



# Accélération du démarrage d'un système GNU/Linux.

AFF065-2009-NOT092

\$Rev: 407 \$

2011-04-03 13:22:44+02:00

Maxime Boucher  
Sébastien Devaux  
dexter@detexia.com

**Résumé :** De nombreux travaux sont en cours visant à accélérer le démarrage d'un système GNU/Linux. Certains ont déjà annoncé des résultats impressionnants avec une durée réduite à cinq secondes. Ces résultats ont été obtenus au prix d'optimisations parfois assez agressives, pas totalement divulguées ou dans le cadre de nouvelle distribution et restent difficilement applicables à un système existant. Avec un objectif plus modeste, nous présentons notre approche basée sur une configuration du noyau et des services adaptée au matériel et à l'usage ciblé. Nous proposons aussi un système de parallélisation du démarrage des services qui s'appuie simplement sur GNU Make. Ces techniques ont été appliquées à des netbooks pour lesquels un démarrage rapide est un réel atout.

**Mots-clé :** logiciel libre, linux, boot, développement industriel, accélération, optimisation

## Table des matières

~ \* ~ \* ~

1	Présentation.....	3
1.1	Principe.....	3
1.2	Plateformes.....	3
2	Séquence de démarrage traditionnelle.....	4
2.1	Principales étapes du démarrage d'un PC.....	4
2.2	Initialisation du matériel par le BIOS.....	4
2.3	Le bootloader : Grub.....	4
2.4	Chargement du noyau.....	5
2.5	Le processus Init.....	5
3	Accélération de la séquence de démarrage.....	7
3.1	Où agir pour accélérer le démarrage.....	7
3.2	Grub en multi-boot et init alternatif.....	8
3.3	Un noyau sans initrd.....	9
3.3.1	Compilation d'un nouveau noyau.....	9
3.3.2	Création des entrée dans /dev.....	11
3.4	Parallélisation du lancement des services par GNUmake.....	12
3.4.1	Make et dépendances.....	12
3.4.2	Cibles et règles.....	12
3.4.3	Synchronisation et Désynchronisation des services.....	13
3.4.4	Mise en oeuvre via init.....	14
4	Résultats.....	15
4.1	Réduction de la durée de démarrage.....	15
4.2	Limites et effets indésirables.....	15
4.3	Améliorations et travaux à venir.....	15
	Table des URL .....	17

# 1 Présentation

~ \* ~ \* ~

## 1.1 Principe

L'accélération du démarrage d'un système GNU/Linux fait l'objet actuellement de nombreux travaux. Certaines équipes ont déjà annoncé des résultats spectaculaires avec une durée ramenée à quelque cing secondes <sup>[[url1](#)]</sup>. Cependant la majorité de ses méthodes sont assez agressives et reposent parfois sur une modification profonde du noyau lui-même. De plus malgré des démonstrations filmées prouvant la réalité des résultats, ils sont encore difficiles à reproduire car les modifications apportées n'ont pas souvent été publiées dans leur intégralité et il sera nécessaire d'attendre encore avant de voir leur intégration dans des distributions officielles.

Toutefois nous présentons dans ce document quelques actions plus ou moins aisées qui peuvent être menées pour réduire la durée du démarrage d'un système linux commun tel qu'on le trouve aujourd'hui par le biais des principales distributions. Ces actions, même si elles requièrent un minimum de connaissances, comme configurer et recompiler un noyau, restent assez faciles à intégrer. Pour parvenir à nos fins, nous avons développé nos recherches selon deux axes:

- Etablir une configuration du noyau adaptée au matériel et à l'usage ciblé : En effet, les distributions généralistes consacrent une bonne partie de leur démarrage à faire l'inventaire du matériel disponible pour activer les bons pilotes. IL n'est pas nécessaire de faire ce travail à chaque lancement puisque dans une utilisation courante la configuration matérielle reste assez stable et les changements de configuration sont ponctuels.
- Limiter les services et paralléliser leur démarrage : Cette idée n'a rien d'innovant en soi et plusieurs systèmes comme initNG <sup>[[url2](#)]</sup> proposent de se substituer complètement à l'ancien système de démarrage issu d'Unix system V pour mener à bien cette tâche. Nous proposons pour notre part de conserver l'ensemble des scripts de démarrages classiques mais d'en contrôler la séquence avec un Makefile qui grâce à l'option -j de GNU Make nous permet à très peu de frais d'obtenir un séquenceur capable de parallélisation tout en tenant compte de contraintes de dépendance entre certaines tâches.

## 1.2 Plateformes

Si l'accélération d'une étape qui a lieu en générale une fois par jour pour un poste bureautique et beaucoup plus rarement pour un serveur peut sembler vaine, elle apportera un réel confort d'utilisation pour les PC Portables et tous les appareils intermédiaires entre le PC portable, l'agenda électronique ou le téléphone mais aussi pour des systèmes plus ou moins embarqués et spécialisés comme des settopbox. C'est pourquoi nous avons choisi de tester nos optimisations sur des netbooks. Ce sont des PC ultra portables, permettant d'effectuer toutes les tâches traditionnelles qu'un ordinateur portable "classique" fait. La puissance de calcul d'un netbook est moindre face à celle d'un portable classique, mais reste comparable à celle d'un PC moyen de gamme d'il y a cinq ans. Pour cette étude, nous avons utilisé trois netbooks différents:

- ASUS eeePC 701 <sup>[[url3](#)]</sup> : processeur Intel celeron M (900MHz), RAM 512 Mo (DDR-2), SSD 4 Go, écran 7 pouces, système d'exploitation eeebuntu <sup>[[url4](#)]</sup>
- ASUS eeePC 901 <sup>[[url5](#)]</sup> : processeur Intel Atom N270 (1,6GHz), RAM 1 Go (DDR-2), SSD: 4 Go + 8 Go, écran 8,9 pouces, système d'exploitation eeedebian <sup>[[url6](#)]</sup>
- DELL inspiron mini 9 <sup>[[url7](#)]</sup> : processeur Intel atom (1,6GHz), RAM 1 Go (DDR-2), SSD 16 Go, écran 8,9 pouces, Système d'exploitation eeebuntu <sup>[[url8](#)]</sup>

## 2 Séquence de démarrage traditionnelle

~ \* ~ \* ~

### 2.1 Principales étapes du démarrage d'un PC

La mise sous tension d'un ordinateur de type PC provoque l'activation de plusieurs systèmes qui se passent la main successivement jusqu'à être prêt à recevoir les requêtes de l'utilisateur :

- initialisation et tests par le BIOS
- Chargement du "bootloader" du périphérique de masse principal.
- Chargement et initialisation du noyau du système d'exploitation.
- Chargement et démarrage des services et des tâches de fond.
- Activation de la console ou de tout autre dispositif d'interaction avec l'utilisateur.

