

# Introduction à Linux embarqué

Mylène Josserand

Développeuse et formatrice Linux embarqué

Cours INSA, Octobre 2019





- ▶ Master pro génie physiologique et informatique, option Imagerie Médicale
- ▶ Ingénierie Kernel Linux et formatrice
- ▶ Différentes entreprises : Navocap, Smile (ex Open Wide), Bootlin
- ▶ Développement de driver Linux, intégration système sous Yocto Project/Open Embedded et Buildroot
- ▶ [josserand.mylene@gmail.com](mailto:josserand.mylene@gmail.com)
- ▶ Mylene sur IRC et MyleneJ sur github



# Sommaire

## Généralités

Concepts

Licence

## Architecture GNU/Linux

## Execution d'un système GNU/Linux

## Introduction aux systèmes embarqués

Systèmes embarqués

Eléments nécessaires

## Toolchain

Compilateur binaire

Compilateur source

## Construire son système embarqué

Bootloader

Kernel

Rootfs

## Mise au point

BSP

User space

## Les build-systèmes

## Introduction à Buildroot



Dessin issu d'une photographie de Samuel Blanc

## Généralités

Mylène Josserand  
*josserand.mylene@gmail.com*

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.



# Naissance de logiciel libre

- ▶ 1983: Richard Stallman - Projet **GNU** et concept de logiciel libre. Développement de gcc, gdb, glibc, etc
- ▶ 1991: Linus Torvalds - Projet noyau (**kernel**) **Linux**. De type Unix
- ▶ 1995: Linux est de plus en plus populaire sur les serveurs
- ▶ 2000: Linux est de plus en plus populaire sur les **systèmes embarqués**
- ▶ 2008: Linux est de plus en plus populaire sur les appareils mobiles
- ▶ 2010: Linux est de plus en plus populaire sur les téléphones



# Logiciel libre ?

- ▶ Un logiciel est considéré libre quand sa license respecte ces **4 libertés**:
  - ▶ Liberté d'**exécuter** le logiciel pour n'importe quel but
  - ▶ Liberté d'**étudier** le logiciel et de le **modifier**
  - ▶ Liberté de **redistribuer** des copies
  - ▶ Liberté de **distribuer** des copies modifiées
- ▶ Le code doit être disponible, le logiciel peut être modifié et distribué aux utilisateurs

**Parfait pour les systèmes embarqués !**



# Historique de Linux

- ▶ Créé par Linus Torvalds en 1991
- ▶ Est un noyau : logiciel gérant les ressources d'une machine
- ▶ N'est pas un système d'exploitation : est lié au projet GNU  
→ distribution GNU/Linux





L'intérêt pour les entreprises est:

- ▶ de maîtriser les sources de leur OS
- ▶ d'économiser le prix des licences
- ▶ de bénéficier du support d'une communauté importante de développeur
- ▶ d'utiliser des composants déjà existants et testés par X personnes

Seule la partie Desktop a du mal face à ses concurrents

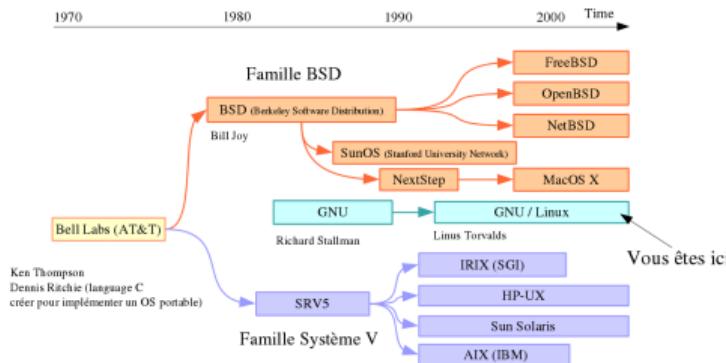


# Concepts



# Basé sur UNIX

- ▶ Implémentation libre d'UNIX diffusé sous licence GPL



- ▶ Les principes d'UNIX sont respectés:
  - ▶ Simplicité, modularité, respect des standards, ouverture
  - ▶ Abstraction du système : noyau = matériel ; userspace = applications
  - ▶ Le noyau permet d'accéder au matériel (pilotes, appels système)
  - ▶ Tout composant est un **fichier**: répertoire, périphérique, élément de communication, etc. (organisation arborescente)
  - ▶ Puissance de la "ligne de commande" (shell et regex)



# Architecture

applications graphiques des utilisateurs  
Navigateur web, office, multimedia...



Applications en ligne de commande  
ls, mkdir, wget, ssh, gcc, busybox...



Librairies partagées

libjpeg, libstdc++, libxml...

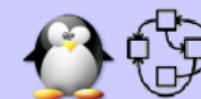
Librairie C

GNU C library, uClibc...



noyau système

Linux, Hurd...



