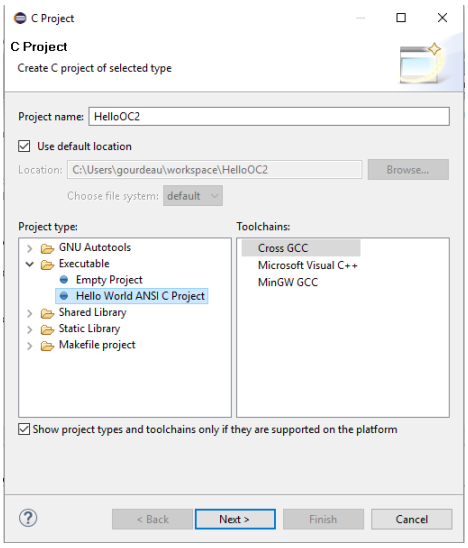
7.3.2 Exemple : compilation croisée pour le Odroid-C2

Voici un exemple avec un SDK pour la plateforme Odroid-C2.

Pour utiliser Eclipse sous MS Windows, il faut d'abord installer MinGW afin d'avoir accès à Make. Après installation, il faut s'assurer que `mingw32-make` est dans le `PATH`.

Puis, il faut installer le SDK Odroid-C2. Ici nous avons extrait l'archive dans `D:/`. Ajustez les instructions suivantes en fonction de l'endroit où vous avez installé le SDK.

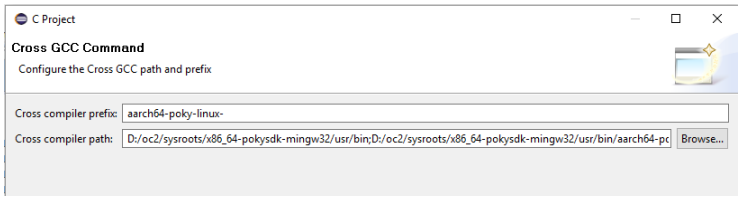
Après avoir démarrer Eclipse, nous allons créer un nouveau projet (`File->New->C Project`) *Cross GCC* :



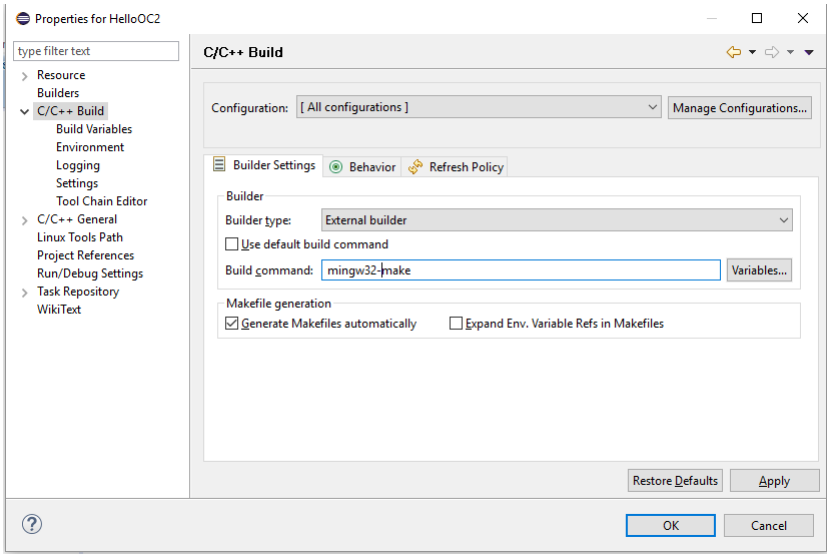
avec `aarch64-poky-linux-` et

```
D:/oc2/sysroots/x86_64-pokysdk-mingw32/usr/bin;
D:/oc2/sysroots/x86_64-pokysdk-mingw32/usr/bin/aarch64-poky-linux;
D:/oc2/sysroots/x86_64-pokysdk-mingw32/usr/bin/aarch64-poky-linux-uclibc;
D:/oc2/sysroots/x86_64-pokysdk-mingw32/usr/bin/aarch64-poky-linux-musl
```

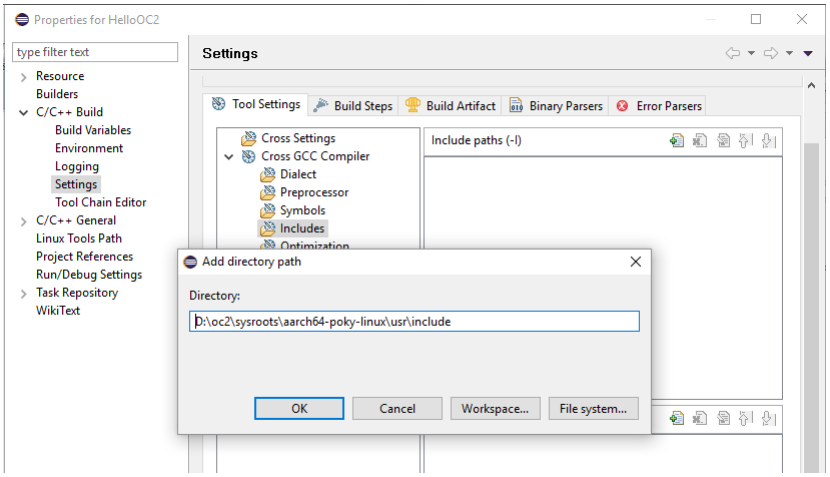
comme *Cross compiler prefix* et *Cross compiler path* :



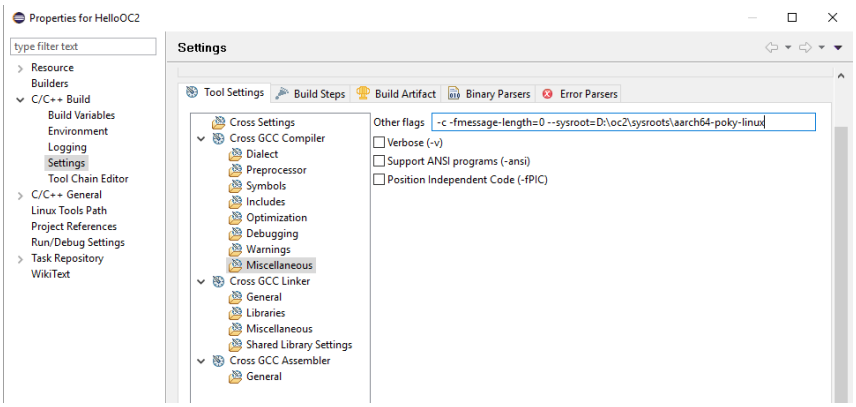
puis pour `[All configurations]` choisir le programme pour le makefile : `mingw32-make`



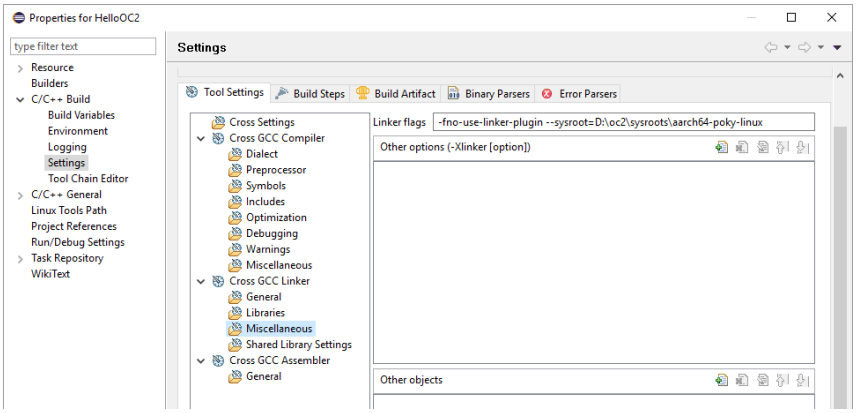
l'ajout dans *Cross GCC Compiler - Includes* de
`D:\oc2\sysroots\aarch64-poky-linux\usr\include`



et l'ajout des options
`--sysroot=D:\oc2\sysroots\aarch64-poky-linux` pour la compilation (*sysroot*)



`-fno-use-linker-plugin --sysroot=D:\oc2\sysroots\aarch64-poky-linux` pour l'édition des liens (*sysroot* et option d'édition des liens)



On peut maintenant compiler l'application qui pourrait être transférée vers la cible pour être exécutée. On peut suivre une procédure similaire pour d'autres plateformes cibles avec leur SDK.

Au laboratoire vous avez utilisé CMake pour configurer votre compilation croisée, mais nous aurions aussi pu le faire manuellement de façon similaire à ce qui a été présenté précédemment.

