Démarrage d'un système Linux

M2 SETI B4 / MS SE SE758

Guillaume Duc
guillaume.duc@telecom-paris.fr
2020–2021

Grandes étapes du démarrage

- Bootloader
- Noyau Linux
- Processus Init

Dans la suite de ce cours, nous allons entrer dans les détails de ces différentes étapes, en prenant pour exemple, sauf mention contraire, un système à base de processeur ARM Cortex-A 32 bits (représentatif de nombreux systèmes embarqués)

Plan

Responsabilités

- En général, première couche logicielle s'exécutant sur une plate-forme
- Responsabilités classiques
  - Initialisation de base de la plate-forme
  - Démarrage des autres couches logicielles (en fonction du cas : système d'exploitation ou application principale)
  - Mise à jour (de lui même ou des couches logicielles)
- Exemples courants
  - Das U-Boot (embarqué)
  - GRUB (PC)
  - Windows Boot Manager (PC)

Avant le bootloader

- Sur certains systèmes, du code s'exécute avant le bootloader
- Exemple, sur un Cyclone V SoC [1]
  - *Boot ROM* (situé en mémoire morte sur le Cyclone V)
    - Détecte la localisation des étapes suivantes (cavaliers)
    - Réalise l'initialisation minimale du HPS
    - Charge l'étape suivante depuis la flash vers l'OCRAM (*On-Chip RAM*)
  - *Preloader*
    - Suite de l'initialisation du HPS
    - Initialisation de la SDRAM
    - Charge le bootloader vers la SDRAM et l'exécute
    - Peut être effectué par le *Secundary Program Loader* (SPL) d'U-Boot

Avant le bootloader

- Autre exemple, sur un PC :
  - BIOS/UEFI
  - Option ROMs
  - Bootloader (Windows Boot Manager ou GRUB par exemple)

Das U-Boot
Présentation

- `https://www.denx.de/wiki/U-Boot/`
- Bootloader extrêmement flexible et configurable
- Nombreuses architectures supportées : PPC, ARM, MIPS, x86, m68k, NIOS, Microblaze...
- Lecture d'un exécutable/image à charger depuis un support de stockage de masse (carte SD, SATA, SPI, I^2C, USB...), depuis une liaison série ou depuis le réseau (TFTP, NFS)
- Support de nombreux systèmes de fichiers (ext2/3/4, FAT, ZFS, JFFS2...)
- Support des arbres de périphériques (possibilité de les modifier à la volée)

Das U-Boot
Découpage

- Découpage possible en deux parties
- *Secundary Program Loader*
  - Exécuté depuis un environnement contraint (peu de RAM par exemple)
  - Prépare le système pour le chargement et l'exécution du reste du bootloader
- U-Boot à proprement parler

Das U-Boot

Aperçu des commandes utilisables

- Manipulation mémoire : `md` (*memory display*), `mm` (*memory modify*)
- Manipulation mémoire Flash : `erase` (effacement), `cp` (copie), `protect` (protection), `mtdparts` (manipulation de la table des partitions)
- Exécution : `source` (exécution de scripts), `bootm` (démarrage d'une image depuis la mémoire), `go` (démarrage d'une application *standalone*)
- Réseau : `bootp` (démarrage d'une image via le réseau avec BOOTP/TFTP), `dhcp` (démarrage d'une image via le réseau avec DHCP/TFTP), `loadb` (récupération d'une image via une liaison série)

Das U-Boot

Aperçu des commandes utilisables

- Environnement : `printenv` (affiche les variables d'environnement), `saveenv` (sauvegarde les variables), `setenv` (modifie une variable), `run` (exécute le contenu d'une variable), `bootd` (exécute le contenu de la variable `bootcmd`)
- Flattened Device Tree : `fdt list` (affiche un niveau), `fdt mknode` (crée un nœud), `fdt set` (définie les propriétés d'un nœud), `fdt rm` (supprime un nœud ou une propriété)