Matériel et périphérique



Espace utilisateur

Espace noyau

Matériel



# Fonctionnalités

- ▶ Noyau monolithique (en un seul fichier) + modules dynamiques
- ▶ Création des processus via fork() et exec()
- ▶ Multi-threading
- ▶ Nombreuses piles réseau (IPv4, IPv6, Ethernet, etc.)
- ▶ Organisation des fichiers arborescente à partir de la racine (/), montage et démontage logique (mount)
- ▶ Multi-utilisateurs avec un super-utilisateur (root) et des groupes



# Licence



## GPL en bref

- ▶ GPL General Public License
- ▶ On la surnomme également copyleft
- ▶ La GPL v2 (1991) est la plus répandue (ex: noyau Linux)
- ▶ La licence s'applique uniquement en cas de redistribution
- ▶ Un code source utilisant du code GPL est du travail dérivé et doit être publié
- ▶ Publication: celui qui reçoit la version binaire peut obtenir le code source
- ▶ Pas de lien (Id) possible entre du code GPL et du code propriétaire



- ▶ La GPL est complexe à gérer dans l'industrie → création de la LGPL
- ▶ Le lien avec du code propriétaire est possible avec la LGPL (Lesser/Library GPL) !
- ▶ En majeure partie, les bibliothèques système sont diffusées sous LGPL (exemple: GNU-libc)
- ▶ Dans le cas d'une application propriétaire il faut donc vérifier qu'aucune bibliothèque liée n'est GPL
- ▶ Le lien dynamique n'affranchit pas de la licence sauf dans des cas très particuliers



- ▶ Dans l'espace noyau (pilotes), SEULE la GPL s'applique (en théorie) !
- ▶ En théorie: On ne peut utiliser les headers du noyau Linux pour créer des binaires non GPL
- ▶ Certaines fonctions ne sont pas disponibles si la licence n'est pas GPL
- ▶ En pratique: tolérance si le pilote n'a pas été créé pour Linux (cas du portage) → nVidia, Broadcom, ...
- ▶ Cependant les pilotes binaires posent des soucis techniques vu qu'un pilote fonctionne pour la version de noyau utilisée pour la compilation



- ▶ Nouvelle version sortie en 2007
- ▶ Oblige à fournir les éléments pour construire un logiciel fonctionnel → réponse à la Tivoisation
- ▶ La GPL v2 demande uniquement la publication des sources à celui qui a reçu le binaire
- ▶ La GPL v3 ne sera pas utilisée pour le noyau Linux
- ▶ Voir: <http://www.gnu.org/licenses/quick-guide-gplv3.fr.html>



# Architecture GNU/Linux

Mylène Josserand  
*josserand.mylene@gmail.com*

Dessin issu d'une photographie de Samuel Blanc

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.



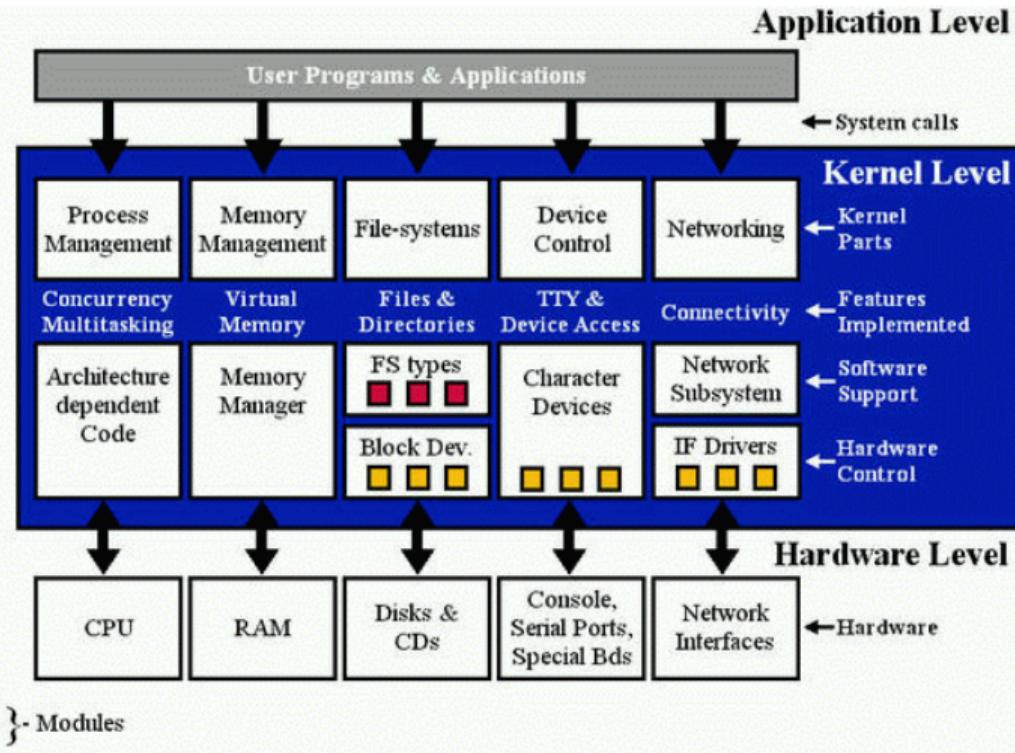
- ▶ Noyau libre semblable à un noyau Unix
- ▶ Le système complet se repose sur les outils GNU:
  - ▶ bibliothèque C, gcc, binutils, fileutils, make, emacs...
  - ▶ Le système complet est donc appelé "**GNU / Linux**"
- ▶ Très tôt partagé comme Logiciel Libre (Licence GPL) → contributeurs et des utilisateurs de plus en plus nombreux
- ▶ Maintenant, fournit en tant que "distribution" (Ubuntu, Fedora, Debian, etc)
- ▶ Système entier composé de:
  - ▶ Un bootloader (Rares sont les cartes capable de booter un noyau Linux)  
Il est trop gros et doit être chargé en RAM. Elle doit être initialisée par un microcode avant → Utilité du **bootloader**
  - ▶ Le noyau Linux qui gère les ressources de la machine
  - ▶ Un système de fichier contenant à minima un programme de démarrage (rootfs)