Pour configurer le USB-gadget sous Windows utilisez la procédure suivante.

De la même façon, un débogage distant pourrait être configuré (procédure identique, sauf pour le début des noms de répertoire, à celle du laboratoire 3 – section 4.2).

Chapitre 8

Le système de meta-distribution Yocto

La configuration et la compilation d'un système Linux complet est une tâche longue et complexe. Il faut premièrement générer un compilateur (possiblement croisé) pour le système cible. Le site *Linux From Scratch*, par exemple, documente comment le faire en compilation native pour un système minimal (*Linux From Scratch*), un système plus complet (*Beyond Linux From Scratch*) ou en compilation croisée (*Cross Linux From Scratch*). En consultant ces instructions, il devient évident que pour supporter des cibles différentes pour une même application, une méthode automatisée de *build* du système s'avère nécessaire pour avoir un flot de travail efficace et reproductible (pour le travail en équipe et le déploiement). En ce moment, les deux principaux systèmes de *build* utilisés sont Buildroot et Yocto Project.

Buildroot est un système simple pour construire un système minimal. Il est utilisé par OpenWRT et autres projets.

Dans le cadre du cours nous allons nous limiter au système utilisé par le Yocto Project. Celui-ci est basé sur l'architecture d'OpenEmbedded et BitBake. Bien que plus complexe que Buildroot, ce système est plus flexible est supporté par les joueurs majeurs de l'industrie au travers de la Linux Foundation.

8.1 Architecture du Yocto Project

L'architecture de de l'environnement Yocto est présentée à la figure 8.1.

— Cet environnement comprend un ensemble d'outils et de méthodes permettant l'évaluation rapide de systèmes Linux embarqués sur de nombreuses cibles et une configuration facile des caractéristiques de de la « distribu-

tion » ;
— Supporte plusieurs architectures : x86, ARM, MIPS, PowerPC, etc ;
— Basé sur le OpenEmbedded Project ;
— Architecture par couches permettant la réutilisation du code ;
— Supporte plusieurs format de distribution de *packages* (`rpm, dep, ipk`) ;
— Un cycle de 6 mois de *releases* ;
— Basé sur les versions récentes et stables du noyau, de chaînes de développement, etc ;
— Fournit des outils de développement d'applications (plugins Eclipse, `ADT`, hob).

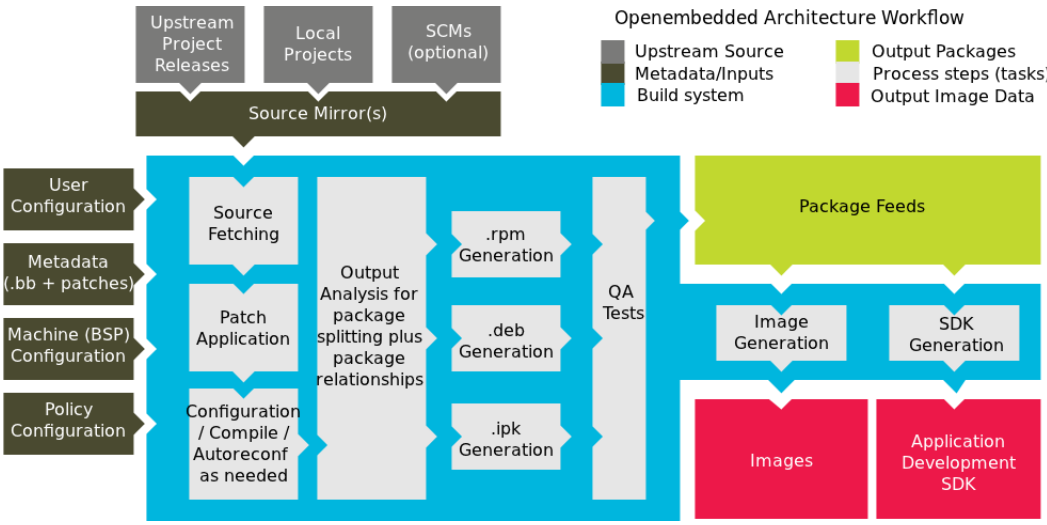


FIGURE 8.1 – Environnement Yocto

Le projet utilise BitBake un engin de *build* qui détermine les dépendances et planifie l'ordre dans lequel les tâches doit être exécutées. BitBake utilise des Metadata représentant une collection de règles et de recettes organisées en couches (layers) permettant de construire le système.

Le document Yocto Project and OpenEmbedded Training constitue une bonne référence pour comprendre la fonction du système de même que la présentation d'introduction du *Developer Day 2015*. Le vidéo *Getting Started with the Yocto Project* permet aussi de se familiariser avec le système.

8.2 Éléments de base

Le projet Yocto utilise une architecture sur forme de couches (*layers*, voir figure 8.2).

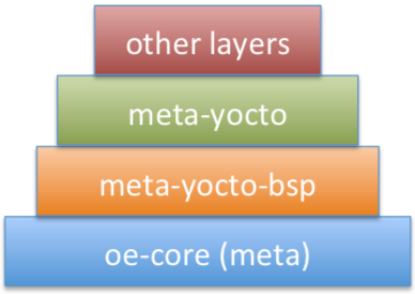


FIGURE 8.2 – Couches sous Yocto

La distribution de référence de Yocto est Poky et elle contient les couches suivantes :

- `oe-core (poky/meta)` : le coeur d'*OpenEmbedded* ;
- `meta-yocto-bsp` : les *Board Specification Packages* (définitions des règles et paramètres associés à différentes architectures et matériels « standards ») ;
- `meta-yocto` : les caractéristiques et règles de la distribution ;
- Autres couches : matériels non inclus dans `meta-yocto-bsp`, librairies et programmes supplémentaires, applications, ...

Les couches contiennent des meta-données (*metadata*) qui appartiennent à quatre catégories :

- Recettes (*recipes*) (fichier `*.bb`) : décrit normalement les instructions pour générer un *package* (*build instructions*) ;
- Groupes de *packages* (fichier *PackageGroups* (`*.bb` spéciaux) : utilisés le plus souvent pour regrouper des *packages* ensembles pour le système de fichiers de l'image à créer.
- Classes (fichier `*.bbclass`) : mécanisme d'héritage pour des fonctionnalités et règles générales (un pau comme les classes abstraites en C++ qui sont utilisées dans des objects ou classes dérivées). Elles sont souvent utilisées, mais l'utilisateur en développe peu.
- Configuration (fichier `*.conf`) : dirige le comportement général du processus de *build*.

Notons qu'il y a aussi des fichiers `*.bbappend` qui permettent des ajouts à une recette portant le même nom (e.g.`qemu_2.2.0.bb` et `qemu_2.2.0.bbappend`).

8.2.1 Recettes (fichiers *.bb)

Nous allons présenté les éléments composant les recettes à l'aide de quelques exemples.

Considérons la librairie Ogg version 1.3.2 décrite par le fichier `libogg_1.3.2.bb` (on remarque le format `packagename_version.bb`) :

```
SUMMARY = "Ogg bitstream and framing libary"
DESCRIPTION = "libogg is the bitstream and framing library \
for the Ogg project. It provides functions which are \
necessary to codec libraries like libvorbis."
HOMEPAGE = "http://xiph.org/"
BUGTRACKER = "https://trac.xiph.org/newticket"
SECTION = "libs"
LICENSE = "BSD"
LIC_FILES_CHKSUM = "file://COPYING;md5=db1b7a668b2a6f47b2af88fb008ad555 \
                    file://include/ogg/ogg.h;beginline=1;endline=11;md5=eda812856f13a3b1326eb8

SRC_URI = "http://downloads.xiph.org/releases/ogg/${BP}.tar.xz"

SRC_URI[md5sum] = "5c3a34309d8b98640827e5d0991a4015"
SRC_URI[sha256sum] = "3f687ccdd5ac8b52d76328fbbfebc70c459a40ea891dbf3dccb74a210826e79b"

inherit autotools pkgconfig
```

qui comprend les éléments suivants :

- `SUMMARY` : Description abrégée du *package* ;
- `DESCRIPTION` : Description détaillée du *package* ;
- `HOMEPAGE` : URL du site web de référence ;
- `BUGTRACKER` : URL du site web de suivi des bugs ;
- `SECTION` : Catégorie du *package* ;
- `LICENSE` : Type(s) de license ;
- `LIC_FILES_CHKSUM` : Nom(s) du(des) fichier(s) de license et leur(s) *checksum(s)* ;
- `SRC_URI` : Localisation des fichiers source, ici `${BP}` le nom de base de la recette sans suffixe (`${BPN}-${PV}` : nom de base du *package* `${BPN}` et numéro de version `${PV}`), de plus des fichiers `*.patch` dans la liste seront automatiquement traités comme des correctifs (*patches*) aux fichiers sources ;

`SRC_URI[md5sum] et SRC_URI[sha256sum]` : MD5 et SHA256 *hashes* du *tarball* qui permettent de vérifier l'intégrité du téléchargement.

