Linux embarqué

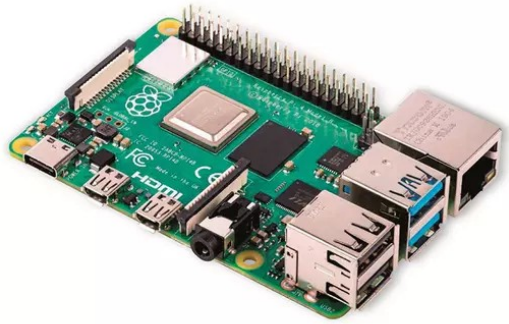
MI11 P22

Guillaume Sanahuja

gsanahuj@hds.utc.fr

Linux embarqué

un cours, deux TP sur la Raspberry Pi 4/ Joy-Pi Note :



- TP 1 :
  - environnement Yocto (noyau, système de fichiers, chaîne de compilation croisée)
  - configuration et compilation du noyau
  - démarrer le tout sur la carte de développement
- TP 2 :
  - programmation des entrées/sorties
  - tâches et charge CPU

Préambule : Système embarqué?

- Qu'est ce qu'un système embarqué ?
  - c'est un système qui assure une **fonction dédiée** pour répondre, la plupart du temps, à une **application spécifique**.
  - on définit souvent un système embarqué par le fait qu'il est soumis à des **contraintes** (énergétiques, encombrement, de ressources, …)
  - *ex : régulateur de vitesse de voiture*
- Qu'est ce qu'un OS embarqué ?
  - c'est un système informatique qui offre des services spécifiques pour l'embarqué

Sommaire

1. Présentation de Linux

1. Présentation de Linux

Bref historique

- Initiateur du projet Linux : le finlandais Linus Benedict Torvalds
  - “Unix clone for PC clones”
  - 5 octobre 1991
    - livraison officielle de la version 0.02
  - Linux 5.17.4 c'est :
    - 128 Mo compressé, 1.3 Go décompressé
    - 74 500 fichiers (principalement C et assembleur) stockés dans 4 900 sous répertoires
    - 22 000 000 de lignes de code

1. Présentation de Linux

Linux ou GNU/linux?

- Projet GNU
  - GNU is not Unix
  - initié par Richard Stallman en 1983
  - fournit la plupart des programmes en mode utilisateur de Linux
    - compilateur/débogueur
    - bash, emacs, grep, GNOME...
- Projet Linux
  - ne fournit “que” le noyau

La licence GPL

- La licence GPL (General Public License)
  - créée par la “free software foundation” (Stallman, 1989)
  - quiconque peut modifier et redistribuer un programme GPL sans verser de royalties
  - les sources d'un programme GPL doivent être accessibles à l'utilisateur du logiciel
  - tout programme dérivé d'un programme GPL doit être distribué sous licence GPL (”contamination”)
- Linux est distribué sous licence GPL

La licence GPL

- La licence GPL s'applique au noyau Linux
  - les pilotes de Linux doivent être open source
  - contournement
    - utilisation de modules dynamiques
- Les programmes utilisateurs ne sont pas concernés par la licence du noyau
  - il existe des dizaines de licences propriétaires ou “open source” (OSI) : LGPL, BSD, Cecill-X, MIT, ...

1. Présentation de Linux

Open Source

- L'OSI (”Open Source Initiative”) donne une définition claire de ce qu'est l'open source
- Maîtrise de son code
  - garantie de pouvoir faire évoluer son produit
  - garantie de pouvoir chercher et corriger les bugs
- Remarque
  - open source ne veut pas dire gratuit ou du domaine public

Distributions Linux

- Pour obtenir un système d'exploitation GNU/Linux complet il faut :
  - le noyau Linux
  - les commandes et programmes du projet GNU
  - tout autre composant logiciel utile
- Distribution
  - rassemble tous les composants nécessaires et les distribue en un seul ensemble cohérent
  - ajoute la gestion des paquetages (apt, rpm, ipk, ...)

Distributions Linux

- Distributions “grand public”
  - Debian, Ubuntu, Mint
  - Redhat, Fedora, Mandriva
  - Suse, Gentoo, ArchLinux
- Distributions pour l'embarqué
  - Libres: Open Embedded, Buildroot, ELBE, Alpine Linux, ...
  - Commerciales: MontaVista, WindRiver, Timesys, ...