- ▶ Le noyau Linux est un binaire de type ELF
- ▶ Son image est souvent compressée pour gagner en taille lors du déploiement et de la copie en RAM
- ▶ Il contient un auto extracteur qui va le décomprésser
- ▶ Fournit des services nécessaires à sa fonction:
  - ▶ ordonateur
  - ▶ gestion mémoire, disque, interface réseaux
  - ▶ service abstraits (système de fichier, pile réseaux ...)
- ▶ Une grande partie du noyau peut être déporté en **module** chargé dynamiquement



# Architecture





# Module noyau

- ▶ Modules noyau: .ko (Kernel Object)
- ▶ Les modules binaires sont liés à la version du noyau !
- ▶ Peut se compiler après mais nécessite les entêtes du noyau
- ▶ Mécanisme système pour charger les modules dynamiquement: modprobe, insmod, rmmod



# Organisation d'un rootfs

- ▶ Organisation commune à 90% entre les UNIX
- ▶ Quelques spécificités GNU/Linux et distribution
- ▶ Tout commence depuis la racine /
- ▶ Puis, organisation en dossier/sous-dossier:
  - ▶ /bin,/sbin,/usr/bin,/usr/sbin: binaires communs et systèmes
  - ▶ /lib,/usr/lib: bibliothèques et modules noyau
  - ▶ /etc: fichiers de configuration
  - ▶ /dev: nœuds d'accès aux périphériques
  - ▶ /var: fichiers variables comme log, mail, ...
  - ▶ /opt: pour les programmes externes (ex: LibreOffice)
  - ▶ /home: accueille les répertoires des utilisateurs



# Organisation d'un rootfs

- ▶ Quelques répertoires spéciaux:
  - ▶ /lib/modules: contient les modules du noyau
  - ▶ /root: home-directory de l'utilisateur root
  - ▶ /media: point de montage des volumes amovibles
  - ▶ /proc: système de fichier virtuel (état du système)
  - ▶ /sys: idem pour les périphériques connectés
  - ▶ /boot: noyau statique (vmlinuz, ulmage, ...)



- ▶ Système de fichier virtuel (lecture/écriture) géré par le noyau. (Réponse sur sollicitation, pas d'écriture sur un support)
- ▶ Intérêt: manipuler les variables systèmes comme de fichiers (cat, echo, grep)
- ▶ Exemples:
  - ▶ /proc/version: version du noyau
  - ▶ /proc/cpuinfo: type(s) de processeur(s)
  - ▶ /proc/interrupts: interruptions
  - ▶ /proc/pid: répertoire décrivant le processus associé au pid
  - ▶ /proc/mounts: partitions montées
  - ▶ /proc/modules: liste des modules noyau chargés
- ▶ Nombreuses commandes systèmes basées sur /proc : lsmod, lspci, top, mount, ...



- ▶ Introduit dans le noyau 2.6 (2003) → sysfs
- ▶ Vue synthétique des périphériques connectés
  - ▶ /sys/class
  - ▶ /sys/modules
  - ▶ /sys/bus
- ▶ But: mieux gérer l'ajout/suppression dynamique des périphériques (hotplug)
- ▶ Utilisé par UDEV pour créer dynamiquement les entrées dans /dev
- ▶ Quelques recouvrements avec /proc (bus PCI, USB, ...)