### 2.2 Initialisation du matériel par le BIOS

Le BIOS ( Basic Input and Output System ) réside sur un circuit mémoire directement intégré au matériel qui est configuré par construction pour exécuter le programme qu'il contient avant toute chose. Sa première tâche est de contrôler la présence et la bonne marche des équipements matériels qui constituent l'ordinateur, il s'agit principalement d'un test de la mémoire, d'un inventaire des disques durs disponibles, et dans le cas courant d'un PC, faire l'inventaire des périphériques présents sur le bus PCI. Il recherche parmi les composants qu'il a découverts si certains embarquent un BIOS propre qui sera exécuté à son tour. Pour terminer, il va rechercher selon un ordre établi sur chaque périphérique de masse disponible la présence d'un programme d'amorçage (communément appelé bootloader). Dans le cas de linux les bootloaders les plus courants sont LILO et Grub, ce dernier tend à remplacer de plus en plus le premier.

### 2.3 Le bootloader : Grub

La plupart des distributions Linux actuelles, dont celles avec lesquelles nous avons travaillé utilisent Grub comme programme d'amorçage, c'est à dire un programme qui peut être chargé et exécuté depuis le BIOS afin de prendre en charge le démarrage de programme trop complexes pour pouvoir être activé directement par le BIOS. Le principal intérêt de Grub par rapport à ses concurrents est une grande souplesse de configuration. Par la simple édition d'un fichier ASCII, il est possible de décrire les caractéristiques de chaque système hébergé sur les mémoires de masse disponibles. Ce fichier autorisera selon les options activées le choix du système à démarrer au travers d'un menu convivial. vu d'un système Linux démarré, ce fichier de configuration est accessible en général par le chemin `/boot/grub/menu.lst`. Chaque système pouvant être démarré y sera décrit par une section ressemblant à celle-ci :

#### entrée standard grub

```
title Debian GNU/Linux, kernel 2.6.28
root (hd0,6)
kernel /boot/vmlinuz-2.6.28 root=/dev/sda7 ro
initrd /boot/initrd.img-2.6.28
```

- *title* donne le titre du système, c'est à dire son nom complet tel qu'il sera affiché par le menu Grub.
- *root* identifie le disque et la partition du disque qui contient le noyau de système d'exploitation à démarrer.
- *kernel* identifie le fichier du noyau à charger, son chemin étant relatif à la partition définie par *root*, des paramètres peuvent suivre leur signification étant propre au noyau.
- *initrd* identifie une image de système de fichier racine que le noyau pourra utiliser pour son processus de démarrage en attendant la disponibilité du système racine définitif.
- Pour les autres options et paramètres, le lecteur curieux pourra se reporter à la documentation de grub.

Selon les options de configuration, Grub montrera un menu qui liste chaque système présent et décrit comme montré ci-dessus. Lorsque l'utilisateur fait son choix, grub charge en mémoire le fichier du noyau correspondant à ce choix et déclenche son exécution.

## 2.4 Chargement du noyau

Suite à son propre chargement, le noyau effectue également des initialisations qui consistent principalement à faire lui-même un inventaire minimal des périphériques disponibles pour son propre fonctionnement, c'est à dire essentiellement, le(s) processeur(s), la mémoire vive, les périphérique de masse. Ceci fait, sa seule tâche sera alors de démarrer un premier processus qui sera le père de tous les processus suivants jusqu'à l'arrêt du système. Dans le cas d'un système de type Unix ce processus souvent appelé *init* se chargera à son tour de poursuivre la procédure d'initialisation et de démarrage.

## 2.5 Le processus Init

Le programme *init* "traditionnel" trouve ses instructions dans le fichier */etc/inittab* qui décrit en détail les actions qu'il doit mener pour initialiser le système. Init a également la faculté de gérer un indice d'état du système appelé niveau d'exécution (ou runlevel) qui permet de gérer la phases transitoires de la vie du système comme le démarrage lui-même. Il permet également d'avoir plusieurs mode de fonctionnement (exclusif) pour un même système. Par convention les niveaux d'exécution disponibles par défaut ont été établi ainsi :

- 0 : arrêt du système
- S : état transitoire d'initialisation minimale.
- 1 : mode mono-utilisateur
- 2 : mode standard multi-tâche, multi-utilisateur
- 3 : mode standard multi-tâche, multi-utilisateur, en réseau.
- 4 : variable selon les systèmes, assez rarement utilisé.
- 5 : mode graphique multi-tâche, multi-utilisateur, en réseau.
- 6 : arrêt pour redémarrage.

Dans beaucoup de distribution linux récente, la distinction entre les modes 2, 3, 4 et 5 tend à disparaître, et le mode de démarrage par défaut est 2 pour lequel l'interface graphique et le réseau sont actifs.

Le fichier */etc/inittab* décrit les actions à mener à l'entrée de chaque niveau d'exécution, il est constitués de lignes elles mêmes constituées de quatre champs séparés par deux points :

- champ 1 : identification unique de la ligne, en général sur deux caractères, certains identifiants pouvant être réservés pour un usage particulier selon les version du programme *init*.
- champ 2 : niveaux d'exécution à l'entrée desquelles la commande qui suit doit être traitée.
- champ 3 : mode d'enchaînement de la commande avec les suivantes actives dans le(s) même(s) niveau(x).
- champ 4 : définition de la commande à exécuter.