2. Architecture de Linux

2. Architecture de Linux

Arborescence du système

- /bin programmes (binaries)
- /sbin programmes systèmes (system binaries)
- /dev fichiers “spéciaux” d'accès aux périphériques
- /etc fichiers de configuration du système
- /home répertoires des utilisateurs
- /lib librairies (“shared object”, .so)
- /usr programmes des utilisateurs
  - /usr/include fichiers d'entête de programmation
- /var fichiers modifiés fréquemment (logs, files d'envoi,...)
- /tmp fichiers temporaires

Les fichiers périphériques

- Spéciaux
  - occupent un inoeud (index node) dans le système de fichiers mais aucuns blocs de données
  - les données lues ou écrites le sont sur le périphérique, pas dans le fichier
- 2 types
  - **bloc**: gère les périphériques qui lisent/écrivent leurs données par blocs (ex: les disques durs manipulent des secteurs de 512 octets)
  - **caractère**: gère les périphériques qui lisent/écrivent leurs données caractère par caractère (ex: les liaisons séries)

Les fichiers périphériques

Contiennent les numéros majeur et mineur du périphérique:

- Le numéro majeur correspond à un type de périphérique
  - chaque numéro majeur est géré par un driver différent
- Le numéro mineur correspond à un périphérique particulier parmi plusieurs de même type
  - les numéros mineurs sont gérés par le même driver

```
$ ls -l /dev
crw--w----   1 root tty         5,     1 avril 19 16:33 console
crw-------   1 root root       89,     0 avril 19 16:33 i2c-0
crw-------   1 root root       89,     1 avril 19 16:33 i2c-1
brw-rw----   1 root disk      259,     0 avril 19 16:33 nvme0n1
crw-rw----   1 root dialout     4,    64 avril 19 16:33 ttyS0
crw-rw----   1 root dialout     4,    65 avril 19 16:33 ttyS1
```

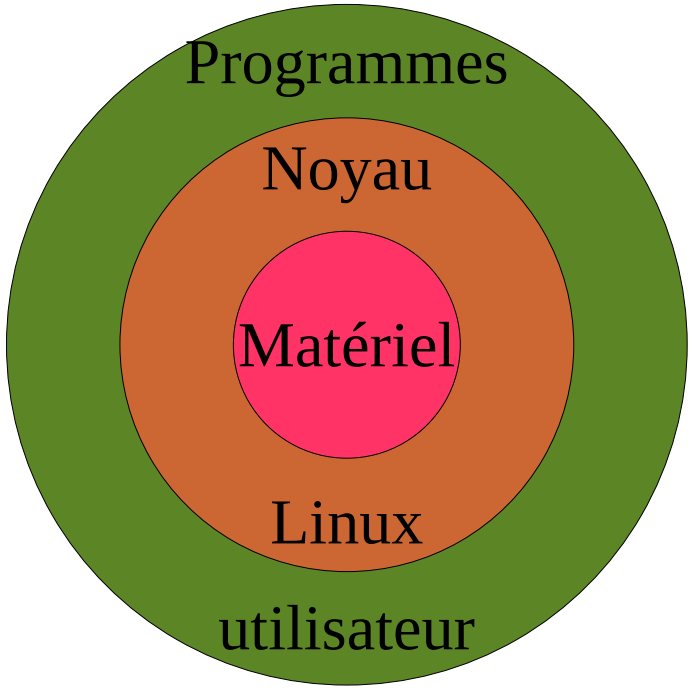
Les fichiers périphériques

Remarques :

- Le nom du fichier périphérique est purement conventionnel
- Les seules information importantes pour le noyau sont les numéros majeurs et mineurs
- Les numéros majeurs et mineurs connus du noyau sont précisés dans le fichier *Documentation/admin-guide/devices.txt*