# Execution d'un système GNU/Linux

Mylène Josserand  
*josserand.mylene@gmail.com*

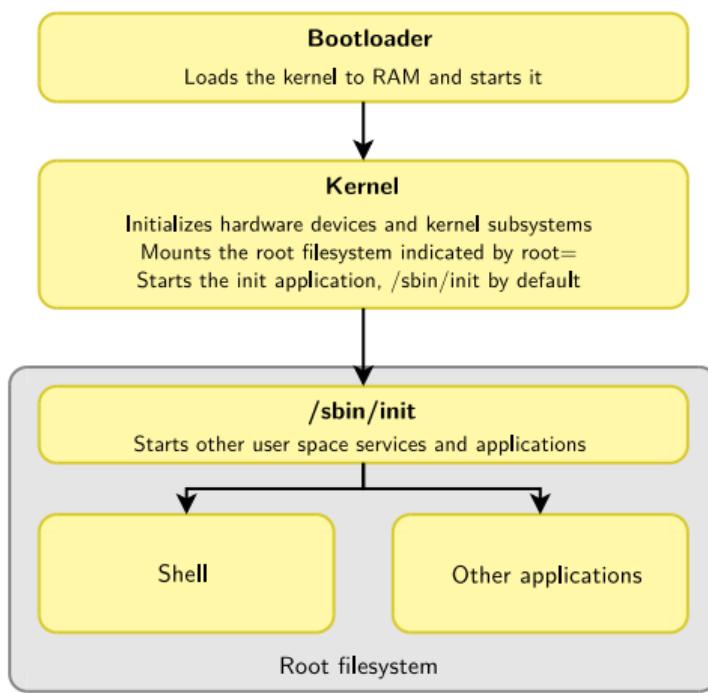
Dessin issu d'une photographie de Samuel Blanc

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.



# Démarrage

1. Le matériel lance le bootloader
2. Le bootloader configure la RAM et le support contenant le noyau
3. Le bootloader copie le noyau en RAM et lui donne la main
4. Le noyau s'auto extrait en RAM
5. Le noyau initialise les périphériques et les sub-systèmes (ordonnanceur, pile réseaux, système de fichier,...)
6. Le noyau récupère dans le rootfs le binaire d'initialisation (`/sbin/init` par défaut) et crée le processus 1
7. Ce processus est chargé de démarrer tous les services en espaces utilisateurs





# Le processus init

- ▶ Le père des processus du système
- ▶ Plusieurs systèmes possibles:
  - ▶ systemV:
    - ▶ facile à configurer car basé sur des scripts shell
    - ▶ architecture vieille et dépassé. Moins performant
  - ▶ systemd:
    - ▶ Parallélise l'initialisation du système. Centralise de nombreux services (logs, etc)
    - ▶ Plus complexe à prendre en main et moins modulaire pour certains
  - ▶ custom:
    - ▶ Linux permet d'utiliser son propre binaire init
    - ▶ Il faut gérer soit même le système



Dessin issu d'une photographie de Samuel Blanc

# Introduction aux systèmes embarqués

Mylène Josserand  
*josserand.mylene@gmail.com*

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.



## Systèmes embarqués



- ▶ Un système embarqué est un système concu pour ne réaliser qu'un certain nombre de tâches définies
- ▶ On associe un système à une fonction (ou un groupe de fonction)
- ▶ On cherche à maîtriser le fonctionnement du système
- ▶ Diffère des systèmes génériques censés savoir tout faire sur n'importe quelle plateforme



# Exemples





# Systèmes embarqués GNU/Linux?

- ▶ Utilisé depuis longtemps comme serveur
- ▶ Pourquoi ne pas l'utiliser pour d'autres systèmes génériques?
- ▶ De nombreuses architectures supportées par Linux (ARM, mips, powerpc ...)
- ▶ Systèmes fiables et performants
- ▶ De nombreuses fonctionnalités déjà codées et validées
- ▶ Nécessite un BSP (Board support package) et un peu d'adaptation



## Eléments nécessaires



# Eléments nécessaires

- ▶ Une système GNU/Linux embarqué nécessite:
  - ▶ Chaîne de compilation croisée (Gcc, as, ld, LibC)
  - ▶ Un Bootloader (U-Boot, barebox, grub, isolinux)
  - ▶ Noyau Linux adapté à l'archi hardware
  - ▶ Outils GNU/LINUX Commandes Linux (sh, ls, cp, etc.)
  - ▶ Applications ...
  - ▶ Outil de génération (Buildroot, OpenEmbedded, ...)



# Toolchain

Mylène Josserand  
*josserand.mylene@gmail.com*

Dessin issu d'une photographie de Samuel Blanc

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.



- ▶ Un point très complexe !
- ▶ Nécessité de construire une chaîne croisée :
  - ▶ GCC
  - ▶ Binutils (as, ld, ...)
  - ▶ Dépendances avec le noyau (system calls, ...) → erreur "Kernel too old"
  - ▶ Choix d'une libC → Glibc, uClibc, Eglibc, ...
  - ▶ GDB
  - ▶ Toute autre bibliothèque utilisée → libstdc++
  - ▶ Dépendances avec le compilateur hôte



- ▶ Interaction entre la libC et le noyau Linux
  - ▶ Appels systèmes (nombre, définition)
  - ▶ Constantes
  - ▶ Structures de données, etc.
- ▶ Compiler la libC – et certaines applications - nécessite les en-têtes du noyau
- ▶ Disponibles dans `<linux/...>` et `<asm/...>` et d'autres répertoires des sources du noyau (include, ...)