Une ligne d'un type particulier (champ 3 = *initdefault*) et d'identifiant lui aussi réservé (champ 1 = *id*) indique le niveau d'exécution à atteindre.

### extrait fichier inittab system V

```
# The default runlevel.
id:2:initdefault:

# Boot-time system configuration/initialization script.
# This is run first except when booting in emergency (-b) mode.
si::sysinit:/etc/init.d/rcS
```

```
10:0:wait:/etc/init.d/rc 0
11:1:wait:/etc/init.d/rc 1
12:2:wait:/etc/init.d/rc 2
13:3:wait:/etc/init.d/rc 3
14:4:wait:/etc/init.d/rc 4
15:5:wait:/etc/init.d/rc 5
16:6:wait:/etc/init.d/rc 6
```

Dans l'exemple qui précède, le fichier `/etc/inittab` commande à `init` d'atteindre le niveau 2 après avoir réalisé les commandes de l'état transitoire `sysinit`. On remarque que pour tous les niveaux, il est demandé de lancer la commande `/etc/init.d/rc X` où `X` est le niveau d'exécution. Le script `/etc/init.d/rc` propose un mécanisme particulier pour déterminer les services à démarrer selon les niveaux d'exécution d'origine et de destination qui repose sur les conventions suivante :

- Chaque service à piloter dispose dans le répertoire `/etc/init.d` d'un script de contrôle qui accepte au moins un argument dont la valeur peut être `start` ou `stop`.
- Pour chaque niveau d'exécution `X`, il existe un répertoire `/etc/rcX.d` qui contient des liens symboliques de la forme `ANNom_service` vers `/etc/init.d/nom_service` avec :
  - `A` : le caractère `S` ou `K` qui sera respectivement traduit en argument `start` ou `stop`
  - `NN` : un numéro d'ordre à deux chiffres.

Le script `rc` consulte le répertoire de liens correspondant au niveau reçu dans son propre argument pour y exécuter dans l'ordre établie par les deux chiffres avec l'argument déterminé par le premier caractère de chaque lien. Ceci permet pour chaque niveau d'exécution de définir quels sont les services à démarrer, les services à arrêter et dans quel ordre il faut mener ces opérations.

En conclusion, la tâche principale d'`init` est de déléguer le séquençage des actions requise lors d'une transition de niveau d'exécution à un automate séquentiel dont le comportement est déterminé par l'ensemble des scripts et liens symboliques présents dans les répertoires `/etc/rc*.d` et `/etc/init.d`.

## 3 Accélération de la séquence de démarrage.

~ \* ~ \* ~

### 3.1 Où agir pour accélérer le démarrage

Chacune des étapes présentées au chapitre précédent peut recevoir son lot d'optimisation en vu d'accélérer le processus global. En effet ces étapes sont complètement séquentielles et relativement étanches, le seule dépendance étant qu'un échec total aboutissant au blocage de l'une d'entre elle empêchera la poursuite du processus. A l'inverse il n'y a pas de raison que la modification d'une étape ait un impact majeur sur les suivantes dès que toutes les tâches dont elle a la responsabilité ont été bien menées. Les possibilités d'action varient bien entendu selon les étapes :

- Le BIOS : tout le monde a déjà constaté, qu'au démarrage, un PC consacre plusieurs secondes à son réveil, de la mise sous tension à l'entrée en scène du bootloader. Étrangement, malgré les progrès spectaculaires réalisés par la micro informatique, cette étape n'a guère évolué, elle s'est même dans certains cas dégradé et l'action du BIOS d'un PC actuel n'est pas plus rapide que celle d'un PC d'il y a dix ans, ni même d'il y a vingt ans. La seule solution pratique aujourd'hui est de remplacer complètement le BIOS livré avec le PC par un autre puisqu'il n'est pas prévu de pouvoir modifier le BIOS en place. Le projet [coreboot](#)<sup>[url<sup>9</sup>]</sup> (anciennement LinuxBIOS) est un BIOS alternatif qui de par sa nature ouverte est personnalisable selon son besoin ce qui pourra conduire à une accélération si on désactive certaines fonctionnalités qui demandent du temps mais dont on a pas l'usage.
- Le bootloader : sa durée est quasi insignifiante, elle sera même artificiellement allongée pendant notre démarche expérimentale afin de laisser à nous autres pauvres être humains à la rapidité de réaction trop limitée, le temps de choisir un mode de démarrage alternatif en cas de défaillance de notre mode de démarrage accéléré.
- Le chargement du noyau Linux : le démarrage du noyau lui même est devenu un processus potentiellement complexe de part la nécessité pour les distributions généraliste de pouvoir démarrer avec n'importe quelle configuration matériel. A chaque démarrage et à différentes étapes une auto détection des périphériques s'avère nécessaire et certaines acrobaties sont requises pour que le chargement du noyau lui même ne soit pas entravé par l'incapacité dans les étapes préliminaires d'accéder à certains périphériques critiques, dont la mémoire de masse principale qui elle-même héberge le noyau et ses drivers. Dans beaucoup de cas ce jeu de devinette consommateur de temps est d'une utilité limitée car la configuration matériel de l'ordinateur change à chaque lancement. Il est donc possible d'accélérer le démarrage du noyau en le configurant spécifiquement pour le matériel à prendre en charge sur un poste donnée afin d'inclure en statique et démarrer directement les bons pilotes de périphérique.
- init et scripts de démarrage système V : le lancement de tous les service est séquentiel et certain d'entre eux ayant recours à des ressources à accès plus ou moins rapide (disque, réseau, ...), ont un taux d'utilisation du processeur assez réduit, même pendant leur phase de démarrage. De plus, beaucoup sont totalement indépendant les uns des autres ce qui autorise leur démarrage sur un mode asynchrone simultané au lieu de complètement séquentiel. Cette idée de parallélisation des script de démarrage a motivé la création du projet [initNG](#) qui vise à remplacer l'ensemble init, script de démarrage pour accomplir cette tâche. Nous allons voir dans la suite qu'il est possible d'arriver à un résultat très proche par une réutilisation astucieuse de l'existant.
- Enfin chaque service peut lui même être optimisé afin d'être disponible au plus tôt, en particulier le serveur graphique et l'environnement de bureau qui forment l'interface principale de dialogue avec l'utilisateur. Il s'agit d'un travail spécifique à chaque service que nous n'aborderons pas ici. On peut considérer ce procédé comme une tricherie car on ne fait que déplacer le problème mais cela permet