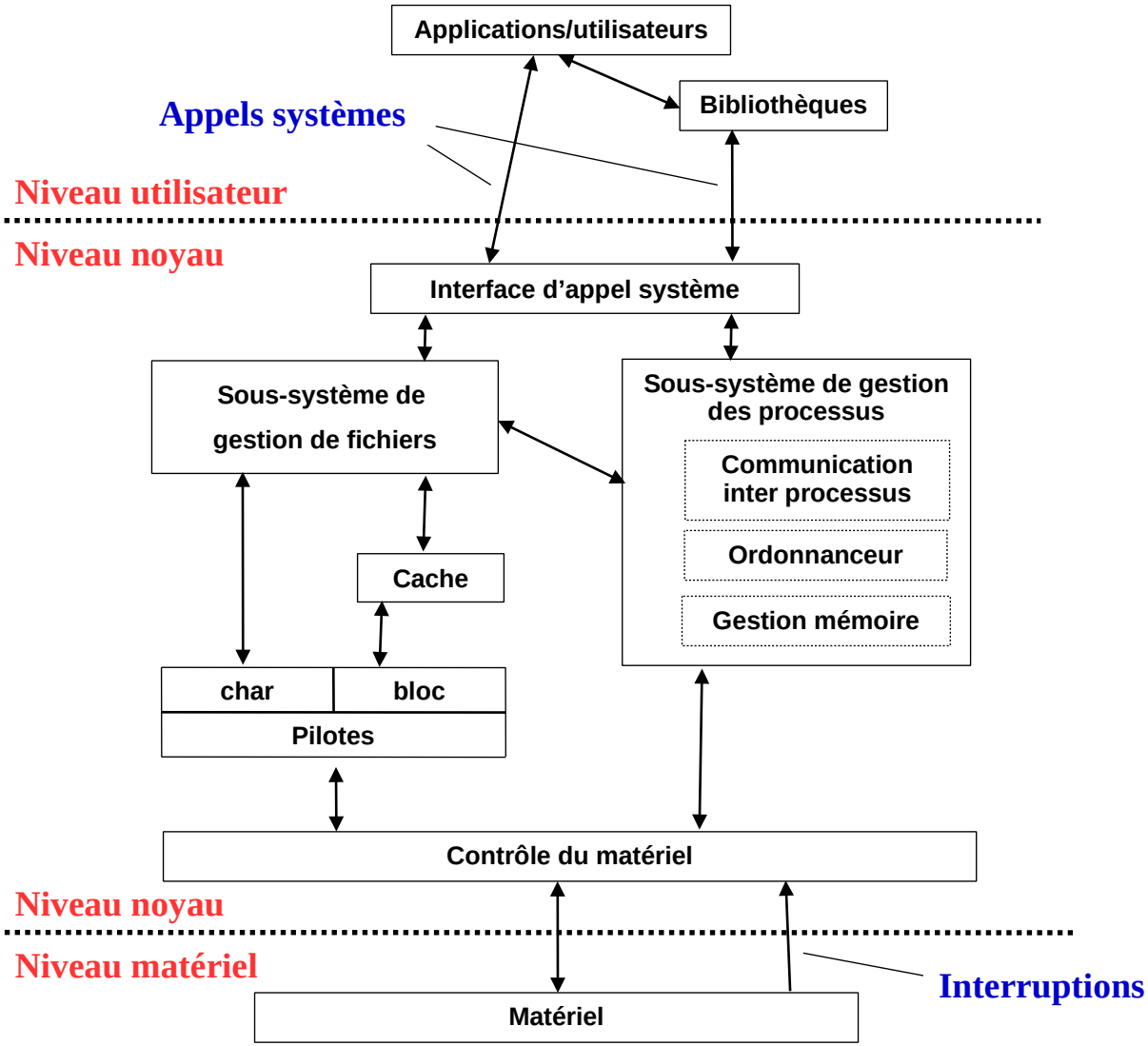
2. Architecture de Linux

Modes de fonctionnement du processeur



- Seul le noyau tourne en mode privilégié du processeur
  - ►mode “noyau”
- Les programmes qui tournent sur Linux utilisent le mode non privilégié du processeur
  - ►mode “utilisateur”

Appels systèmes



Noyau non préemptif

- Les appels systèmes (mode noyau) ne sont pas interruptibles
- Délai de réponse plus lent aux tâches de hautes priorités
- Meilleure bande passante CPU
- Programmation en mode noyau simplifiée