Das U-Boot

Exemple : démarrage du noyau Linux

- Exemple typique dans un système embarqué
  - Chargement en mémoire du noyau depuis une flash interne ou depuis un fichier sur une partition d'une carte SD
  - Chargement en mémoire du système de fichiers mémoire initial
  - Chargement en mémoire de l'arbre des périphériques
  - Démarrage du noyau avec les bonnes informations
  - Possibilité d'interrompre la séquence de démarrage par une action de l'utilisateur (saisie d'un caractère sur une liaison série par exemple) pour avoir accès aux autres fonctionnalités d'U-Boot

Plan

Démarrage de Linux

Souce : https://www.kernel.org/doc/Documentation/arm/Booting

- Vu du noyau, voici ce qui doit être fait par le bootloader
  - Initialisation de toute la RAM de la plate-forme
  - Initialisation d'un port série (recommandé)
  - Détection du type de machine (obligatoire sauf support des arbres de périphérique) pour le passer au noyau
    - Utilisé pour sélectionner le code spécifique pour l'initialisation de la plate-forme dans le noyau
    - Liste : https://www.arm.linux.org.uk/developer/machines/
  - Préparer soit la structure ARM Tags (ATAG) soit l'arbre des périphériques pour la passer au noyau
  - Charger l'image disque mémoire initiale (optionnel)
  - Appeler l'image du noyau

Démarrage de Linux
Structure ARM Tags (ATAG)

- Ancienne méthode pour passer des informations entre le bootloader et le noyau
- Liste de triplets (tag, taille, contenu) placée en mémoire
  - `ATAG_CORE` : Début de la liste (obligatoire)
  - `ATAG_NONE` : Fin de la liste (obligatoire)
  - `ATAG_MEM` : Description de la mémoire physique (obligatoire)
  - `ATAG_INITRD2` : Localisation de l'image disque initiale
  - `ATAG_CMDLINE` : Ligne de commande du noyau
- Il est recommandé de placer cette structure dans les 16 premiers ko de mémoire
- Le pointeur vers le début de la liste est placé dans le registre `r2`

Démarrage de Linux
Arbres des périphériques

- L'arbre des périphériques est une structure de données permettant de décrire les composants matériels d'un système
- Initialement conçu pour les machines de travail et les serveurs à base d'architecture SPARC
- Proposé par le projet *OpenFirmware*
- Est maintenant la structure obligatoire pour décrire une plate-forme à base d'ARM dans le noyau Linux (à la place de l'identifiant de machine et de la structure ATAG)

Démarrage de Linux
Arbres des périphériques

- Deux représentations : une sous forme lisible par un humain (*Device Tree Source, DTS*) et une facilement interprétable par une machine (*Device Tree Blob, DTB* ou *Flattened Device Tree*)
- En plus de la description du matériel, l'arbre peut contenir également la ligne de commande du noyau
- La syntaxe des arbres sera abordée dans la suite du cours
- Il est recommandé de placer l'arbre juste au dessus des 128 premiers Mo de RAM
- Le pointeur vers le début de la liste est placé dans le registre `r2`

Démarrage de Linux
Appel du noyau

- Il est recommandé de placer l'image du noyau dans les 128 premiers Mo de RAM, mais au dessus des 32 premiers Mo (pour ne pas avoir à la déplacer avant se décompression)
- `r0` contient `0`
- `r1` contient l'identifiant du type de la plate-forme
- `r2` contient un pointeur vers la structure ARM Tags (ATAG) ou l'arbre des périphérique (le noyau identifie si l'adresse correspond à un arbre grâce à la valeur magique `0xd00dfeed`)

Démarrage de Linux
Appel du noyau

- Pause de tous les périphériques faisant du DMA
- Toutes interruptions désactivées
- Mode HYP (si disponible) ou SVC
- MMU désactivée
- Caches de données désactivés
- Saut direct à la première instruction de l'image noyau, en mode ARM (sauf sur les processeurs qui ne supportent que le mode *Thumb*)