## Compilateur binaire



# compilateur binaire

- ▶ Utiliser un compilateur binaire :
  - ▶ ELDK: <http://www.denx.de/wiki/DULG/ELDK>
  - ▶ Code Sourcery :  
<https://www.mentor.com/embedded-software/sourcery-tools-services/>
  - ▶ Linaro Toolchain <https://www.linaro.org/downloads/>
  - ▶ Bootlin's toolchains: <https://toolchains.bootlin.com>
  - ▶ Installation simple
  - ▶ Support (payant) possible
  - ▶ Configuration connue → support par les forums
- ▶ Par contre:
  - ▶ Versions des composants figées
  - ▶ Non utilisation des possibilités du CPU
  - ▶ Choix libC limité



## Compilateur source



# Compiler son compilateur

- ▶ Construire un compilateur:
  - ▶ Crosstool → obsolète
  - ▶ Crosstool-NG → assez complexe à prendre en main
  - ▶ Buildroot / OpenEmbedded
- ▶ Aucun n'est "plug and play"
- ▶ La mise au point peut prendre des jours, voire plus !
- ▶ Binaires produits : arm-linux-\* (gcc, as, ld, ar, nm, ...)



Dessin issu d'une photographie de Samuel Blanc

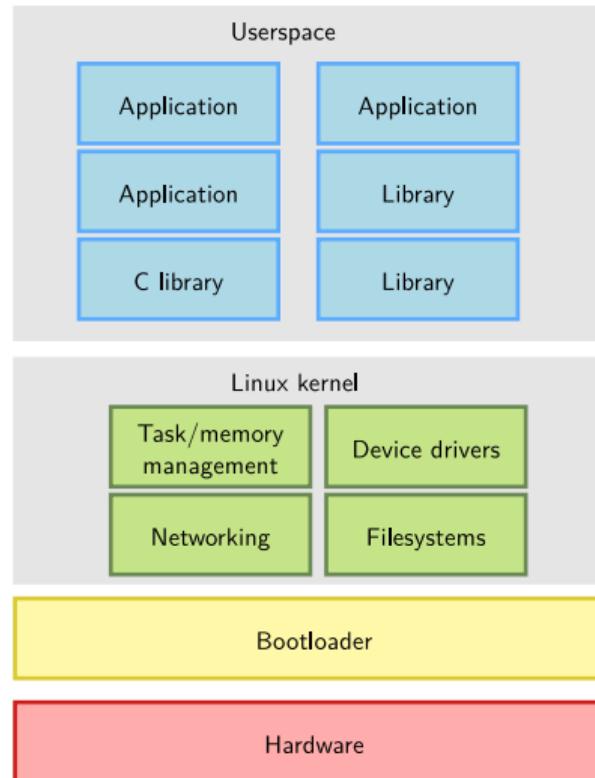
# Construire son système embarqué

Mylène Josserand  
*josserand.mylene@gmail.com*

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.



# Architecture





## Bootloader



# Bootloader

- ▶ Premier logiciel lancé au démarrage de la machine
- ▶ Initialise le matériel (RAM, stockage, ...) nécessaire au boot
- ▶ Charge le noyau en RAM et lui donne la main
- ▶ Permet de donner des arguments au noyau
- ▶ Exemples:
  - ▶ U-boot (supporte de nombreux architectures, la référence)
  - ▶ Barebox (u-boot v2, meilleure archi, moins de fonctionnalités pour l'instant)
  - ▶ grub (x86, serveur et desktop principalement)
  - ▶ isolinux (alternative x86 à grub)
- ▶ Un système embarqué nécessitera une configuration et un support matériel dans le bootloader.



## Kernel



- ▶ Génération d'un noyau:
  1. Récupération du noyau (kernel.org)
  2. Développement du support des drivers manquants
  3. Création du support de la carte (board config 3.10 device tree)
  4. Configuration le noyau (make menuconfig)
  5. Compilation du noyau (make zImage)
  6. Installation sur la cible
- ▶ Binaire utilisable dans arch/arm/boot/zImage ou bien arch/arm/boot/uImage (pour U-Boot)
- ▶ Binaire vmlinuX utile pour le debug
- ▶ Utilisation des variables ARCH et CROSS\_COMPILE.



## Rootfs



# Binaires de base

- ▶ Binaires de base indispensables au système (ls, bash, dd, top, cp, mv, init,...)
- ▶ Trois possibilités:
  - ▶ Récupérer les sources et les compiler un par un →laborieux
  - ▶ GNU/Linux coreutils →trop lourd pour l'embarqué
  - ▶ Busybox : regroupe la majorité des commandes Linux en un seul executable
    - ▶ 95% des distributions Linux embarqués l'utilise
    - ▶ Simple, léger, portable
    - ▶ Diffusé sous licence GPLv2



- ▶ Utilisation des variables ARCH et CROSS\_COMPILE. (comme le noyau)
- ▶ Génération de busybox:
  1. Récupération des sources
  2. Configuration (make menuconfig)
  3. Compilation (make uImage)
  4. Installation sur la cible (make CONFIG\_PREFIX=... install)



# Rootfs - Peuplement

- ▶ Peuplement d'un rootfs:
  1. Installation de la libc et des autres librairies fournies par la toolchain
  2. Installation des binaires de busybox
  3. Création de la configuration de démarrage (/etc/init.d)
  4. Ajout des nodes dans /dev (si pas de mécanisme automatique)
  5. Ajout des librairies et applications du projet.
- ▶ Deux méthodes:
  - ▶ A la main, copier ou installer chaque fichier/logiciel manuellement. (LFS)  
→Inenvisageable à moyen et long terme
  - ▶ Automatiser soit en scriptant soit en utilisant un système existant (OE, yocto, buildroot, uclinix ...) →Plus rapide et plus sûr. Indispensable en production pour maîtriser son livrable.