de répondre à l'objectif premier qui est de pouvoir disposer au plus tôt d'un système fonctionnel. Mais ce sera le cas uniquement si les service dont le démarrage est retardé n'a aucune incidence sur ce que perçoit l'utilisateur, c'est à dire que cette activité ne doit pas empêcher l'utilisateur d'utiliser immédiatement sa machine avec une réactivité normale (éviter un sablier à chaque click). Par exemple il n'y a rien de pénalisant à ce qu'un serveur ssh ou ftp soit démarré quelques secondes après l'ouverture de la session graphique de l'utilisateur.

Le noyau Linux a été critiqué quasiment depuis sa naissance en raison de son architecture monolithique. Même si un peu de modularité a été introduite depuis, notamment par le mécanisme des modules chargeables, son initialisation interne reste très séquentielle bien que des capacités de multi-threading soit disponibles assez tôt dans la vie du noyau. Des travaux sont en cours pour faciliter un mode de démarrage asynchrone des sous ensembles du noyau <sup>[url10]</sup>.

## 3.2 Grub en multi-boot et init alternatif

Pendant nos expérimentations, nous avons été menés à redémarrer linux sur des systèmes de boot plus ou moins stables, ainsi par mesure de sécurité, nous avons créé un clone d'*init* légèrement modifié pour ne plus recevoir ses instructions du fichier */etc/inittab-finit*, mais d'une copie de *inittab* que nous pourrions modifier sans problème. Cela nous permet de créer une entrée spécifique dans le menu grub pour démarrer notre noyau linux personnalisé en précisant également de démarrer comme premier processus la commande */sbin/finit* à la place de */sbin/init*, tout en conservant l'entrée relative au système d'origine. Cette capacité de choisir entre le processus de boot standard ou celui expérimental permet un retour au système d'origine pour continuer d'accéder à tous les fichiers de notre disque dans le cas où notre système expérimental n'est pas en mesure d'achever son démarrage.

La procédure suivie pour produire notre *init* alternatif est :

- Récupérer les fichiers sources d'*init* ([sysvinit.tar.gz](#)). En principe toute distribution linux doit donner un accès au sources. S'ils s'avèrent difficile à trouver, on pourra se reporter au projet [Linux From Scratch](#) <sup>[url11]</sup> qui publie les sources d'un système GNU/Linux complet ainsi que la procédure permettant de le construire intégralement à partir de ces sources.
- Décompresser l'archive.
- Ouvrir le dossier *src/* puis éditer le fichier *path.h* avec votre éditeur de texte favori pour y remplacer la ligne

```
#define INITTAB "/etc/inittab"
```

par

```
#define INITTAB "/etc/inittab-finit"
```

- Compiler le programme à l'aide de la commande *make*.
- Une fois la compilation terminée, le nouveau fichier *init* se trouve dans le dossier. Renommons le en *finit* (pour ne pas avoir de conflit avec *init*). Il suffit maintenant de le copier dans le dossier */sbin/*.
- La dernière étape consiste à éditer le menu de *Grub* (*/boot/grub/menu.lst*) pour lui indiquer qu'il doit utiliser le fichier *finit* au lieu d'*init*. Il faut donc ouvrir le fichier *menu.lst*, rendez-vous à la fin de ce fichier. comme je l'ai cité précédemment il existe plusieurs entrée de cette forme:

### entrée standard grub

```
title Debian GNU/Linux, kernel 2.6.28
root (hd0,6)
kernel /boot/vmlinuz-2.6.28 root=/dev/sda7 ro
initrd /boot/initrd.img-2.6.28
savedefault
```

Copier l'entrée que vous utilisez habituellement. Modifiez en le titre comme bon vous semble ( pour pouvoir différencier l'entrée d'origine de la nouvelle). On ajoute l'option *init=/sbin/finit* à la ligne *kernel*, pour indiquer que l'on souhaite utiliser la nouvelle commande *init* modifiée à la place de celle d'origine :

#### entrée modifiée grub

```
title Fast boot
root (hd0,6)
kernel /boot/vmlinuz-2.6.28 root=/dev/sda7 ro init=/sbin/finit
initrd /boot/initrd.img-2.6.28
savedefault
```

### 3.3 Un noyau sans initrd

Comme nous l'avons rapidement évoqué précédemment, les noyaux des distributions généralistes doivent pouvoir affronter toutes les situations pour mener leur démarrage à terme. Cela est particulièrement épineux quand un pilote particulier est requis pour accéder à la racine du système de fichier. Pour surmonter cette difficulté, il est possible de préciser dans la configuration du bootloader (cf paramètre *initrd* dans le fichier de configuration de *grub*), qu'une image de système de fichier racine minimale est disponible. Cette image contiendra le minimum vital pour le bon démarrage du système en toute circonstance, c'est à dire essentiellement l'ensemble des pilotes de périphériques. Pendant son chargement, le noyau monte le système de fichier contenu dans cette image en tant que racine temporaire qui lui donne accès à tout ce dont il a besoin pour réaliser son démarrage jusqu'à être en mesure de remplacer ce système de fichier racine temporaire par le système de fichier racine définitif. Au bout du compte, ce mécanisme consomme obligatoirement du temps supplémentaire et n'a que peu d'intérêt dans un environnement où la configuration est maîtrisée et stable. Cela apporte seulement le confort de ne pas avoir à configurer ni compiler quoi que ce soit lors d'une installation d'un nouveau système (ce qui reste vital pour rendre GNU/Linux accessible au plus grand nombre).