Plan

Décompression

- L'image du noyau peut avoir été générée sous deux formes : compressée (`zImage` le plus souvent, éventuellement avec un entête pour U-Boot : `uImage`) ou non-compressée
- Dans le cas d'une image compressée, l'exécution démarre dans `arch/arm/boot/compressed/head.S` au niveau du symbole `start`
- Le code a été compilé pour pouvoir s'exécuter quelque soit l'adresse à laquelle il a été positionné (PIC : *Position Independent Code*)

Décompression

- Le code va chercher l'adresse physique du début de la RAM. Par défaut, il va prendre l'adresse contenue dans `pc` (adresse physique de l'instruction en cours d'exécution) et faire un ET avec la constante `0xf8000000` (le noyau fait l'hypothèse qu'il a été chargé dans les 128 premiers Mo de la RAM)
- Il y ajoute `TEXT_OFFSET` (traditionnellement `0x8000`, soit 32 ko) pour obtenir l'adresse physique finale de la section `.text` du noyau après décompression
  - Ce décalage permet de sauter la table des vecteurs d'interruption, l'éventuelle structure ATAG, et de laisser de la place pour la table des pages initiale du noyau (16 ko en dessous de l'adresse de démarrage du noyau)

Décompression

- Si possible, il crée ensuite une table des pages minimaliste afin d'activer les caches I et D pour la zone contenant le noyau compressé et le futur noyau décompressé (pas de traduction d'adresse pour le moment)
- Il met ensuite en place une pile et une zone d'allocation mémoire juste après le noyau afin d'avoir un support minimal pour appeler du code écrit en C

Décompression

- Il vérifie ensuite si un arbre des périphérique (DTB) est concaténé avec l'image compressée (c'est autorisé) en cherchant le nombre magique `0xD00DFEED`
  - Si la structure ATAG est passée via `r2`, l'arbre est mis à jour avec les informations en provenance de la structure ATAG
  - Si un arbre des périphérique est passé via `r2`, il est ignoré et `r2` est modifié pour pointer vers le DTB concaténé
  - La taille de l'image compressée est mise à jour pour prendre en compte le DTB
  - La pile et la zone pour l'allocation mémoire sont décalés pour ne pas écraser le DTB

Décompression

- Il vérifie ensuite si, une fois décompressé, le noyau risque d'écraser l'image compressée
  - Si oui, le code va copier l'image compressée (et éventuellement le DTB concaténé) après l'adresse calculée de fin de l'image une fois décompressée
  - Une fois déplacée, il saute à la nouvelle adresse du symbole `restart` (mise en place de la pile et de la zone d'allocation)

Décompression

- Il calcule ensuite le déplacement entre l'adresse d'exécution et l'adresse à laquelle le code de décompression a été lié
  - Il met ensuite à jour les entrées grâce à la *Global Offset Table* (GOT)
- Il initialise à zéro la zone BSS
- Il saute au symbole `decompress_kernel` qui est une fonction C dans `arch/arm/boot/compressed/misc.c` et qui va gérer la décompression

Après la décompression

- Une fois l'image décompressée, le code désactive et vide les caches
- Le code saute au symbole `__enter_kernel` qui remet en place les registres `r0` (0), `r1` (machine ID) et `r2` (ATAG/DTB)
- Puis, on saute enfin vers le début de l'image décompressée (début de la RAM + `TEXT_OFFSET`), ce qui correspond au symbole `stext` dans `arch/arm/kernel/head.S`

Conclusion sur la phase de décompression

- Outre l'intérêt culturel (ça ne fait pas de mal de regarder un peu d'assembleur de temps en temps), la connaissance du fonctionnement de la décompression a plusieurs intérêts pratiques
- La plupart des noyaux sont compressés, et donc la décompression est une étape essentielle
- Un mauvais positionnement du noyau par le bootloader en mémoire peut entraîner une recopie avant la décompression, recopie coûteuse en temps
- Cela explique aussi les contraintes sur le positionnement de la structure ATAG ou du DTB mentionnées précédemment