## Mise au point

Mylène Josserand  
*josserand.mylene@gmail.com*

Dessin issu d'une photographie de Samuel Blanc

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.



## BSP



- ▶ Une plateforme "neuve" nécessite des ajustements pour fonctionner:
  - ▶ Adaptation de la toolchain
  - ▶ Support des drivers dans le **bootloader** et le **kernel**
  - ▶ Ajout des fonctionnalités dans le bootloader et le kernel (réseau, NAND, HDD, ...)
- ▶ Fourni par le constructeur de la plateforme électronique.
- ▶ Si on est le constructeur, il faut développer le support.
- ▶ Outils à disposition (gdb, sonde JTAG, ftrace, oscilloscope, multimetre ...)



## User space



## User space

- ▶ Lorsque le BSP est OK, la mise au point en espace utilisateur peut commencer
- ▶ De nombreux outils permettent de mettre au point des applications:
  - ▶ printf: outil simple à mettre en oeuvre et connu de tout le monde
  - ▶ syslog: utile pour vérifier le bon fonctionnement ou détecter des anomalies
  - ▶ valgrind: vérifie la bonne utilisation de la mémoire
  - ▶ strace/ltrace: affiche les appels systèmes et librairies
  - ▶ gdb: debugguer GNU/LINUX
    - ▶ permet d'accéder à la memoire, controler l'execution, monitoré les threads ...
    - ▶ gdbserver pour les cibles embarqués pour déporter le debug



Dessin issu d'une photographie de Samuel Blanc

## Les build-systèmes

Mylène Josserand  
*josserand.mylene@gmail.com*

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.

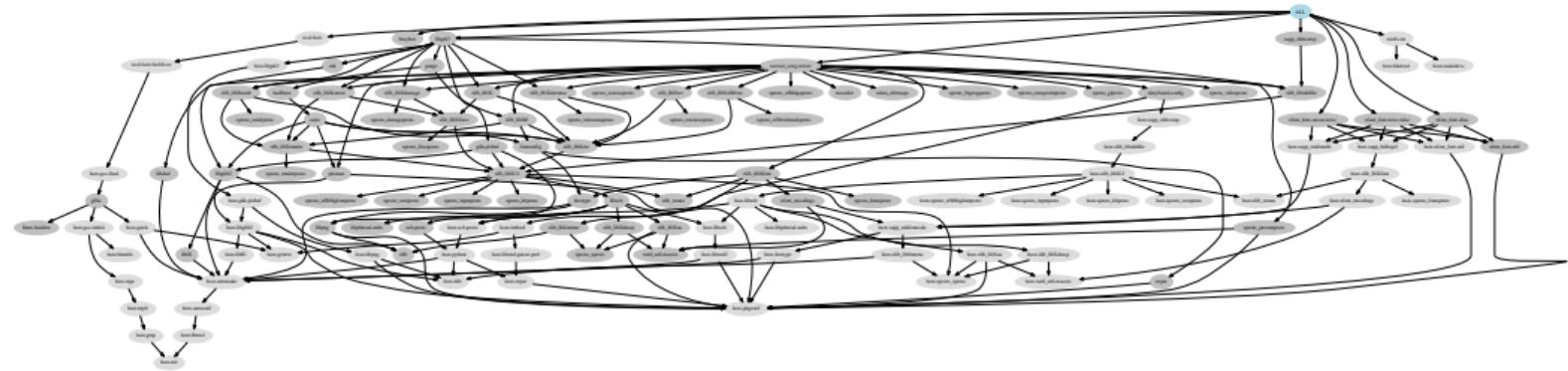


# Les différents travaux en Linux embarqué

- ▶ **BSP:** portage du bootloader et du noyau Linux + développement de drivers.
- ▶ **Integration système:** assembler tous les composants userspace nécessaires pour le système, les configurer, mécanismes de mise à jour, etc
- ▶ **Développement applicatif:** écrire des applications ou bibliothèques spécifiques à l'entreprise