►Très bien pour un serveur

Noyau préemptif

- Les appels systèmes (mode noyau) sont interruptibles
  - (pas toujours *fully preemptible*!)
- Délai de réponse plus court aux tâches de hautes priorités
- Moins bonne bande passante CPU
- Programmation en mode noyau nettement plus complexe (synchronisation)

►Très bien pour un système temps réel

3. Chaîne de développement croisée

3. Chaîne de développement croisée

Développement “croisé”

- Le développement se fait sur une station de travail dont l'architecture est rarement identique à la cible embarquée
  - ex : PC de développement dispose d'une architecture x86 et la cible est ARM ou PowerPC
- Chaîne de développement
  - compilateur croisé et librairie C standard croisée
  - debugeur croisé
  - outils de manipulation des binaires (nm, ldd, objdump, ...)

3. Chaîne de développement croisée

Compilation manuelle

- Paquetages à compiler
  - binutils
  - gcc
  - glibc
- Ordre
  - l'ordre est important
  - gcc utilise les outils du paquetage binutils (ar, as, ld…)
  - la glibc utilise gcc

3. Chaîne de développement croisée

Compilation assistée

- Buildroot
  - chaîne de développement + système de fichiers racine
  - plus facilement configurable et plus léger
- OpenEmbedded
  - chaîne de développement + système de fichiers racine
  - le plus complet
- Crosstool-ng
  - uniquement chaîne de compilation

4. Compilation d'un noyau

4. Compilation d'un noyau

Téléchargement des sources

- Noyaux officiels (*vanilla kernel*) :
  - disponibles sur http://www.kernel.org
  - pour une version complète cliquer sur *tarball*
  - pour un patch d'évolution depuis la dernière version cliquer sur *patch*
- Noyaux constructeurs (basés sur *vanilla kernel*) :
  - fournis par les fabricants
  - modifications du code (drivers) non intégrées aux noyaux officiels

4. Compilation d'un noyau

Téléchargement des sources

- Décompression de l'archive :
  - compressée avec gzip/xz

    tar –xf linux-xxx.tar.gz/xz

- Remarque :
  - les sources des distributions sont patchées et différent fortement des noyaux *vanilla*

4. Compilation d'un noyau

Configuration du noyau

- Se déplacer dans le répertoire créé au moment de la décompression (linux-xxx)
- Obtention d'un fichier de configuration initial
  - make (<cible>_)defconfig : point de départ personnalisable
  - liste des cibles ayant une configuration par défaut
    - make ARCH=<arch> help
    - liste des architectures supportées dans le répertoire *arch*
  - exemple
    - make ARCH=arm tegra_defconfig