Finalement, `inherit autotools pkgconfig` spécifie que la configuration hérite des règles d'Autotools et de `pkg-config` pour le *build* qui comprend minimalement les éléments suivants (et d'autres que nous ne mentionnons pas ici) :

- `do_fetch` : localise et télécharge le code source ;
- `do_unpack` : extrait le code source ;
- `do_patch` : applique des *patches* si requis ;
- `do_configure` : configure le *package* (ici avec Autotools) ;
- `do_compile` : compile, normalement, en appelant Make ;
- `do_install` : installe le résultat de la compilation dans le répertoire donné par `WORKDIR` (souvent un `make install`) ;
- `do_populate_sysroot` : comme `do_install` mais dans le système de fichiers de la cible ;
- `do_package_*` : crée un packetage binaire (`rpm, dep, ipk`).

La commande

```
$ bitbake -c listtasks <recipe_name>
```

permet de lister toutes les taches disponibles pour une recette.

Considérons la librairie PNG version 1.6.21 décrite par le fichier `libpng_1.6.21` :

```
SUMMARY = "PNG image format decoding library"
HOMEPAGE = "http://www.libpng.org/"
SECTION = "libs"
LICENSE = "Libpng"
LIC_FILES_CHKSUM = "file://LICENSE;md5=06a1b6fde6d93170bb72201c8000bf3d \
                    file://png.h;endline=112;md5=9a8b5f83e1e8046df672deab87f235be"
DEPENDS = "zlib"

LIBV = "16"

SRC_URI = "${SOURCEFORGE_MIRROR}/project/libpng/libpng${LIBV}/older-releases/${PV}/libpng-${PV}.tar.xz
          "
SRC_URI[md5sum] = "3bacb4728f6694a64ad9052769d6a4ce"
SRC_URI[sha256sum] = "6c8f1849eb9264219bf5d703601e5abe92a58651ecae927a03d1a1aa15ee2083"

BINCONFIG = "${bindir}/libpng-config ${bindir}/libpng16-config"

inherit autotools binconfig-disabled pkgconfig
```

```
# Work around missing symbols
EXTRA_OECONF_append_class-target = " ${@bb.utils.contains("TUNE_FEATURES", "neon", "--enable-arm-neon=

PACKAGES =+ "${PN}-tools"

FILES_${PN}-tools = "${bindir}/png-fix-itxt ${bindir}/pngfix"

BBCLASSEXTEND = "native nativesdk"
```

qui comprend les nouveaux éléments suivants :

`DEPENDS` : Dépendance du *package* (ici sur la libraire `zlib`) ;

`SOURCEFORGE_MIRROR` : Utilise un système de miroirs pour le téléchargement ;

`BINCONFIG` : Variable qui point sur les scripts lorsque `binconfig-disabled` est hérité ;

`PACKAGES` : Ajoute un *package* (`${PN}-tools` = `libpng-tools`) au *package* de base (`libpng`) ;

`FILES_${PN}-tools` : Liste les fichiers à mettre dans le *package* `${PN}-tools` ;

`BBCLASSEXTEND` : Spécifie que la recette qui a été initialement définie pour la cible (*target*) est aussi bonne pour l'hôte (`native` -> *host*) et le SDK (`nativesdk`).

Comme dernier exemple, considérons `flac_1.3.1.bb` (FLAC version 1.3.1) :

```
SUMMARY = "Free Lossless Audio Codec"
DESCRIPTION = "FLAC stands for Free Lossless Audio Codec, a lossless audio compression format."
HOMEPAGE = "https://xiph.org/flac/"
BUGTRACKER = "http://sourceforge.net/p/flac/bugs/"
SECTION = "libs"
LICENSE = "GFDL-1.2 & GPLv2+ & LGPLv2.1+ & BSD"
LIC_FILES_CHKSUM = "file://COPYING.FDL;md5=ad1419ecc56e060eccf8184a87c4285f \
                    file://src/Makefile.am;beginline=1;endline=17;md5=0a853b81d9d43d8aad3b53b05cfcc37e
                    file://COPYING.GPL;md5=b234ee4d69f5fce4486a80fdaf4a4263 \
                    file://src/flac/main.c;beginline=1;endline=18;md5=d03a766558d233f9cc3ac5dfafd49deb
                    file://COPYING.LGPL;md5=fbc093901857fcd118f065f900982c24 \
                    file://src/plugin_common/all.h;beginline=1;endline=18;md5=7c8a3b9e1e66ed0aba765bc6
                    file://COPYING.Xiph;md5=a2c4b71c0198682376d483eb5bcc9197 \
                    file://include/FLAC/all.h;beginline=65;endline=70;md5=64474f2b22e9e77b28d8b8b25c98
DEPENDS = "libogg"

SRC_URI = "http://downloads.xiph.org/releases/flac/${BP}.tar.xz"
```

```
SRC_URI[md5sum] = "b9922c9a0378c88d3e901b234f852698"
SRC_URI[sha256sum] = "4773c0099dba767d963fd92143263be338c48702172e8754b9bc5103efe1c56c"

inherit autotools gettext

EXTRA_OECONF = "--disable-oggtest \
                --with-ogg-libraries=${STAGING_LIBDIR} \
                --with-ogg-includes=${STAGING_INCDIR} \
                --disable-xmms-plugin \
                --without-libiconv-prefix \
                ac_cv_prog_NASM="" \
                "

EXTRA_OECONF += "${@bb.utils.contains("TUNE_FEATURES", "altivec", " --enable-altivec", " --disable-alt
EXTRA_OECONF += "${@bb.utils.contains("TUNE_FEATURES", "core2", " --enable-sse", "", d)}"
EXTRA_OECONF += "${@bb.utils.contains("TUNE_FEATURES", "corei7", " --enable-sse", "", d)}"

PACKAGES += "libflac libflac++ liboggflac liboggflac++"
FILES_${PN} = "${bindir}/*"
FILES_libflac = "${libdir}/libFLAC.so.*"
FILES_libflac++ = "${libdir}/libFLAC++.so.*"
FILES_liboggflac = "${libdir}/libOggFLAC.so.*"
FILES_liboggflac++ = "${libdir}/libOggFLAC++.so.*"
```

Ici un seul nouvel élément, `EXTRA_OECONF` qui spécifie les options additionnelles pour le `configure` et des paramètres spécifiques à GCC pour certaines architectures.

8.2.2 Configuration (fichier `*.conf`)

Ce sont les fichiers qui définissent le *build*. Les deux plus importants sont `bblayers.conf` et `local.conf`.

`bblayers.conf` permet de lister toutes les couches qui seront utilisées pour la création de notre image.

`local.conf` permet de définir la cible (`MACHINE`), les répertoires par défaut (`DL_DIR`), les éléments à ajouter à l'image soit pour l'exécution ou le débogage (`IMAGE_INSTALL_append`, `EXTRA_IMAGE_FEATURES`, etc) et autres. Vous avez pu expérimenter avec ces fichiers au laboratoire.

8.3 Création d'images et de SDK

Une fois l'environnement configuré, vous avez vu au laboratoire qu'il est simple de créer une image et un SDK pour votre système avec les commandes :

```
$ nice bitbake core-image-base
$ bitbake -c populate_sdk core-image-base
```

pour un système de base ou

```
$ nice bitbake core-image-minimal
$ bitbake -c populate_sdk core-image-minimal
```

pour un système minimal[1].

Le SDK ainsi obtenu est pour la compilation sur système similaire à l'ordinateur qui le génère (au laboratoire `x86_64`) car la ligne suivant est commentée par défaut :

```
#SDKMACHINE ?= "i686"
```

Quelles sont les choix possibles pour `SDKMACHINE` ? Pour une installation « standard », `i686` (32 bits) et `x86_64` (64 bits).

Cependant, il existe une couche (*Layer*) qui permet de compiler un SDK pour Microsoft Windows. On se rappelle que la couche `meta-openembedded/meta-oe` a été ajoutée pour avoir OpenCV et autres *packages* supplémentaires au laboratoire.