# Complexité de l'intégration userspace



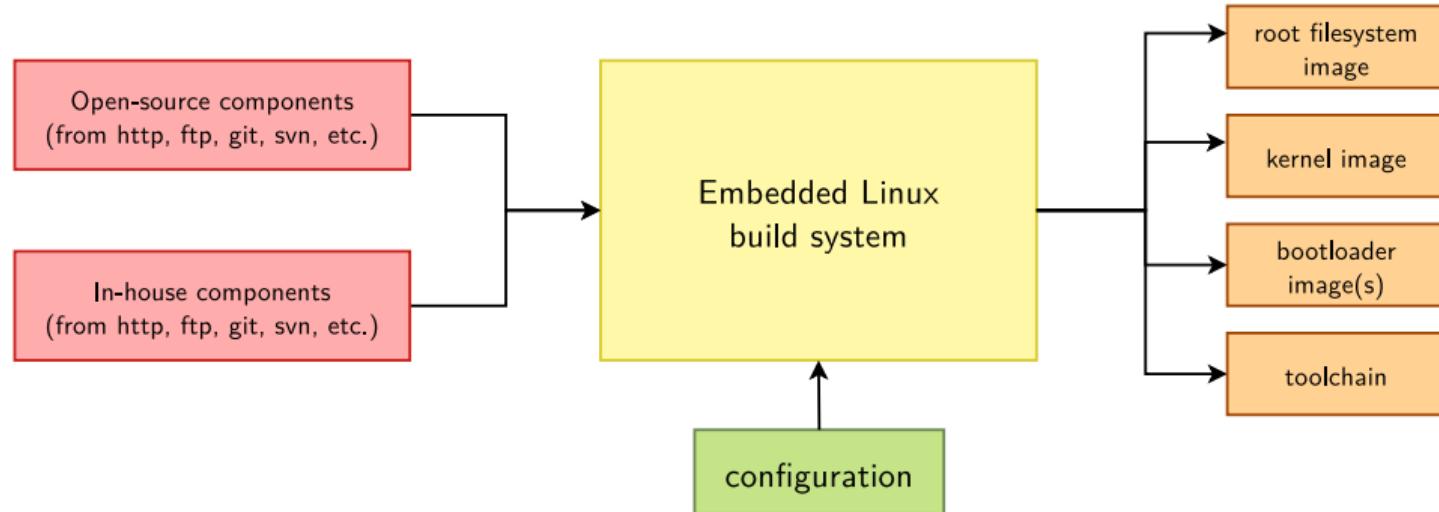


# Intégration système: plusieurs possibilités

	<b>Pros</b>	<b>Cons</b>
<b>Compiler tout manuellement</b>  Debian, Ubuntu, Fedora, etc.	Complètement flexible Gagne en expérience	Enfer des dépendances Doit comprendre tout un tas de détails Compatibilité des versions Manque de reproductibilité
<b>Distributions binaires</b>  Debian, Ubuntu, Fedora, etc.	Facile à créer et étendre	Difficile à customiser Difficile à optimiser (temps de boot, taille) Difficile de compiler depuis les sources Système gros Compilation native Pas de mécanisme bien définis pour générer une image Pleins de dépendances obligatoires Pas disponibles pour toutes les architectures
<b>Systèmes de build</b>  Buildroot, Yocto, PTXdist, etc.	Quasiment complètement flexible Compile depuis les sources : customisation et optimisation faciles Complètement reproductible Cross-compilation Contains des paquets spécifique à l'embarqué	Pas aussi facile qu'une distribution binaire Temps de build



# Build system: principe



- ▶ Compilation depuis les sources → flexibilité
- ▶ Cross-compilation → rapidité à compiler si utilise des machines de build
- ▶ Recettes pour compiler des composants → facile



- ▶ Beaucoup de solutions possibles : Yocto/OpenEmbedded, PTXdist, Buildroot, OpenBricks, OpenWRT, etc.
- ▶ Mais 2 solutions émergent comme les plus populaires :
  - ▶ **Yocto/OpenEmbedded**  
Compile une distribution Linux complète avec des paquets binaires. Très puissant mais complexe et prend beaucoup de temps à apprendre
  - ▶ **Buildroot**  
Compile une image de rootfs, pas de paquets binaires. Beaucoup plus simple à utiliser, comprendre et modifier.



Dessin issu d'une photographie de Samuel Blanc

## Introduction à Buildroot

Mylène Josserand  
*josserand.mylene@gmail.com*

© Copyright 2004-2019, Bootlin.  
Creative Commons BY-SA 3.0 license.



- ▶ Peut compiler une toolchain, un rootfs, un kernel, un bootloader
- ▶ **Facile à configurer:** menuconfig, xconfig, etc.
- ▶ **Rapide:** compile un rootfs en quelques minutes
- ▶ Facile à comprendre: écrit en make, bonne documentation
- ▶ **Petit:** root filesystem, débute en 2 MB
- ▶ **2200+ paquets** pour userspace
- ▶ **Pleins d'architectures** supportées
- ▶ **Technologies bien connues:** *make* et *kconfig*
- ▶ Neutre
- ▶ Communauté active, releases régulières
- ▶ <https://buildroot.org>



## Buts dans le design

- ▶ Buildroot est designé avec des buts précis:
  - ▶ Simple à utiliser
  - ▶ Simple à personnaliser
  - ▶ Builds **reproductibles**
  - ▶ Petit root filesystem
  - ▶ Temps de build relativement rapide
  - ▶ Facile à comprendre
- ▶ Pas toutes les fonctionnalités possibles supportées
- ▶ Plus compliqués et avec beaucoup de fonctionnalités : Yocto Project, OpenEmbedded