En fait, la plupart des pilotes de périphérique peuvent se présenter sous deux formes :

- Module chargeable et déchargeable à chaud (cf commandes *modprobe*, *lsmod* et *rmmod*)
- Module statique : son code sera inclus directement dans l'image amorçable du noyau (cf. fichier */boot/vmlinuz-<version>* qui est l'emplacement du noyau pour la plupart des distributions).

Inclure les pilotes des périphériques de la plateforme ciblée accélère le démarrage par ces deux effets :

- Si tous les pilotes requis pour accéder à la partition racine sont inclus, il n'y a plus aucune raison de consacrer du temps à travailler avec une image temporaire, puisqu'il devient ainsi possible de monter directement la bonne racine du système de fichier.
- Les pilotes inclus sont de fait chargés avec le noyau et il n'est plus nécessaire de faire un inventaire et une recherche de ces pilotes déjà chargés. On ne conservera la fonctionnalité de chargement dynamique, uniquement pour la prise en charge des périphériques amovibles qui par nature ne peuvent pas être initialisés avant leur utilisation effective.

En pratique, on ne pourra pas toujours inclure dans le noyau, tous les pilotes correspondant au matériel "fixe" de l'ordinateur car cela produirait une image amorçable de noyau trop volumineuse pour pouvoir démarrer correctement (Vérifier que cette affirmation est toujours vraie pour les noyaux et les bootloaders récents)

#### 3.3.1 Compilation d'un nouveau noyau

La première chose à faire est de se procurer les sources d'un noyau linux. Le site [Kernel.org](http://Kernel.org)<sup>[url12]</sup> publie les sources officiels maintenus par Linus Torvalds et en archive toutes les versions. Lors de l'écriture de cette note, la dernière version est la 2.6.28.4. Attention les sources sont publiées sous deux formes :

- Archive complète (full)
- Différentiel depuis la version antérieure (patch)

On prendra donc soin lors d'un premier téléchargement de choisir une archive complète. Une fois l'archive décompressée, se rendre dans son répertoire racine et utiliser la commande *make menuconfig* pour accéder à un menu de configuration relativement convivial qui permet de choisir tous les options et le mode de compilation des pilotes qui présente trois possibilité :

- Y : pour la compilation en mode statique, on choisira de préférence cette option pour tous les pilotes requis au démarrage conformément à la démarche exposée ci-avant.
- M : compilation sous forme de module chargeable.
- N : ne pas compiler le module. Il sera alors inutilisable.

On utilisera également ce menu pour donner un suffixe personnel à notre noyau ce qui évitera d'écraser le noyau éventuellement déjà présent dans la même version sur notre poste de travail.

A l'issue de l'utilisation du menu de configuration, il est créé un fichier *.config* (attention, selon les conventions unix, c'est un fichier caché) qu'on pourra sauvegarder pour une utilisation future et dont on pourra aussi contrôler et affiner le contenu. Chaque option présentée dans le menu de configuration y est retranscrite par une ligne :

- # NOM\_DU\_MODULE is not set : signifie que le module ne sera pas compilé.
- NOM\_DU\_MODULE=y : signifie que ce module sera compilé en statique.
- NOM\_DU\_MODULE=m : signifie que ce module sera compilé en tant que module chargeable (non statique).

Toutes nos plateformes de test ont une architecture relativement identique, à base de processeur Pentium M ou de processeur Atom, accompagné du chipset intel (i914 ou i945) et d'un contrôleur de disque SATA. Nous avons par conséquent ajusté les options de compilation selon ces paramètres en particulier pour inclure les pilotes pour les puces intel, pour les périphériques SATA et le système de fichier de nos distributions linux, à savoir ext3. Le boot sans *initrd* sera donc possible si les options suivantes ont pour valeur "y" dans le fichier *.config* : *CONFIG\_BLK\_DEV\_IDE\_SATA*, *CONFIG\_SCSI*, *CONFIG\_BLK\_DEV\_SD*, *CONFIG\_SCSI\_SAS\_ATA*, *CONFIG\_ATA*, *CONFIG\_ATA\_PIIX*, *CONFIG\_EXT3\_FS*. Les pilotes SCSI sont inclus car l'architecture de pilote SATA de Linux consiste à les présenter de la même manière que des périphériques SCSI.

Une fois la configuration établie, nous pouvons accomplir la compilation du noyau :

```
make
make modules
make install
make modules_install
```

Attention pour effectuer les deux dernières commandes, il est nécessaire de posséder les droits administrateur (cf. commande *sudo*).

On vérifiera que le noyau et ses modules ont bien été installés par une vérification du contenu des répertoires */boot* et */lib/modules*. On terminera par ajuster si besoin la configuration de *grub* (fichier */boot/grub/menu.lst*) pour prendre en compte ce nouveau noyau :

#### Entrée Grub nouveau noyau

```
title Noyau experimental Dexter sans initrd
root (hd0,6)
kernel /boot/vmlinuz-2.6.28.4-dexter init=/sbin/finit root=/dev/sda1 ro
savedefault
```

Selon la distribution GNU/Linux de base choisie pour ces expériences, ceci ne sera peut-être pas suffisant pour charger correctement le noyau et relayer le démarrage au programme *init* tel que nous l'avons modifié.

Il pourra être nécessaire de procéder un ajustement du contenu de `/dev` tel que cela est exposé au paragraphe suivant.