Si l'on veut obtenir un compilateur croisé pour Microsoft Windows vers notre cible, il faut d'abord avoir un compilateur de Linux vers Microsoft Windows. Sous Ubuntu, il suffit d'installer `mingw-w64` (`GCC` pour cibles Microsoft Windows) qui, avec ses dépendances, va installer des compilateurs pour les architectures 32 bits et 64 bits de Microsoft Windows. Il suffit donc d'utiliser la commande :

```
sudo apt-get install mingw-w64
```

Il nous faut maintenant les recettes pour batir un *SDK* sous Linux pour la compilation sous Microsoft Windows d'exécutables pour la cible (Odroid-C2 par exemple).

La couche `meta-mingw` permet de choisir comme valeur de `SDKMACHINE` les choix supplémentaires suivants : `i686-mingw32` et `x86_64-mingw32`.

Il suffit d'utiliser GIT pour obtenir la même branche que notre version de `poky` et d'ajouter ce *meta* dans le fichier `bblayers.conf` d'un nouveau répertoire de

1. On peut lister les images disponibles avec la commande `bitbake -s | grep core-image`

build (chaque SDK a besoin de son propre répertoire de *build*). Pour un compilateur 64 bits, on ajoute `SDKMACHINE ?= "x86_64-mingw32"` dans notre fichier `local.conf`. Par exemple, la commande

```
$ bitbake -c populate_sdk core-image-base
```

va générer dans le repertoire `tmp/deploy/sdk` un fichier `tar,bz2` contenant le SDK.

Attention, ce fichier compressé contient des liens symboliques qui ne sont pas supportés par Microsoft Windows. Il faut donc decompresser ce fichier dans un répertoire temporaire et le re-compresser avec `zip` qui remplace les liens par une copie des fichiers. C'est cette archive que l'on distribue. Une autre façon de faire est de placer le SDK sur un disque réseau Samba partagé. Ainsi, toute le groupe de développement aura accès au SDK.

8.4 Commandes pratiques à connaitre

Commande `bibake`	Description
`bitbake <image>`	Génère une image (ajouter l'option `-k` pour continuer même après erreur(s)).
`bitbake <package> -c <task>`	Exécute une tâche particulière. Exemple `bitbake busybox -c fetch` télécharge les fichiers requis.
`bitbake <package> -g -u depexp`	Affiche les dépendances pour `package`.
`bitbake <package> -c listtasks`	Liste toutes les tâches pour `package`.
`bitbake <image> -c fetchall`	Télécharge tous les fichiers requis pour `image`
`bitbake-layers show-layers`	Affiche toutes les couches
`bitbake-layers show-recipes "*-image-*"`	Affiche les recettes avec `-image-`
`bitbake-layers show-recipes`	Affiche toutes les recettes disponibles
`bitbake virtual/kernel -c menuconfig`	Configuration ineractive du noyau Linux
`bitbake -s | grep <package>`	Vérifie la présence de `package`
`bitbake -s`	Liste tout les `packages`

Chapitre 9

Traitement des entrées/sorties

9.1 Introduction

Dans ce chapitre, une introduction aux dispositifs d'entrée-sortie (pilotes de périphérique : *drivers*) est présentée. Ce type de dispositif est nécessaire pour accéder au matériel et aux périphériques : système de fichiers, disques SATA, SCSI ou IDE, mémoire de cartes ISA ou PCI, composantes de réseau, carte d'entrée-sortie (son, conversion analogique/numérique et numérique/analogique, etc).

Sous Linux, les dispositifs d'entrée-sortie sont mis en oeuvre dans le *kernel space*, le noyau du système d'exploitation. Bien que la programmation de ces dispositifs ne soit pas nécessairement plus difficile qu'un programme d'application, les conséquences d'une mauvaise programmation sont plus néfastes que pour un simple programme. Ainsi, un pilote de périphérique (*device driver*) peut monopoliser tout le temps CPU ou rendre le système instable.

Comme le code source des pilotes de périphériques utilisés sous Linux est disponible, une des meilleures façons d'apprendre à les programmer est d'analyser ces programmes. Dans les prochaines sections, nous allons présenter des éléments permettant de mieux comprendre le fonctionnement d'un pilote de périphérique.

Pourquoi écrire des pilotes de périphériques ? La grande partie des périphériques que l'on retrouve dans un PC Intel x86 sont supportés par Linux. Cependant, des périphériques spécialisés (carte d'acquisition de données, ...), très récents (carte vidéo dernier cri, ...) ou construits sur place (une carte PCI produite dans votre *compagnie* avec un DSP, ...) n'auront pas de pilotes disponibles. Il faudra donc en écrire afin de les utiliser. Dans d'autres cas, le pilote disponible depuis peu de temps contiendra encore un bogue que l'on voudra corriger.

Les pilotes sont des programmes qui offrent une interface aux applications permettant l'interaction avec les périphériques. Certains pilotes ne contrôlent pas un périphérique *physique* mais un périphérique *logiciel*. Ce périphérique logiciel en oeuvre certaines fonctionnalités qui ne dépendent pas d'une composante matérielle. Citons par exemple : un *RAM disk* ou `/dev/random` qui génère des données de façon aléatoire (pour plus d'informations utiliser la commande `man 4 random`).

Les pilotes présentent une couche d'interface entre le *kernel* et le matériel. Ainsi, par une interface standardisée, différents périphériques pourront s'intégrer au *kernel* même ci celui-ci a été compilé avant que le pilote soit disponible. Par exemple, le système de fichiers est décomposé en un système de fichiers virtuel générique (*virtual file system* : *VFS*) qui communique avec les systèmes de fichiers spécifiques (CD-ROM, SCSI, IDE, FAT, EXT2, etc).

9.2 Classes de pilotes

Les pilotes de périphériques peuvent être classés selon leur comportement :

périphériques de type caractères où les lectures (ou écritures) se font de façon séquentielle octet par octet (dérouleur de bande, carte de son, etc) ;

périphériques de type blocs où les lectures (ou écritures) sont non séquentielles donnant accès à un nombre de blocs par l'intermédiaire d'un tampon de cache (VFS, disques rigides SCSI ou IDE, système de fichiers, etc) ;

interfaces réseau où l'information est reçue (ou envoyée) par des *packets*.

À l'exception des interfaces réseau, les pilotes de périphériques peuvent être accédés par le système de fichiers. Le répertoire `/dev` contient des fichiers spéciaux de pilotes. Par exemple, la commande :

```
ls -l /dev
```

permet d'obtenir la liste des *devices* :

```
...
crw-rw----   1 root lp     6,   0 Oct 25 06:41 lp0
crw-rw----   1 root lp     6,   1 Oct 25 06:41 lp1
crw-rw----   1 root lp     6,   2 Oct 25 06:41 lp2
...
brw-rw-rw-   1 root root   8,   0 Oct 25 06:41 sda
brw-rw-rw-   1 root root   8,   1 Oct 25 06:41 sda1
```

```
brw-rw-rw-   1 root root   8,   2 Oct 25 06:41 sda2
...
crw-rw-rw-   1 root root   4,   0 Oct 25 06:41 tty0
crw-rw-rw-   1 root root   4,   1 Oct 25 06:42 tty1
...
```

Dans cette liste, les indicateurs `c` et `b` sont utilisés pour différentier les périphériques de type caractères et les périphériques de type blocs. Les nombres suivants les *owner* et *group* sont les numéros majeurs et mineurs des pilotes. Ainsi, `/dev/hda1` est un périphérique de type blocs avec 3 et 1 comme numéro majeur et numéro mineur. Ici, `sda` est le premier disque SATA et `sda1` est la première partition sur ce disque. Le numéro majeur donne le disque et le mineur la partition. Un seul pilote s'occupe de l'interface avec le premier disque SATA (majeur = 3), et c'est le numéro mineur qui permet de distinguer les partitions. Le premier port parallèle (majeur = 6, mineur = 0) est de type caractères `lp0`. On ne choisit pas arbitrairement un numéro de pilote, ceux-ci étant réservés. Les valeurs standards sont données dans le fichier `/usr/src/linux/Documentation/devices.txt`.

Pour créer ces fichiers spéciaux, il faut utiliser la commande :

```
mknod name type major minor
```

Exemple : `mknod /dev/lp0 c 6 0`

9.3 Modes d'exécution

Sous Linux, les instructions peuvent être exécutées selon deux modes :
— le mode usager (*user-mode*) ;
— et le mode de supervision (*kernel-mode*).