Sources

- `arch/arm/boot/compressed/head.S`
- Linus Walleij, *How the ARM32 Linux kernel decompresses*, août 2020, `https://people.kernel.org/linusw/how-the-arm32-linux-kernel-decompresses`

Plan

Démarrage du noyau

- Le noyau commence son exécution au niveau du symbole `stext` situé dans `arch/arm/kernel/head.S`
- La première tâche réalisée par le noyau va être de mettre en place les tables de traduction d'adresse et d'activer la MMU
- Cette opération est un peu complexe

Organisation de l'espace d'adressage

Architecture ARM 32 bits

- L'espace d'adressage *virtuel* est divisé en deux zones
  - `0x0000_0000` – `0xBFFF_FFFF` (3 Go) : mémoire du processus en cours d'exécution (*userspace*)
  - `0xC000_0000` – `0xFFFF_FFFF` (1 Go) : mémoire du noyau (code et structures de données) (*kernelspace*)
- Le symbole `PAGE_OFFSET` contient l'adresse de début de la zone consacrée au noyau (`0xC000_0000`)
- Il est possible de changer cette répartition (mais le cas indiqué ci-dessus est de loin le plus fréquent)

Adresse courante d'exécution

- Le code du noyau n'est pas *position-independent*
- Il a été généré par l'éditeur de liens pour être placé à une certaine adresse virtuelle (dans la zone `0xC000_0000 - 0xFFFF_FFFF`)
- Or, au démarrage, l'image du noyau peut avoir été placé à peu près n'importe où en mémoire physique (et très probablement pas à l'adresse prévue par l'éditeur de liens)
- Le code va donc chercher à calculer le déplacement entre l'adresse à laquelle le code s'exécute réellement et celle à laquelle il a été généré pour s'exécuter

Adresse courante d'exécution

```
        adr     r3, 2f
        ldmia   r3, {r4, r8}
        sub     r4, r3, r4      @ (PHYS_OFFSET - PAGE_OFFSET)
        add     r8, r8, r4      @ PHYS_OFFSET

2:      .long   .
        .long   PAGE_OFFSET
```

Aperçu de la suite de l'initialisation bas-niveau

- Mise en place d'une table des pages initiales (couvrant uniquement le code et les données du noyau, placés à l'adresse virtuelle `0xC000_0000`)
- Activation de la MMU et saut vers l'espace d'adressage virtuel
- Mise en place de l'environnement d'exécution de la tâche `init` (PID 0) : pile, structures de données représentant la tâche
- Initialisation du CPU sur lequel le code s'exécute
- Initialisation de l'architecture
  - Code compilé en dur
  - Identifiant de machine + ATAG
  - Arbre des périphériques

Aperçu de la suite de l'initialisation bas-niveau

- Identification des régions mémoires disponibles
- Suite de la mise en place de la mémoire virtuelle (zone de mapping direct)
- Première analyse des arguments du noyau
- Mise en place du mapping mémoire final
- Mise en place de la table des vecteurs d'interruption
- Initialisation des structures nécessaires pour l'allocation mémoire
- Le reste de l'initialisation est globalement commun à toutes les architectures

Sources

- `arch/arm/kernel/head.S`
- Linus Walleij, *How the ARM32 kernel starts*, août 2020, `https://people.kernel.org/linusw/how-the-arm32-kernel-starts`
- Linus Walleij, *Setting Up the ARM32 Architecture, part 1*, octobre 2020, `https://people.kernel.org/linusw/setting-up-the-arm32-architecture-part-1`
- Linus Walleij, *Setting Up the ARM32 Architecture, part 2*, octobre 2020, `https://people.kernel.org/linusw/setting-up-the-arm32-architecture-part-2`