### 3.3.2 Création des entrée dans `/dev`

Sous Unix, tout est fichier (ou presque), ainsi tous les périphériques sont présentés sous la forme d'un ou plusieurs pseudo fichiers hébergés dans le `/dev`. Le travail d'un pilote consiste en fait à traduire les opérations de lecture et écriture classiques sur ces fichiers en ordres compréhensibles par le matériel qu'il prend en charge. Auparavant tout le contenu de `/dev` était établi une bonne fois pour toute, et on y trouvait une multitude de fichiers pour tous les périphériques potentiellement pris en charge par le noyau. Maintenant, le démon `udev` propose une gestion dynamique de son contenu pour ajouter et créer les entrée de `/dev` au fur et à mesure de l'apparition ou de la disparition des périphériques. Cependant nous avons besoin d'un minimum de chose avant l'activation de `udev` lui même, en particulier pour pouvoir accéder au premier disque de boot qui contient en générale le système de fichier racine.

Le plus simple consiste à relever les entrées correspondant à ce disque et ses partitions directement depuis le système de base avant de le redémarrer pour tester notre noyau personnalisé. On pourra s'aider de la commande `fdisk -l` pour les identifier : tous les disques présents et les partitions associées vont apparaitre à l'écran avec leurs noms. Conventionnellement, le premier disque est appelé `sda` et ses partitions `sda1 sda2 ...` Le deuxième disque `sdb` et ses partitions `sdb1 sdb2...` Rendez-vous maintenant dans `/dev/` et regardons son contenu (commandes `ls -l`). Nous retrouvons des fichiers se nommant `sda, sda1, sdb, sdb1, etc`, et observons les informations importantes sur une ligne :

#### entrée type dans `/dev`

```
brw-rw---- 1 root disk      8,    0 2009-02-10 11:07 sda
```

- `brw-rw----` : Le `b` indique le type de périphérique `b=block`, `rw-rw---` indique les droits d'accès (voir commande `chmod`).
- `root disk` : `root` est l'utilisateur propriétaire, ici il s'agit de l'administrateur, et `disk` est le groupe propriétaire
- `8, 0` : `8` est le numéro majeur du périphérique et `0` le numéro mineur. Ce sont les numéros logiques d'identification du périphérique pour le noyau. Ces numéros sont importants ils nous servirons par la suite. Donc ici le numéro majeur de `sda` est `8` et le mineur est `0`. Donc `sda1` (première parition de `sda`) aura le même numéro majeur mais `1` comme numéro mineur. Même principe pour `sda2`, même majeur mais `2` comme mineur, et ainsi de suite.
- `sda` : correspond au nom de l'entrée

Mais comment créer ces entrée si elle sont déjà présente ? Comme nous il a été dit plus haut, le contenu de `/dev` est maintenant géré par le démon `udev` et rien ne dit que les fichiers sont déjà présents avant l'activation de `udev`. Nous avons besoin d'accéder au répertoire `/dev/` de la partition racine sans que `udev` n'y soit actif. Pour cela il y a deux possibilité :

- Utiliser un livecd pour démarrer un nouveau système depuis lequel on pourra accéder à cette partition pour y vérifier et ajuster son contenu
- Extraire le disque pour l'insérer en tant que disque secondaire d'un autre système Linux.

Utiliser la commande `mount` pour accéder à la partition puis ajuster si besoin le contenu du répertoire `dev` qu'elle contient avec la commande `mknod` :

#### syntaxe `mknod`

```
mknod -m <droit sur le fichier> <nom du fichier> <type de périphérique> <numéro
majeur> <numéro mineur>
```

#### création `sda`

```
mknod -m 0660 sda b 8 0
```

Cette commande va donc créer l'entrée sda de type block avec 8 comme numéro majeur et 0 comme mineur. Il est évident qu'il faut adapter les paramètres à sa configuration. Pour créer sda1 il suffit de modifier le nom et de mettre 1 à la place de 0. Ainsi de suite jusqu'à sda9. Une dernière manipulation est à faire, il faut mettre toutes les entrées créées dans le groupe *disk* en utilisant la commande suivante :

#### changement de groupe

```
chgrp <nom du groupe> <nom du fichier>*
```

C'est à dire, dans notre cas : *chgrp disk sda\**.

Après un démontage propre de la partition (voir commande *umount*), le système est prêt pour l'étape suivante.

## 3.4 Parallélisation du lancement des services par GNUMake

### 3.4.1 Make et dépendances

Pour comprendre la suite, des connaissances de base en ce qui concerne la syntaxe et l'utilisation de Make sont nécessaire. Make est un outil permettant l'exécution automatique d'une séquence de tâche dont la nécessité et l'ordre sont déterminé par des règles de dépendance. Si cet outil a été inventé par des programmeurs pour des programmeurs, rien ne le restreint à son principal emploi qui est la compilation de logiciels. Il peut être employé pour toute autre tâche à partir du moment où elle repose sur une séquence de commande avec des contraintes d'enchaînement. Nous l'employons d'ailleurs également pour automatiser la mise en forme de la présente documentation.

Lorsque nous avons brièvement abordé le processus de démarrage par *init* dans un paragraphe précédent, nous avons vu qu'il s'agit essentiellement d'exécuter dans un ordre précis des scripts hébergés dans */etc/init.d*. Ceci est justement une tâche pour laquelle *make* a été conçue. Mais qu'en est-il de la parallélisation ? Il s'avère que GNUMake, qui est la déclinaison GNU du *make* présente par défaut sur la plupart des systèmes Linux, possède une option très intéressante lui donnant la capacité de piloter en parallèle de lancement des commandes tout en préservant les dépendances. C'est à dire qu'une tâche particulière ne sera démarrée que si toutes les tâches dont elle dépend ont été complètement réalisées avec succès. Il suffit donc d'établir le bon *Makefile*, c'est à dire rédiger dans la syntaxe *make*, la liste des dépendances entre les services et indiquer pour chaque niveau d'exécution quels sont les services "finaux" à démarrer. C'est à dire, les services qui doivent être actifs mais donc aucun autre service ne dépend.

### 3.4.2 Cibles et règles

Dans un *makefile*, le travail à réaliser est décrit par un ensemble de règles qui définissent les actions à mener pour réaliser une cible tout en précisant les prérequis. Nous allons d'abord rédiger un premier *Makefile* principal que nous nommerons */etc/init.make* et qui sera invoqué par *init* via le fichier */etc/inittab-finit* qui recueillera un ensemble de règles génériques. Un deuxième *makefile* précisera de son côté quels sont les services finaux attendus pour chaque niveau d'exécution et les dépendances entre services.