Les processeurs x86 supportent 4 modes d'exécution (*Ring 0, 1, 2, et 3*) où *Ring 0* est le mode de plus haut privilège. Sous Linux le *kernel-mode* correspond au *Ring 0* et le *user-mode* au *Ring 3*. Bon nombre de fonctionnalités disponibles dans le *user-mode* (par exemple, les fonctions disponibles dans libc : `printf`, `strcpy`, ...) ne le sont nécessairement pas en *kernel-mode*. Il faut donc suivre les règles suivantes lors de la création de code exécutant en *kernel-mode* :

— il ne faut pas utiliser le coprocesseur mathématique à moins de sauvegarder son état ;
— n'utilisez pas de boucles d'attente sur une condition car votre code pourra monopoliser tout le CPU ;

— garder votre code simple et compréhensible car le débogage est beaucoup plus difficile dans le *kernel-mode*.

9.4 Compilation d'un module

Linux permet de charger des pilotes (modules) sans que ceux-ci aient été compilés à même le noyau. Ceci permet de générer un noyau compact qui pourra charger dynamiquement les pilotes lorsque nécessaire. On peut se référer à un exemple de module minimal qui contient les éléments essentiels d'un module :

— l'inclusion des fichiers d'entête `linux/module.h`, `linux/modversions.h` et `linux/kernel.h` ;
— la fonction `static int __init hello_init(void)` qui est en charge de l'initialisation du module lors du chargement ;
— et la fonction `static void __exit hello_cleanup(void)` qui est exécutée lorsque l'on décharge le module.

Dans cet exemple, les fonctions ne font qu'écrire un message dans un *log* que l'on peut consulter à l'aide de la commande `dmesg`. La fonction du kernel `printk` est l'équivalent du `printf` de libc à la différence que `printk` ne supporte pas les données en point flottant et que la chaîne de format commence par le niveau de priorité du message défini dans `linux/kernel.h` par :

```
#define KERN_EMERG      "<0>"   /* system is unusable                   */
#define KERN_ALERT      "<1>"   /* action must be taken immediately     */
#define KERN_CRIT       "<2>"   /* critical conditions                  */
#define KERN_ERR        "<3>"   /* error conditions                     */
#define KERN_WARNING    "<4>"   /* warning conditions                   */
#define KERN_NOTICE     "<5>"   /* normal but significant condition     */
#define KERN_INFO       "<6>"   /* informational                        */
#define KERN_DEBUG      "<7>"   /* debug-level messages                 */
```

Après le compilation du module `hello.c`, on peut charger le module avec la commande `insmod` (*install module*) et le décharger avec `rmmod` (*remove module*). De plus, la commande `lsmod` permet d'obtenir la liste des modules qui sont chargés(vous devez être l'usager `root` pour utiliser `insmod` et `rmmod`).

9.5 Types de données et variables globales

Comme `Linux` peut fonctionner sous différentes architectures (32 bits et 64 bits) et que la portabilité entre celles-ci est importante, dans plusieurs cas, il est

préférable d'utiliser des types standards définis sous Linux pour les entiers (car un `int` n'a pas nécessairement la même dimension sous les architectures 32 bits et 64 bits) :

`__u8`, ..., `__u64`	entiers non signés de 8 bits, ..., 64 bits
`__s8`, ..., `__s64`	entiers signés de 8 bits, ..., 64 bits

On peut aussi utiliser d'autres types (comme `size_t`, etc) qui sont définis dans `linux/types.h` et `asm/posix_types.h`. En utilisant ces types de données, il sera plus facile de développer un module portable entre différentes architectures.

Comme tous les modules partagent le même *namespace*, les variables et fonctions globales d'un module doivent être `static` pour éviter les conflits de noms (à l'exception de `module_init` et `module_exit`).

9.6 Aspects généraux d'un pilote

Enregistrement d'un pilote

Les pilotes (par caractères et par blocs) se comportent comme des fichiers. Pour réaliser un pilote, il faudra enregistrer celui-ci en fournissant une mise en oeuvre des différentes opérations valides pour un fichier, l'usage de la mémoire du périphérique et le support d'un système de fichiers. Cette information est fournie à l'aide de la structure suivante pour les pilotes par caractères :

```
struct file_operations {
   struct module *owner;
   loff_t (*llseek) (struct file *, loff_t, int);
   ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
   ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
   ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
   ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
   int (*iterate) (struct file *, struct dir_context *);
   unsigned int (*poll) (struct file *, struct poll_table_struct *);
   long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
   long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
   int (*mmap) (struct file *, struct vm_area_struct *);
   int (*mremap)(struct file *, struct vm_area_struct *);
   int (*open) (struct inode *, struct file *);
   int (*flush) (struct file *, fl_owner_t id);
   int (*release) (struct inode *, struct file *);
   int (*fsync) (struct file *, loff_t, loff_t, int datasync);
```

```
   int (*aio_fsync) (struct kiocb *, int datasync);
   int (*fasync) (int, struct file *, int);
   int (*lock) (struct file *, int, struct file_lock *);
   ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
                        loff_t *, int);
   unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                       unsigned long, unsigned long, unsigned long);
   int (*check_flags)(int);
   int (*flock) (struct file *, int, struct file_lock *);
   ssize_t (*splice_write)(struct pipe_inode_info *, struct file *,
                           loff_t *, size_t, unsigned int);
   ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
                          size_t, unsigned int);
   int (*setlease)(struct file *, long, struct file_lock **, void **);
   long (*fallocate)(struct file *file, int mode, loff_t offset,
                     loff_t len);
   void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
   unsigned (*mmap_capabilities)(struct file *);
#endif
};
```

Pour les pilotes par blocs, on utilise en plus la structure suivante :

```
struct block_device_operations {
    int (*open) (struct block_device *, fmode_t);
    void (*release) (struct gendisk *, fmode_t);
    int (*rw_page)(struct block_device *, sector_t, struct page *, int rw);
    int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
    int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
    long (*direct_access)(struct block_device *, sector_t,
                          void **, unsigned long *pfn, long size);
    unsigned int (*check_events) (struct gendisk *disk,
                                  unsigned int clearing);
    /* ->media_changed() is DEPRECATED, use ->check_events() instead */
    int (*media_changed) (struct gendisk *);
    void (*unlock_native_capacity) (struct gendisk *);
    int (*revalidate_disk) (struct gendisk *);
    int (*getgeo)(struct block_device *, struct hd_geometry *);
    /* this callback is with swap_lock and sometimes page table lock held */
    void (*swap_slot_free_notify) (struct block_device *, unsigned long);
    struct module *owner;
};
```

Si votre pilote ne met pas en oeuvre une des fonctions, il suffit de fournir un pointeur `NULL` pour la fonction correspondante. Voici une brève description

de certaines des différentes fonctions (la documentation de libc et les man pages contiennent beaucoup d'informations au sujet de l'utilisation de ces fonctions) :

- `llseek` : change de position dans la structure de fichier (pour plus d'informations voir `man llseek`) ;
- `read` : lecture (du point de vue de l'application) dans le fichier (pour plus d'informations voir `man read`) ;
- `write` : écriture (du point de vue de l'application) dans le fichier(pour plus d'informations voir `man write`) ;
- `readdir` : lecture du contenu d'un répertoire (n'a de sens que pour les systèmes de fichiers) (pour plus d'informations voir `man readdir`) ;
- `poll` permet à l'application d'être avisée de l'occurrence d'un événement par le pilote (pour plus d'informations voir `man poll`) ;
- `ioctl` : contrôle d'I/O (pour plus d'informations voir `man ioctl`) ;
- `mmap` : *memory mapping* d'un espace fichier permettant l'accès au fichier comme pour un accès mémoire (pour plus d'informations voir `man mmap`) ;
- `open` : ouverture du pilote (ou fichier) (pour plus d'informations voir `man open`) ;
- `flush` : *flush* du tampon d'un pilote par blocs (pour plus d'informations voir `man flush`) ;
- `release` : fermeture du fichier (ou du pilote), appelé lors d'un `close` ;
- `fsync` : synchronisation mémoire (cache de disque) du pilote par bloc (pour plus d'informations voir `man fsync`) ;
- `fasync` : appelé lorsque l'application utilise `fcntl` ;
- `lock` : verrou de fichier (pour système de fichiers).

La fonction de type `static int __init` doit appeler une des fonctions suivantes :

```
int register_blkdev(unsigned int major, const char * name,
                    struct block_device_operations *bdops)
int register_chrdev(unsigned int major, const char *name,
                    struct file_operations *fops);
```

où

- `major` : est le numéro majeur du pilote ;
- `name` : est le nom du pilote ;
- `fops` ou `bdops` : est la structure contenant les pointeurs de fonctions.

Bien sûr, la fonction de type `static void __exit` doit appeler une des fonctions :

```
int unregister_blkdev(unsigned int major, const char * name);
int unregister_chrdev(unsigned int major, const char * name);
```

Usage count

Comme un pilote peut être déchargé, il faut s'assurer qu'aucune application utilise le pilote lors du déchargement. Le *usage count* est une varible qui assure le suivi du nombre de modules ou d'applications utilisant le module. Ainsi, lors de l'exécution de la commande `rmmod`, la valeur de *usage count* est vérifiée et le module sera déchargé si et seulement si cette valeur est zéro.