Suite du démarrage du noyau

- Initialisation des périphériques compilés en dur dans le noyau
- Mise en place des différents mécanismes du noyau (notamment la gestion du temps, l'ordonnanceur...)
- Un fois que tout est initialisé, il reste maintenant à passer la main à l'espace utilisateur pour le démarrage des applications

Plan

Système de fichiers

- Pour pouvoir démarrer les applications, il faut pouvoir lire les exécutables depuis un système de fichiers
- Le noyau doit donc localiser le système de fichiers racine (`/`) et le monter
- Pour se faire, le noyau doit disposer
  - De la localisation de ce système de fichiers racine (argument `root` passé au noyau, exemples : `/dev/mmcblk0p1` (1re partition de la 1re carte SD), `/dev/mtdblock0` (mémoire flash), `/dev/sda1` (1re partition du premier disque SCSI/SATA)...)
  - Du/des pilote(s) de périphérique nécessaire(s) pour accéder au système de fichiers
  - Du pilote pour le système de fichiers lui-même (`ext2/3/4`, `fat32`...)

Pilotes pour accéder au système de fichiers racine

- Si tous les pilotes nécessaires ont été compilés en dur dans le noyau, pas de problème
- Cependant, dans le cas d'une distribution devant gérer de nombreuses configurations différentes, compiler tous les pilotes possibles en dur, imposerait d'avoir un noyau très volumineux
- Et donc, le plus souvent, dans ce cas, les pilotes sont généralement compilés sous forme de modules (afin de pouvoir charger uniquement ceux utiles dans la configuration donnée)
- Or les modules sont des fichiers et pour pouvoir les charger, il faut que le système de fichiers sur lequ ils sont soit lui-même monté...

Utilité d'un système de fichiers initial

- Donc, dans de nombreuses distributions classiques (c'est moins vrai dans le monde de l'embarqué puisque le noyau est souvent compilé ad-hoc pour le système ciblé), les pilotes de périphérique nécessaires pour accéder au système de fichiers racine ne sont pas compilés en dur
- Donc le noyau ne peut pas monter ce système de fichiers racine...
- La solution passe par une image de système de fichiers racine initial (`initrd`/`initramfs`)
- L'image est chargé en mémoire par le bootloader et son adresse est passée au noyau

Contenu du système de fichiers initial

- Le noyau monte cette image comme système de fichiers racine (`/`)
- Cette image contient
  - L'exécutable qui sera utilisé comme processus initial (`/init` dans le cas d'un `initramfs`) : obligatoire (sinon le noyau cherchera le système de fichiers racine ailleurs)
  - Ses dépendances (éventuelles bibliothèques partagées, éditeur de liens dynamique et chargeur : `/lib/ld-linux.so`, fichiers de configuration...)
  - Un squelette d'arborescence racine classique (`/dev`, `/tmp`, `/var`...)
  - Un mécanisme pour la gestion des fichiers spéciaux dans `/dev`
  - Les modules nécessaires pour le chargement du système de fichiers racine

Contenu du système de fichiers initial

- Éventuellement, cette image contient tous les exécutables et fichiers nécessaires au fonctionnement du système, auquel cas, un autre système de fichiers racine n'est pas nécessaire

Mécanismes

- Deux mécanismes : `initrd` (ancien) et `initramfs` (introduit dans la série 2.6)
- `initrd` : image (éventuellement compressée) d'un système de fichiers (ex. `ext2/3`, `cramfs`...)
- `initramfs` : archive `cpio` (éventuellement compressée) qui est décompressée par la noyau dans un `rootfs` (système de fichier mémoire)

initrd
Fonctionnement (suite)

- Le bootloader place l'image du noyau et celle de l'image `initrd` en mémoire
- Le noyau crée un disque mémoire `/dev/ram0` et y décompresse l'image `initrd`, puis libère la mémoire occupée par celle-ci
- Si le périphérique racine (paramètre `root`) n'est pas `/dev/ram0`, utilisation de l'ancienne méthode dite *change_root* (voir plus loin)
- Le système de fichiers `/dev/ram0` est monté à la racine `/`
- Le fichier `/sbin/init` est exécuté avec l'identifiant utilisateur 0 (super-utilisateur)

initrd
Fonctionnement (suite)