# Buildroot dans l'industrie?

- ▶ **Entreprises**

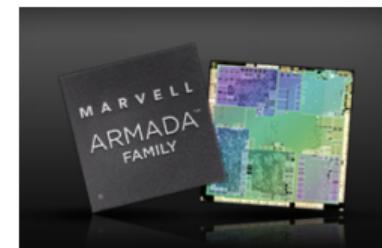
- ▶ Google
- ▶ Barco
- ▶ Rockwell Collins

- ▶ **Vendeurs de processors**

- ▶ Imagination Technologies
- ▶ Marvell
- ▶ Microchip (Atmel)
- ▶ Analog Devices

- ▶ **Vendeur de SoM et de cartes**

- ▶ Beaucoup d'entreprises lors de *R&D* sur des produits
- ▶ Beaucoup de **hobbyists** des cartes de développement : Raspberry Pi, BeagleBone Black, etc.





- ▶ Releases stables tous les 3 mois:
  - ▶ YYYY.02, YYYY.05, YYYY.08, YYYY.11
- ▶ Archives disponibles pour chaque release stable
  - ▶ <https://buildroot.org/downloads/>
- ▶ Plus pratique via git
  - ▶ Visualisation de ses modifications
  - ▶ Simplification des contributions
  - ▶ `git clone git://git.busybox.net/buildroot`
  - ▶ Tags git disponible pour chaque release stable
- ▶ Une **long term support (LTS)** release chaque année
  - ▶ Maintenue durant 1 an
  - ▶ Fixes de sécurité, bugs, compilation, etc
  - ▶ LTS actuelle : 2019.02



# Utilisation de Buildroot

- ▶ Implémenté via `make`
  - ▶ Avec quelques shell scripts d'aide
- ▶ Toutes les interactions via `make` dans le répertoire principal de Buildroot

```
$ cd buildroot/  
$ make help
```

- ▶ Pas besoin d'être `root`, désigné pour être exécuté avec privilèges utilisateurs
  - ▶ Executer en `root` est même fortement découragé!



# Configuration de Buildroot

- ▶ Comme le kernel Linux, utilise *Kconfig*
- ▶ Un choix des interfaces de configuration :
  - ▶ make menuconfig
  - ▶ make nconfig
  - ▶ make xconfig
  - ▶ make gconfig
- ▶ Vérifier que les librairies nécessaires sont installées (*ncurses* pour menuconfig/nconfig, *Qt* pour xconfig, *Gtk* pour gconfig)



# Main menuconfig menu

/home/thomas/projets/buildroot/.config - Buildroot 2017.08 Configuration

**Buildroot 2017.08 Configuration**

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> selects a feature, while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [\*] feature is selected [ ] feature is excluded

- ||| **T**arget options --->
- Build options --->
- Toolchain --->
- System configuration --->
- K**ernel --->
- Target packages --->
- Filesystem images --->
- Bootloaders --->
- Host utilities --->
- Legacy config options --->

**<Select>    < Exit >    < Help >    < Save >    < Load >**



## Lancer une compilation

- ▶ Aussi simple que:

```
$ make
```

- ▶ Si veut garder un log de la compilation pour analyse ou investigation:

```
$ make 2>&1 | tee build.log
```



# Résultats de compilation

- ▶ Localisé dans `output/images`
- ▶ Selon la configuration, le répertoire contiendra:
  - ▶ Une ou plusieurs images de root filesystem dans des formats différents
  - ▶ Une image kernel, un ou plusieurs Device Tree blobs
  - ▶ Une ou plusieurs images de bootloader
- ▶ Pas de standard pour installer les images sur une carte
  - ▶ Trop spécifiques à la carte
  - ▶ Buildroot fournit des outils pour générer une image de carte SD / clef USB (*genimage*) ou de flasher directement sur certaines plateformes : SAM-BA pour Microchip, imx-usb-loader pour i.MX6, OpenOCD, etc.



# Conclusion

- ▶ **Bootloader** : initialisation de la RAM + chargement du kernel [*u-boot ; barebox*]
- ▶ **Kernel** : Gère le matériel, les ressources, protocoles, etc
- ▶ **Userspace** : Rootfs avec LibC, Init, packages (i.e. applications, librairies, etc) x
- ▶ **Build system** : Buildroot ou Yocto
- ▶ Logiciels libres:
  - ▶ Facilite le debug
  - ▶ Facilite le développement
  - ▶ Contrôle total sur le système
  - ▶ Communauté en support
  - ▶ Logiciels testés et corrigés par des millions de personnes

Prochaines sessions : Travaux Pratiques !

# Merci de votre attention !

# Questions ? Commentaires ?

Mylène Josserand — josserand.mylene@gmail.com

Slides under CC-BY-SA 3.0

© Copyright 2004-2019, Bootlin

© Copyright Maxime Chevallier

© Copyright Mylène Josserand

<https://github.com/MyleneJ/cours-insa>