9.7 Pour aller plus loin

Bien qu'elle traite du noyau version 2.6.10, la référence Linux Device Drivers, Third Edition constitue un excellent document pour l'apprentissage de la programmation de pilotes.

Bibliographie

[1] ARM. *ARM® Architecture Reference Manual: ARMv7-A and ARMv7-R edition*, 2014.

[2] ARM. *ARM® Cortex®-A Series Programmer's Guide for ARMv8-A*, 2015.

[3] John L. Hennessy, David A. Patterson, and Krste Asanović. *Computer architecture : a quantitative approach*. Elsevier Morgan Kaufmann, Amsterdam, 5th edition, 2012.

[4] Intel. *Combined Volume Set of Intel® 64 and IA-32 Achitectures Software Developer's Manuals*, June 2015.

[5] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 9th edition, 2013.

Annexe A

Représentation des nombres

A.1 Introduction

Un rappel sur la représentation des nombres en format binaire (représentation interne des données dans l'ordinateur), format octal et hexadécimal est présenté, de même que la représentation à virgule flottante (format IEEE 754). Les opérations élémentaires (addition, soustraction, etc) peuvent résulter en des débordements qui s'interprètent différemment selon que l'on considère le nombre comme signé ou non signé.

A.2 Représentation binaire

A.2.1 Passage binaire à décimal

La représentation binaire des nombres utilise les chiffres 0 et 1. Chaque chiffre représente le multiplicateur d'une puissance de 2. Par exemple, le nombre 1011 0111 :

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
1	0	1	1	0	1	1	1	
128	+0	+32	+16	+0	+4	+2	+1	= 183

est l'équivalent de 183 en décimal.

A.2.2 Passage de décimal à binaire

On peut passer de la représentation décimale à la représentation binaire par une succession de divisions par 2 où les restes donnent le nombre binaire dans l'ordre inverse. Par exemple, le nombre 83 :

	83	$\div 2 = 41$	$\div 2 = 20$	$\div 2 = 10$	$\div 2 = 5$	$\div 2 = 2$	$\div 2 = 1$	$\div 2 = 0$
	-82	-40	-20	-10	-4	-2	-0	
reste	1	1	0	0	1	0	1	

donne 0101 0011 en binaire. On peut vérifier la validité de ce résultat :

64	32	16	8	4	2	1	
1	0	1	0	0	1	1	
64	+0	+16	+0	+0	+2	+1	= 83

Le bit le moins significatif est le bit le plus à droite du nombre binaire. Le bit le plus significatif est le bit le plus à gauche du nombre binaire. En assembleur `as` ou en langage `C`, le nombre binaire 0101 0011 s'écrit en plaçant `0` suivi de la lettre `b` avant le nombre en binaire. Exemple : `0b01010011`.

A.3 Représentation octale

A.3.1 Passage d'octal à décimal

La représentation octale des nombres utilise les chiffres 0 à 7 et chaque chiffre représente le multiplicateur d'une puissance de 8. Par exemple, le nombre 1751 :

8^3	8^2	8^1	8^0	
1	7	5	1	
512	+448	+40	+1	= 1001

est l'équivalent de 1001 en décimal.

En assembleur `as` ou en langage `C`, le nombre octal 1751 s'écrit en plaçant `0` au début du nombre octal. Exemple : `01751`.

A.3.2 Passage de décimal à octal

On peut passer de la représentation décimale à la représentation octale par une succession de divisions par 8 où les restes donnent le nombre octal dans l'ordre inverse. Par exemple, le nombre 83 :

	83	$\div 8 = 10$	$\div 8 = 1$	$\div 8 = 0$
	-80		-8	-0
reste	3		2	1

donne 123 en octal. On peut vérifier la validité de ce résultat :

64	8	1	
1	2	3	
64	+16	+3	= 83

A.3.3 Passage de binaire à octal

On groupe le nombre binaire par paquet de 3 bits en commençant par le bit le moins significatif. Ensuite, on trouve le chiffre octal de chaque paquet. Par exemple, le nombre binaire 01100110 = 01 100 110 = 146 en octal.

A.3.4 Passage d'octal à binaire

Chaque chiffre du nombre octal est décomposé en chiffre binaire de 3 bits. Par exemple, le nombre octal 762 = 111 110 010. Ce qui donne le nombre binaire suivant : 111110010.

A.4 Représentation hexadécimale

A.4.1 Passage d'hexadécimal à décimal

La représentation hexadécimale des nombres utilise les chiffres 0 à 9 et les lettres A à F (pour les valeurs comprises entre 10 et 15). Chaque chiffre (ou lettre) représente le multiplicateur d'une puissance de 16. Par exemple, le nombre 1A5C :

16^3	16^2	16^1	16^0	
1	10	5	12	
4096	+2560	+80	+12	= 6748

est l'équivalent de 6748 en décimal.

En assembleur `as` ou en langage `C`, le nombre hexadécimal 1A5C s'écrit en plaçant `0` suivi de la lettre `x` avant le nombre hexadécimal. Exemple : `0x1A5C`.

A.4.2 Passage décimal à hexadécimal

On peut passer de la représentation décimale à la représentation hexadécimale par une succession de divisions par 16 où les restes donnent le nombre hexadécimal dans l'ordre inverse. Par exemple, le nombre 183 :

	183	$\div 16 = 11$	$\div 16 = 0$
	-176	-0	
reste	7	B	

donne $B7$ en hexadécimal. On peut vérifier la validité de ce résultat :

16	1	
B	7	
176	+7	= 183

A.4.3 Passage de binaire à hexadécimal

On peut passer de la représentation binaire à la représentation hexadécimale par la conversion en hexadécimal de regroupements de quatre chiffres à commençant par la droite. En binaire, le nombre 1111 donne 15 en décimal ou F en hexadécimal (le plus grand chiffre hexadécimal). Par exemple, le nombre 1100 1111 donne CF en hexadécimal ($\underbrace{1100}_{12=C}\,\underbrace{1111}_{15=F}$).

La représentation hexadécimale est donc bien adaptée comparativement au binaire pour représenter les nombres dans les programmes car elle est compacte (deux chiffres sont nécessaires pour représenter un octet) et s'aligne sur les frontières entre les octets.

A.4.4 Passage de hexadécimal à binaire

Chaque chiffre du nombre hexadécimal est décomposé en un unique paquet binaire de 4 bits. Par exemple, le nombre hexadécimal A5F7 donne le nombre binaire 1010 0101 1111 0111, car on sait que A =1010, 5 = 0101, F =1111, et 7=0111.

A.4.5 Passage de octal à hexadécimal et de hexadécimal à octal

Si on veut passer de la représentation octale à hexadécimale ou vice versa, alors on décompose la représentation octale (ou hexadécimale) en binaire, et ensuite la représentation binaire en hexadécimale (ou octale).

A.5 Nombres entiers signés et non signés

La représentation des nombres signés (entiers positifs et négatifs) et non signés (entiers positifs) doit permettre, par exemple, d'additionner deux nombres avec le même « algorithme », qu'ils soient signés ou non. Après l'opération, les différents types de débordements (signés ou non signés) seront signalés dans des indicateurs différents (OF ou CF).

A.5.1 Représentation des entiers signés en complément à 2

Considérons le nombre 7 représenté par un octet en binaire : 0000 0111. Comment représenter son négatif (-7) ? Comme $7 + (-7) = 0$, nous avons

$$\begin{array}{r} 0000\,0111 \\ +\ ????\,???? \\ \hline 0000\,0000 \end{array} \Rightarrow \begin{array}{r} 0000\,0111 \\ +\ 1111\,1001 \\ \hline 1\ \underbrace{0000\,0000}_{0} \end{array}$$

Comme il n'y pas de neuvième bit dans un octet, 1111 0001 peut donc être considéré comme le négatif de 0000 0111. Le bit le plus significatif de l'octet est lié au signe : 0(+) et 1(−). Pour obtenir le négatif en complément de 2, il suffit de faire les opérations suivantes :

1. faire le complément du nombre (inversion de tous les bits) ;
2. ajouter 1 au résultat.

Voici des exemples :

valeur initiale	0000 0111	$\Rightarrow 7$	valeur initiale	0000 0000	$\Rightarrow 0$
complément	1111 1000		complément	1111 1111	
+1	1111 1001	$\Rightarrow -7$	+1	0000 0000	$\Rightarrow 0$

Voici une « astuce » pour obtenir le négatif : partant du bit le moins significatif, on garde les bits comme ils sont jusqu'à ce que l'on rencontre un 1 (inclusivement), puis tous les autres bits sont inversés.