- Traditionnellement, cet exécutable effectue les opérations suivantes :
  - Montage du système de fichiers racine final
  - Échange des systèmes racines pour que le système de fichiers racine final soit monté à la racine `/`, grâce à l'appel système `pivot_root`
  - Appel de l'exécutable `/sbin/init` du système de fichiers racine final
  - Suppression de l'image `initrd` de la mémoire

initrd

Fonctionnement : Ancien mécanisme change_root

- Si le périphérique racine n'est pas `/dev/ram0` :
- Le système de fichiers `/dev/ram0` est monté à la racine `/`
- Le fichier `/linuxrc` est exécuté s'il existe
- Lorsqu'il se termine (ou s'il n'existe pas), le noyau monte le système de fichiers racine final à la place de `/dev/ram0` et remonte ce dernier dans le répertoire `initrd` (s'il existe)
- Le fichier `/sbin/init` est exécuté

- Le mécanisme `initrd` est tombé en désuétude au profit d'`initramfs`
- Plus d'informations :
  - Page de manuel `initrd(4)`
  - Fichier source `init/do_mounts_initrd.c` du noyau
  - Fichier documentation `Documentation/admin-guide/initrd.rst`

initramfs
Fonctionnement

- Lors de la compilation du noyau, une image `initramfs` compressée est générée et intégrée avec l'image du noyau
  - Cette image par défaut est vide mais l'option de configuration `CONFIG_INITRAMFS_SOURCE` peut indiquer au processus de compilation du noyau au choix : une image `cpio` existante à utiliser, un répertoire contenant les fichiers à intégrer à l'image ou un fichier de configuration indiquant comment générer cette image
  - Si une image `initramfs` est passée au noyau au démarrage, son contenu vient remplacer le contenu correspondant dans l'image intégrée au noyau

initramfs
Fonctionnement (suite)

- Le noyau met en place un système de fichiers `rootfs`, système basé sur `ramfs` ou `tmpfs` et ne pouvant pas être démonté
- Il décompresse l'archive liée avec lui dans ce système de fichiers
- Si une autre image `initramfs` a été passée par le bootloader, le noyau la décompresse également dans le `rootfs` (écrasant ainsi éventuellement les fichiers contenus dans l'image liée au noyau)
- Le noyau appelle le fichier `/init` (avec l'identifiant utilisateur 0). Cet exécutable n'est pas censé retourner

initramfs

Fonctionnement (suite)

- Cet exécutable `/init` effectue traditionnellement les actions suivantes :
  - Prépare le montage du système de fichiers racine final
  - Efface le contenu du `rootfs`
  - Monte le système de fichiers racine final
  - Puis remplace la racine par ce système de fichiers (avec `mount -move`)
  - Lance l'exécution de `/sbin/init` sur le nouveau système racine (ces dernières opérations sont complexes, le programme `utils/run_init.c` de la `klibc` les effectue correctement)
- S'il n'existe pas, le noyau localise le système de fichiers racine (argument `root`), le monte et exécute `/sbin/init` (comportement normal sans `initrd/initramfs`)

`initramfs`
Fonctionnement (suite)

- L'image `initramfs` peut également contenir l'intégralité du système de fichiers, notamment dans le cas des petits systèmes embarqués
- Dans ce cas, le fichier `/init` peut simplement se contenter de démarrer les services et applications nécessaires et ne pas monter d'autres systèmes de fichiers
- Nous verrons en TP comment générer une telle image à la main à l'aide de *BusyBox*

initramfs
Conclusion

- Avantages par rapport à `initrd`
  - Image intégrée au noyau (toujours à part pour `initrd`)
  - Image en général plus petite (archive vs. image d'un système de fichiers)
  - Code de décompression très concis dans le noyau et ne requiert pas de pilote de système de fichiers (hors `rootfs`)
- Plus d'informations :
  - `Documentation/filesystems/ramfs-rootfs-initramfs.rs`