#### */etc/init.make*

```
DIR=/var/finit

all: dolevel$(LEVEL)
@ echo "Level $(LEVEL) initializations done."

%.s: /etc/init.d/%
@ $< start 2>&1 | tee $(DIR)/$@.log

%.s: /etc/init.d/%.sh
@ $< start 2>&1 | tee $(DIR)/$@.log

%.k: /etc/init.d/%
```

```
@ $< stop 2>&1 | tee $(DIR)/$@.log

%.k: /etc/init.d/%.sh
@ $< stop 2>&1 | tee $(DIR)/$@.log

mod.%:
@ modprobe $(patsubst mod.%,%, $@)

include /etc/init.d/rules.make
```

- La première cible *all* est la cible par défaut, c'est à dire la cible à atteindre si on ne précise pas de cible explicitement à make. Elle ne demande que la réalisation de la cible *dolevelX* avec X le niveau d'exécution qui sera transmis en paramètre par le biais de la variable *LEVEL*
- Suit un ensemble de quatre règles dont la signification est la suivante : les cibles *service.s* et *service.k* requièrent la présence d'un script dont le nom est *service* ou *service.sh* dans le répertoire */etc/init.d* (ce qui est le cas pour tous les service courants de la plupart des distributions GNU/Linux). Sa réalisation consiste simplement à exécuter le script avec pour paramètre start ou stop selon le suffixe de la cible. La sortie et l'erreur standard sont redirigées dans un fichier en plus de la console pour faciliter la mise au point.
- la règle *mod.module* permet le chargement explicite d'un module noyau. Ceci permettra de conditionner le démarrage d'un service au chargement préalable d'un pilote de périphérique.
- Enfin l'inclusion du fichier */etc/init.d/rules.make* permet de prendre en compte toutes les règles qui y sont consignées.

### 3.4.3 Synchronisation et Désynchronisation des services

Le makefile contenant les règles de dépendance suit le principe générale de fonctionnement de make. Pour réaliser les traitements commandés par une règle, il faut d'abord que toutes les dépendances soit satisfaites, c'est à dire que toutes les cibles mentionnées dans la partie droite de la règle soient atteintes. Par contre cela n'impose rien sur l'ordre de traiter les multiples cibles qui son regroupées en tant que dépendance d'une unique règle, et, si la commande make utilisée le permet, rien n'empêche de les traiter en parallèle. Ce qui est le cas pour la version GNU de make.

Supposons que nous ayons quatre services à démarrer : *serv0*, *servA*, *servB* et *serv9*. Les services *servA* et *servB* doivent attendre que *serv0* soit démarré pour s'initialiser à leur tour. De son coté *serv9* doit attendre que *servA* et *servB* soient tous les deux démarrés. On traduira ces contraintes dans la syntaxe make en s'appuyant sur les règles génériques exposées au paragraphe précédents ainsi :

#### Exemple règles de dépendance

```
dolevel2: serv9

servA.s: serv0.s

servB.s: serv0.s

serv9.s: servA.s servB.s
```

La première règle précise simplement que l'entrée dans le niveau d'exécution 2 demande le démarrage de *serv9*, comme lui même, par application de la dernière règle, requière que d'autres services soient démarrés. Par application transitives des règles, tous les services mentionnés dans cet exemple seront démarrés avec le respect des contraintes exposées plus haut.

On pourra éventuellement si on le souhaite conserver en parti l'ancien système de démarrage Unix system V. Lors de nos expérimentation nous n'avons pas souhaiter perturber l'arrêt ni le reboot du système. Nous

avons donc simplement renvoyé le traitement des niveaux d'exécution 0 et 6 vers le script qui s'en charge à l'origine :

#### délégation aux scripts system V

```
dolevel0:
/etc/rc.d/rc 0

dolevel6:
/etc/rc.d/rc 6
```

### 3.4.4 Mise en oeuvre via init

Il reste maintenant à relier nos makefile à *init* pour que la commande *make* soit correctement invoquée à chaque transition de niveau d'exécution :

#### /etc/inittab-finit

```
# Le niveau d'exécution par défaut
id:5:initdefault:

# Initialisations préliminaires
si::sysinit:/usr/bin/make -j -r -i -f /etc/init.make LEVEL=SI

# Pour chaque niveau d'exécution, appliquer les règles...
10:0:wait:/usr/bin/make -j -r -i -f /etc/init.make LEVEL=0
11:1:wait:/usr/bin/make -j -r -i -f /etc/init.make LEVEL=1
12:2:wait:/usr/bin/make -j -r -i -f /etc/init.make LEVEL=2
13:3:wait:/usr/bin/make -j -r -i -f /etc/init.make LEVEL=3
14:4:wait:/usr/bin/make -j -r -i -f /etc/init.make LEVEL=4
15:5:wait:/usr/bin/make -j -r -i -f /etc/init.make LEVEL=5
16:6:wait:/usr/bin/make -j -r -i -f /etc/init.make LEVEL=6

# quelques consoles textes sont toujours utiles
1:12345:respawn:/sbin/getty 38400 tty1
2:2345:respawn:/sbin/getty 38400 tty2
3:2345:respawn:/sbin/getty 38400 tty3
4:2345:respawn:/sbin/getty 38400 tty4
```

Pour chaque niveau d'exécution la même commande *make* est invoquée, la variable *LEVEL* précisant la valeur du niveau à prendre en compte. Quelques options supplémentaires ont été utilisées :

- *-j* : option indispensable si on souhaite effectivement paralléliser le démarrage des services qui ne sont pas interdépendants.
- *-r* : suppression des règles implicites habituellement définies par défaut. Cette option n'est pas strictement requise, il s'agit seulement de s'assurer que seules les règles explicitement présentes dans le Makefile sont prise en compte.
- *-i* : continuer le traitement même en cas d'échec d'une règle. En toute rigueur, on devrait à terme ne pas employer cette option, mais il ne faut pas oublier que dans le cas de *make*, par défaut, si une cible n'est pas satisfaite, tout ce qui en dépend n'aura jamais lieu, ce qui est un comportement un peu trop strict dans la phase actuel d'étude de cette technique.
- *-f* : précise le makefile à traiter, par défaut la commande *make* recherche un fichier baptisé *Makefile* dans le répertoire courant.