Un octet peut donc représenter des entiers signés et non signés dans les intervalles suivants :

octet non signé : $0 \rightarrow 255$	
représentation	valeur
1111 1111	255
1111 1110	254
⋮	
0000 0001	1
0000 0000	0

octet signé : $-128 \rightarrow 127$	
représentation	valeur
0111 1111	127
0111 1110	126
⋮	
0000 0001	1
0000 0000	0
1111 1111	-1
⋮	
1000 0000	-128

Ainsi, l'opération d'addition sur les nombres signés et non signés se fait de la même façon. C'est au programmeur (ou au compilateur) de faire la bonne « interprétation » du résultat. Après une opération d'addition, le processeur placera les informations pertinentes dans les indicateurs de retenue « CF - Carry Flag » ou de débordement « OF - Overflow Flag » selon l'interprétation du résultat en non signé ou signé respectivement.

A.6 Addition d'entiers

À titre d'exemple, considérons l'addition de deux nombres binaires pouvant être considérés comme des entiers non signés ($0 \rightarrow 255$) ou signés ($-128 \rightarrow 127$) utilisant chacun un octet. Voici un premier exemple :

		(non signés)	(signés)
	0000 1111	(15)	(15)
+	0000 0001	(1)	(1)
=	0001 0000	(16) et $CF = 0$	(16) et $OF = 0$

Le résultat peut être représenté par un octet, il n'y a pas de débordement, $CF = 0$ et $OF = 0$.

Voici un deuxième exemple :

		(non signés)	(signés)
	1111 1111	(255)	(-1)
+	0000 0001	(1)	(1)
=	$\underbrace{1}_{CF}$ 0000 0000	(0) et $CF = 1$	(0) et $OF = 0$

Le résultat non signé ne peut pas être représenté par un octet (0-255), il y a un débordement non signé (une retenue) et $CF = 1$. Le résultat signé peut être représenté par un octet, il n'y a pas de débordement signé et $OF = 0$.

Voici un autre exemple :

		(non signés)	(signés)
	1000 0000	(128)	(-128)
+	1000 0000	(128)	(-128)
=	$\underbrace{1}_{CF}$ 0000 0000	(0) et $CF = 1$	(0) et $OF = 1$

Le résultat de l'addition de deux nombres signés négatifs résulte en un nombre positif ! Il y a donc un débordement signé. De plus, on a un débordement non signé car le résultat de l'addition des deux nombres non signés dépasse la capacité d'un octet.

Voici un quatrième exemple :

		(non signés)	(signés)
	0111 1111	(127)	(127)
+	0000 0011	(3)	(3)
=	1000 0010	(130) et $CF = 0$	(-126) et $OF = 1$

Le résultat de l'addition de deux nombres signés positifs résulte en un nombre négatif ! Il y a un débordement signé car $130 > 127$ et $OF = 1$.

Voici un dernier exemple :

		(non signés)	(signés)
	1011 1111	(191)	(-65)
+	1010 1011	(171)	(-85)
=	$\underbrace{1}_{CF}$ 0110 1010	(106) et $CF = 1$	(106) et $OF = 1$

Le résultat de l'addition de deux nombres négatifs résulte en un nombre positif ! Il y a un débordement signé car $-150 < -128$ et $OF = 1$.

Pour les nombres signés, il n'y a jamais de débordement lors de l'addition de

deux nombres de signes opposés.

A.7 Soustractions d'entiers

Pour la soustraction d'entiers on utilise le fait que $A - B = A + (-B)$ et que $-B = \overline{B} + 1$ où $\overline{B}$ est le complément binaire de B. Donc, $A - B = A + \overline{B} + 1$ qui ce traite comme deux additions successives.

Voici un exemple de soustraction sans débordement, $16 - 15$:

	0001 0000	(16)
-	0000 1111	(15)
=		
	0001 0000	(16)
+	1111 0000	($\overline{15} = -16$)
	0000 0000	(0)
+	0000 0001	(1)
=	0000 0001	(1)

Voici un exemple de soustraction avec débordement non signé, $15 - 16$:

	0000 1111	(15)
-	0001 0000	(16)
=		
	0000 1111	(15)
+	1110 1111	($\overline{16} = -17$)
	1111 1110	(-2)
+	0000 0001	(1)
=	1111 1111	(-1 en signé ou 255 en non signé)

Donc, $CF = 1$ et $OF = 0$

Voici un exemple de soustraction avec débordement signé, $-128-2$ ou $128-2$:

	1000 0000	(-128 ou 128)
-	0000 0010	(2)
=		
	1000 0000	(-128 ou 128)
+	1111 1101	($\overline{2} = -3$)
	0111 1101	(125)
+	0000 0001	(1)
=	0111 1110	(126 au lieu -130 en signé, 126 en non signé)

Donc, $CF = 0$ et $OF = 1$

A.8 Nombres à virgule flottante : format IEEE 754

Soit un nombre `X = n.f` où `n` représente la partie entière et `f` la partie fractionnaire, la représentation binaire de `n` est calculée selon la méthode du paragraphe A.2.2, et la représentation binaire de `f` est trouvée selon l'algorithme suivant :

```
Soit f la partie fractionnaire d'un nombre réel
Tant que (f != 0)
        si f < 0.5
                bit = 0
        sinon
                bit = 1
        f = partie fractionnaire de (f*2)
```

Selon cet algorithme, on obtient les bits du plus significatif au moins significatif de la partie fractionnaire.

Exemple : soit le nombre réel 278.153.

278 = 1 0001 0110
0.153 = 0.0010 0111 0010 1011 d'après le tableau A.1.
donc 278.153 = 1 0001 0110.0010 0111 0010 1011

Norme IEEE 754

Pour obtenir un nombre à virgule flottante en format IEEE 754, il faut :

1. Trouver la représentation binaire de la partie entière et la partie fractionnaire du nombre.

TABLE A.1 – Représentation binaire de la partie fractionnaire 0.153

Partie fractionnaire	Bit Courant
0.153	0
0.306	0
0.612	1
0.224	0
0.448	0
0.896	1
0.792	1
0.584	1
0.168	0
0.336	0
0.672	1
0.344	0
0.688	1
0.376	0
0.752	1
0.504	1

2. Normaliser la représentation binaire selon que $\pm abs(M) * 2^E$ où M est la mantisse, $1 <= M < 2$ et E l'exposant. Arrondir la mantisse à 23 bits ou 52 bits.
3. Trouver la caractéristique de l'exposant. La caractéristique pour simple précision est $C = E + 127$ et pour double précision $C = E + 1023$.
4. Retirer le premier bit de la mantisse car il est sous-entendu.
5. Placer les bits selon le format du tableau A.2.

Si on applique les étapes de norme `IEEE 754` simple précision à l'exemple précédent, on a

1. 278.153 = 1 0001 0110.0010 0111 0010 1011
2. Normaliser.
   1 0001 0110.0010 0111 0010 1011 = 1.0001 0110 0010 0111 0010 1011 *2^8.
   Arrondir.
   1.0001 0110 0010 0111 0010 110 *2^8.
3. C = 8+127 = 135 = 1000 0111
4. Mantisse = 0001 0110 0010 0111 0010 110
5.

Signe	Caractéristique	Mantisse
0	1000 0111	0001 0110 0010 0111 0010 110

TABLE A.2 – Formats standards IEEE 754

	NB Bit Signe 0=Positif 1=Négatif	NB Bits Exposant	NB Bits Mantisse	Total
Simple Précision	1	8	23	32
Double Précision	1	11	52	64

Annexe B

Codes ASCII

Table B.1 – Table ASCII (caractères 0-127)

dec		0	16	32	48	64	80	96	112
	hex	0	1	2	3	4	5	6	7
0	0	NUL	DLE	SPACE	0	@	P	‘	p
1	1	SOH	DC1	!	1	A	Q	a	q
2	2	STX	DC2	”	2	B	R	b	r
3	3	ETX	DC3	#	3	C	S	c	s
4	4	EOT	DC4	$	4	D	T	d	t
5	5	ENQ	NAK	%	5	E	U	e	u
6	6	ACK	SYN	&	6	F	V	f	v
7	7	BEL	ETB	’	7	G	W	g	w
8	8	BS	CAN	(	8	H	X	h	x
9	9	HT	EM	)	9	I	Y	i	y
10	a	LF	SUB	*	:	J	Z	j	z
11	b	VT	ESC	+	;	K	[	k	{
12	c	FF	FS	,	<	L	\	l	|
13	d	CR	GS	-	=	M	]	m	}
14	e	SO	RS	.	>	N	^	n	~
15	f	SI	US	/	?	O	_	o	DEL