Plan

`init`
Présentation

- `init` est le premier processus lancé par le noyau
- Il s'exécute avec l'identifiant de processus (PID) `1`
- Il est lancé (voir fonction `kernel_init` dans `init/main.c`) depuis le fichier :
  - Indiqué par l'argument `init` passé au noyau
  - `/sbin/init`
  - `/etc/init`
  - `/bin/init`
  - `/bin/sh`
- Le noyau panique s'il n'est pas capable de le trouver (`Kernel panic - not syncing: No working init found.`)

`init`
Présentation

- Son entrée standard (descripteur de fichier `0`), sa sortie standard (`1`) et sa sortie d'erreur (`2`) sont `/dev/console`
- C'est un démon dont la durée de vie est celle du système (il ne doit pas s'arrêter avant, sinon le noyau panique : `Kernel panic - not syncing: Attempted to kill init!`)

`init`
Présentation

- Il est l'ancêtre (direct ou indirect) de tous les processus s'exécutant sur le système (ils sont tous lancés par un `fork` depuis `init` ou un de ses descendants)
- Tous les processus zombies (terminés et en attente d'acquittement par leurs pères) dont les pères se terminent sans avoir acquitté la fin de leurs fils (par un appel à `wait`), sont rattachés à `init` qui acquitte leur terminaison
- Il est responsable d'initialiser le système et, le moment venu, d'arrêter les différents services lors de l'extinction

`init`
Présentation

- Dans les systèmes Linux (embarqués ou non), on trouve principalement trois variantes (d'autres existent mais sont plus rares)
  - *BusyBox*
  - *SysV-style init*
  - systemd, utilisé par la majorité des distributions depuis quelques années

BusyBox init

- *BusyBox* fournit une implémentation simple d'`init`
- Au démarrage, par défaut, il va exécuter le fichier `/etc/init.d/rcS`
- Ce fichier (le plus souvent un script shell) va se charger de toute l'initialisation
  - Montage des systèmes de fichiers (le plus souvent par un `mount -a` à l'aide du fichier `/etc/fstab`)
  - Configuration des interfaces réseaux
  - Démarrage des services...
- Au redémarrage/arrêt du système, `init` va, par défaut, démonter proprement les montages (`unmount -a -r`) et désactiver le swap (`swapoff -a`)

SysV-style `init`
Niveaux d'exécution

- Un niveau d'exécution (*runlevel*) indique dans quel état se situe la machine
- La configuration d'`init` lui indique quels services ils doit démarrer/arrêter lors du passage d'un niveau d'exécution à un autre
- Il en existe 7 numérotés de `0` à `6` (un huitième, `s` ou `S` est parfois un alias pour l'un d'entre eux)
- La commande `telinit` permet au super utilisateur de passer d'un niveau d'exécution à un autre

SysV-style `init`

Niveaux d'exécution classiques

- `0` : Arrêt du système (standard)
- `1` : Mode *single-user* (standard)
- `3` : Mode multi-utilisateurs normal (*Linux Standard Base*)
- `5` : Mode multi-utilisateurs normal + interface graphique (*Linux Standard Base*)
- `6` : Redémarrage du système (standard)

SysV-style `init`

`/etc/inttab`

- `init` se base sur le fichier de configuration `/etc/inittab`
- Dans ce fichier, le format des lignes est le suivant : `id:runlevels:action:process`
  - `id` : identifiant unique de la ligne
  - `runlevels` : liste des niveaux d'exécution pour lesquels l'action doit être faite
  - `action` : action à effectuer (voir liste)
  - `process` : processus à exécution (le cas écheant)