On pourra également conserver quelques entrées issues d'un fichier *inittab* pour conserver la faculté de redémarrer le système par la combinaison CTRL-ALT-SUPPR ou tout autre définition qui n'est pas directement liée au processus normal de démarrage et d'arrêt du système.

## 4 Résultats

~ \* ~ \* ~

### 4.1 Réduction de la durée de démarrage

Configurations testées avec la distribution eebuntu :

- Dell mini 9 :
  - Sans optimisations : 45 secondes.
  - Avec optimisations : 19 secondes.
- ASUS eee PC 701 :
  - Sans optimisations : 55 secondes.
  - Avec optimisations : 22 secondes.

Les optimisations incluent une configuration et compilation du noyau afin de ne plus utiliser d'image de démarrage *initrd* ainsi que la sélection et la parallélisation des services par *GNUMake*. Ces deux axes ont été suffisants pour réduire la durée de démarrage de plus de sa moitié ce qui représente déjà un gain confortable et appréciable au quotidien sans avoir eu besoin de recourir à des modifications profondes de la distribution utilisée comme base de départ. Il est donc relativement aisé d'améliorer la durée de démarrage de la plupart des distributions GNU/Linux actuelles. Il subsiste encore quelques anomalies qui ne permettent pas encore l'application sur un système en production.

Les fichiers de configurations complets (inittab, makefiles, configuration du noyau) sont disponibles dans un [fichier archive](#)<sup>[ur13]</sup> librement téléchargeable.

### 4.2 Limites et effets indésirables

Les contraintes de démarrages des services udev, dbus et hal n'ont pas encore été suffisamment bien cernées et nous sommes confrontés au dilemme de devoir soit augmenter considérablement la durée du démarrage, soit nous contenter de fonctionnalités limitées de ces services, c'est à dire qu'en l'état actuel, certains aspect dynamiques de la gestion du matériel ne sont pas opérationnels. La principale lacune étant que le montage et l'ouverture automatique des périphériques de masse USB ne fonctionnent pas.

L'affichage pendant le démarrage est très chaotique puisque nous avons volontairement supprimé tout affichage graphique de barre de progression et les comptes rendus textuels du démarrage des services sont illisibles. Ils sont imbriqués et mélangés en raison de la parallélisation des démarrages ce qui entraîne des écritures simultanées sur la console par plusieurs programmes.

Enfin, le gain de performance général même s'il est très significatif, n'est pas au niveau de ce qui a été réalisé par d'autres et nous sommes encore assez loin des quelques cinq secondes qui semble être le meilleur résultat atteignable à ce jour pour un système assez complet pour une utilisation de type bureautique.

### 4.3 Améliorations et travaux à venir

- Poursuivre l'étude des scripts de démarrage présents dans */etc/init.d* afin d'y apporter si possible quelques optimisations et restaurer un fonctionnement correct de *hal* pour pouvoir profiter à nouveau de la gestion automatique des support USB amovibles.
- Etudier l'impact que peut avoir un système de fichier optimisé pour les disques SSD présents sur nos plateforme d'étude sur les performances au démarrage et pour l'utilisation générale du système.
- Un dernier point mériterai une attention particulière, mais qui ne pourra pas être complètement résolu sans une bonne volonté des fabricants de matériel, est la durée qui s'écoule entre l'activation du bouton de mise sous tension et le démarrage du GRUB. tout ce temps dépend tout particulièrement du BIOS dont la seule alternative envisageable est le projet [coreboot](#)<sup>[ur14]</sup>. A titre d'information, voici les durées mesurées pour cette étape sur nos plateformes de test :

- Asus eee PC 701 : 10 secondes
- Dell mini 9 : 10 secondes

Ceci représente en particulier presque le double de la durée de démarrage de la distribution moblin  
[url15] seule (de grub jusqu'à affichage de la session graphique : 6 secondes) sur la plateforme Dell.

## Table des URL

~ \* ~ \* ~

- [url1] *cinq secondes* : <http://lwn.net/Articles/299483/>
- [url2] *initNG* : <http://www.initng.org/>
- [url3] *ASUS eeePC 701* : <http://www.eee-pc.fr/faq-eeepc/asus-eeepc-701/>
- [url4] *eeebuntu* : <http://eeebuntu.org/>
- [url5] *ASUS eeePC 901* : [http://www.asus.fr/event/Eeepc\\_901-1000/specifications.html](http://www.asus.fr/event/Eeepc_901-1000/specifications.html)
- [url6] *eedebian* : <http://wiki.debian.org/DebianEeePC/HowTo/Install>
- [url7] *DELL inspiron mini 9* : <http://www.zdnet.fr/produits/specifications/dell-inspironmini-9-39383538.htm>
- [url8] *eeebuntu* : <http://eeebuntu.org/>
- [url9] *coreboot* : <http://www.coreboot.org/>
- [url10] *démarrage asynchrone des sous ensembles du noyau* : <http://lwn.net/Articles/314808/>
- [url11] *Linux From Scratch* : <http://www.linuxfromscratch.org>
- [url12] *Kernel.org* : <http://www.kernel.org>
- [url13] *fichier archive* : [http://dexter.detexia.net/files/AFF065-2009-NOT092\\_FBboot.tar.gz](http://dexter.detexia.net/files/AFF065-2009-NOT092_FBboot.tar.gz)
- [url14] *coreboot* : <http://www.coreboot.org/>
- [url15] *moblin* : <http://www.moblin.org/>