TABLE B.2 – Caractères de contrôle ASCII

dec	hex	Ctrl-	mnémonic	description
0	00		NUL	Null character
1	01	Ctrl-A	SOH	Start of header
2	02	Ctrl-B	STX	Start of text
3	03	Ctrl-C	ETX	End of text
4	04	Ctrl-D	EOT	End of transmission
5	05	Ctrl-E	ENQ	Enquiry
6	06	Ctrl-F	ACK	Acknowledge
7	07	Ctrl-G	BEL	Bell
8	08	Ctrl-H	BS	Backspace
9	09	Ctrl-I	HT	Horizontal tab
10	0a	Ctrl-J	LF	Line feed
11	0b	Ctrl-K	VT	Vertical tab
12	0c	Ctrl-L	FF	Form feed
13	0d	Ctrl-M	CR	Cariage return
14	0e	Ctrl-N	SO	Shift out
15	0f	Ctrl-O	SI	Shift in
16	10	Ctrl-P	DLE	Data link escape
17	11	Ctrl-Q	DC1	Device control 1
18	12	Ctrl-R	DC2	Device control 2
19	13	Ctrl-S	DC3	Device control 3
20	14	Ctrl-T	DC4	Device control 4
21	15	Ctrl-U	NAK	Negative acknowledgement
22	16	Ctrl-V	SYN	Synchronous idle
23	17	Ctrl-W	ETB	End transmission block
24	18	Ctrl-X	CAN	Cancel
25	19	Ctrl-Y	EM	End of medium
26	1a	Ctrl-Z	SUB	Substitute
27	1b	Ctrl-[	ESC	Escape
28	1c	Ctrl-\	FS	File separator
29	1d	Ctrl-]	GS	Group separator
30	1e	Ctrl-^	RS	Record separator
31	1f	Ctrl--	US	Unit separator

Annexe C

Génération d'un compilateur croisé

La génération d'un compilateur croisé implique plusieurs étapes permettant d'obtenir les outils de compilation, les fichiers d'entête, les librairies et un répertoire racine du système cible (*sysroot*).

Pour générer un compilateur croisé GGC il faut avoir une installation de type Unix/Linux avec un compilateur GCC relativement récent, GNU make, GNU bison, flex. GCC aura aussi besoin des librairies GNU GMP, GNU MPFR, et GNU MPC pour le support de point flottant. Les librairies optionnelles ISL et CLooG permettront des optimisations supplémentaires.

Comme nous l'avons vu au chapitre 5, le compilateur gcc (ou g++) génère du code assembleur pour le processeur cible, l'assembleur génère un fichier objet qui est inclus dans l'exécutable final par l'éditeur de liens.

Il faut donc avoir les outils de Binutils configurés pour le processeur cible afin que GCC puisse les utiliser pour l'architecture cible. Il faudra donc choisir une combinaison de versions de GCC et Binutils compatibles.

Après avoir configuré et compilé Binutils, il faudra configurer GCC. Pour y arriver, les fichiers d'entête du noyau (*kernel*) sont requis dans le cadre d'un système Linux pour générer la librairie standard Glibc qui prend en charge les appels système. Comme GCC depend de Glibc et vice versa, GCC ne peut pas être compilé en une seule étape, mais plutôt dans un aller-retour GCC–Glibc qui satisfait les dépendances.

Les prochaines sections sont traduites et adaptées de ce tutoriel pour générer une version 5.2.0 pour Arm 64 bits.

C.1 Téléchargement

Il faut premièrement télécharger les composantes requises. Les numéros de versions s'assurent de compatibilité entre les divers éléments. Dans l'exemple ci-dessous, la compilation ne requiert pas de *patches*, ce qui n'est pas toujours le cas. Dans un projet Yocto-Linux, ce travail de choix de versions et d'application de *patches* est fait pour vous.

```
$ wget http://ftpmirror.gnu.org/binutils/binutils-2.25.1.tar.gz
$ wget http://ftpmirror.gnu.org/gcc/gcc-5.2.0/gcc-5.2.0.tar.gz
$ wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.17.2.tar.xz
$ wget http://ftpmirror.gnu.org/glibc/glibc-2.23.tar.xz
$ wget http://ftpmirror.gnu.org/mpfr/mpfr-3.1.4.tar.xz
$ wget http://ftpmirror.gnu.org/gmp/gmp-6.1.0.tar.xz
$ wget http://ftpmirror.gnu.org/mpc/mpc-1.0.2.tar.gz
$ wget ftp://gcc.gnu.org/pub/gcc/infrastructure/isl-0.14 .tar.bz2
$ wget ftp://gcc.gnu.org/pub/gcc/infrastructure/cloog-0.18.1.tar.gz
$ for f in *.tar*; do tar xf $f; done
$ cd gcc-5.2.0
$ ln -s ../mpfr-3.1.4 mpfr
$ ln -s ../gmp-6.1.0 gmp
$ ln -s ../mpc-1.0.2 mpc
$ ln -s ../isl-0.14 isl
$ ln -s ../cloog-0.18.1 cloog
$ cd ..
```

Le liens symboliques permettront la compilation de GCC avec les librairies choisies.

C.2 Binutils

Nous allons maintenant compiler et installer Binutils pour notre architecture cible.

```
$ mkdir build-binutils
$ cd build-binutils
$ export INSTALL_PATH=/export/tmp/4205_nn/opt/crossAA64
$ export TARGET=aarch64-linux
$ export CONFIGURATION_OPTIONS="--disable-multilib"
$ export PATH=$INSTALL_PATH/bin:$PATH
```

Les dernières commandes spécifient le chemin pour l'installation du compilateur croisé, l'architecture cible et l'option pour binaires 64bits seulement. On peut maintenant configurer, compiler et installer

```
$ ../binutils-2.25.1/configure --prefix=$INSTALL_PATH \
  --target=$TARGET $CONFIGURATION_OPTIONS
$ make -j4
$ make install
$ cd ..
```

C.3 Fichier d'entête du noyau

Maintenant, installons les fichiers d'entête du noyau. Pour Linux, `aarch64` est remplacé par `arm64`.

```
$ cd linux-3.17.2
$ make ARCH=arm64 INSTALL_HDR_PATH=$INSTALL_PATH/$TARGET headers_install
$ cd ..
```

C.4 GCC initial

Une première compilation de GCC pour les langages C, C++ seulement (pas de Fortran, Java etc.).

```
$ mkdir build-gcc
$ cd build-gcc
$ ../gcc-5.2.0/configure --prefix=$INSTALL_PATH --target=$TARGET \
   --enable-languages=c,c++ $CONFIGURATION_OPTIONS
$ make -j4 all-gcc
$ make install-gcc
$ cd ..
```

C.5 Glibc fichiers d'entête et de *startup*

Il nous faut mantenant installer les fichiers d'entête de Glibc, générer les fichiers de *startup* et les copier dans l'installation. Deux fichiers « vides » sont créés et seront remplacés à l'étape finale (`libc.so` et `stubs.h`).

```
$ mkdir build-glibc
$ cd build-glibc
$ ../glibc-2.23/configure --prefix=$INSTALL_PATH/$TARGET --build=$MACHTYPE \
  --host=$TARGET --target=$TARGET --with-headers=$INSTALL_PATH/$TARGET/include \
 $CONFIGURATION_OPTIONS libc_cv_forced_unwind=yes
$ make install-bootstrap-headers=yes install-headers
$ make -j4  csu/subdir_lib
$ install csu/crt1.o csu/crti.o csu/crtn.o $INSTALL_PATH/$TARGET/lib
$ $TARGET-gcc -nostdlib -nostartfiles -shared -x c /dev/null \
  -o $INSTALL_PATH/$TARGET/lib/libc.so
$ touch $INSTALL_PATH/$TARGET/include/gnu/stubs.h
$ cd ..
```

C.6 Librairies de support du compilateur

Les compilateurs créés précédemment sont utilisés pour générer les librairies de support pour GCC : `libgcc`, etc.

```
$ cd build-gcc
$ make -j4 all-target-libgcc
$ make install-target-libgcc
$ cd ..
```

C.7 Compilation de Glibc

On peut maintenant compiler et installer Glibc.

```
$ cd build-glibc
$ make -j4
$ make install
$ cd ..
```

C.8 Librairie C++ et installation finale

Tous les éléments restant peuvent maintenant être compilés et installés : `libstdc++`, etc.

```
$ cd build-gcc
$ make -j4
$ make install
$ cd ..
```