SysV-style `init`

`/etc/inttab`, actions

- `initdefault` : indique le niveau d'exécution par défaut au démarrage (le champ `process` est ignoré)
- `sysinit`, `boot` et `bootwait` : processus exécuté au démarrage du système (le champ `runlevels` est ignoré)
- `wait` : démarre le processus et attend qu'il se termine
- `respawn` : démarre le processus (et le redémarre dès qu'il se termine)
- `ctrlaltdel` : processus exécuté lorsque `init` reçoit le signal `SIGINT`, typiquement lorsque l'utilisateur fait `ctrl+alt+suppr` dans une console

SysV-style `init`

`/etc/inttab`, exemple

```
# Start runlevel
id:2:initdefault:

# System initialization
si::sysinit:/etc/init.d/rcS

# Single user mode
~:S:wait:/sbin/sulogin

# /etc/init.d executes S et K scripts when the
# runlevel changes
l0:0:wait:/etc/init.d/rc 0
l1:1:wait:/etc/init.d/rc 1
l2:2:wait:/etc/init.d/rc 2
l3:3:wait:/etc/init.d/rc 3
l4:4:wait:/etc/init.d/rc 4
l5:5:wait:/etc/init.d/rc 5
l6:6:wait:/etc/init.d/rc 6
```

SysV-style `init`

`/etc/inttab`, exemple (suite)

```
ca::ctrlaltdel:/sbin/shutdown -t5 -rf now

# Runlevel 2,3: getty on virtual consoles
# Runlevel   3: getty on terminal (ttyS0)
1:23:respawn:/sbin/getty tty1 VC linux
2:23:respawn:/sbin/getty tty2 VC linux
S0:3:respawn:/sbin/getty -L 9600 ttyS0 vt320
```

SysV-style `init`

`/etc/init.d/rc`

- Le fichier `inittab` précédent nous indique qu'à l'entrée dans un niveau d'exécution donné (ex. `3`), le script `/etc/init.d/rc` est exécuté avec l'argument `3` (dans certaines distribution, il se trouve dans `/etc/rc.d/rc`)
- Ce script va aller dans le répertoire `/etc/rc.d/rcx.d` (où `x` est remplacé par l'argument, ici `3`)
- Dans ce répertoire, on trouve des fichiers (le plus souvent les liens symboliques) dont le nom est de la forme `Knnwwwwww` (où `nn` est un nombre et `wwwwww` un nom quelconque) ou `Snnwwwwww`

SysV-style `init`

`/etc/init.d/rc`

- Les fichiers `Knnwwwwww` vont être appelés, dans l'ordre alphabétique, avec l'argument `stop`
- Les fichiers `Snnwwwwww` vont être appelés, dans l'ordre alphabétique, avec l'argument `start`
- Le plus souvent, ces fichiers sont des liens symboliques vers des scripts contenus dans `/etc/rc.d/init.d`
- Ces scripts portent le nom du service qu'ils gèrent et vont réagir à l'argument `start` ou `stop` en démarrant/arrêtant le service concerné

SysV-style `init`
Conclusion

- Le système d'initialisation issue de *System-V* est très simple à comprendre (même si la syntaxe de `inittab` est assez étrange au premier abord)
- Son gros défaut est l'absence de parallélisation lors du démarrage des services (les services sont démarrés les uns à la suite des autres) et l'absence de gestion des dépendances (le service `x` à besoin du service `y`)
- Ces dépendances sont à gérer à la main, en ajustant les numéros des liens dans `/etc/rc.d/rcx.d`

SysV-style `init`
Conclusion

- Plusieurs alternatives ont été développées et `systemd` s'est progressivement imposé dans la majorité des distributions Linux généralistes ces dernières années
- De part sa simplicité, le SysV-style `init` est encore utilisé par exemple dans l'embarqué

Plan

Références I

[1] Intel.
HPS SoC Boot Guide - Cyclone V SoC Development Kit.
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an709.pdf, January 2016.