4. Compilation d'un noyau

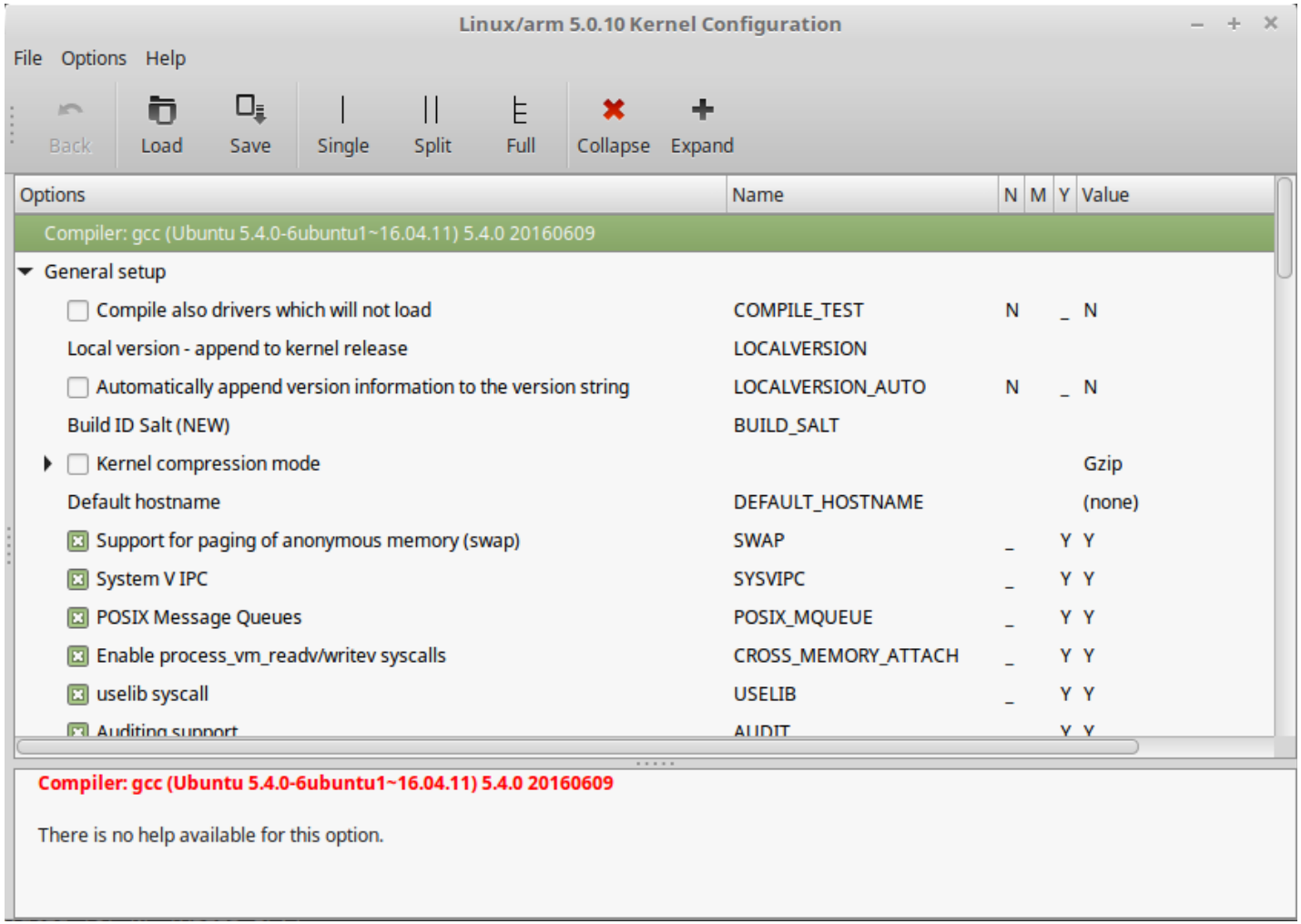
Configuration du noyau

Personnalisation :

- make config: version ligne à ligne (peu confortable)
- make menuconfig: version en mode texte avec menus
- make xconfig: version graphique (nécessite qt devel)
- make gconfig: version graphique améliorée (nécessite gtk devel)

4. Compilation d'un noyau

Configuration du noyau



4. Compilation d'un noyau

Noyau modulaire

- Activation :
  - choisir ‘Y’ dans *Enable loadable module support*
- Dans le cas d’un noyau modulaire, 3 possibilités sont offertes :
  - ‘Y’ pour insérer une fonctionnalité directement dans le noyau
  - ‘N’ pour ne pas gérer cette fonctionnalité
  - ‘M’ pour gérer cette fonctionnalité sous forme de module à charger dynamiquement (depuis l’OS)

4. Compilation d'un noyau

Noyau modulaire

- Avantages :
  - noyau plus petit
  - permet de changer un pilote de périphérique sans éteindre la machine
- Inconvénient :
  - consomme de l’espace disque (modules et outils pour les gérer)

4. Compilation d'un noyau

Device Tree

- Multitudes de SoC, cartes de dev, cartes extensions :
  - bus USB ou PCI permet la découverte automatique
  - mais pas la mémoire, CPUs, périphériques sur autres bus
  - autant de fichiers sources (*board* files) pour les initialiser dans le noyau
  - difficile à maintenir
  - nombreux *forks* de noyaux pour les gérer indépendamment

Device Tree

- Solution : Device Tree
  - noyau + générique capable de gérer différentes configurations matérielles
  - la configuration est décrite en dehors du noyau
  - changement de configuration sans recompiler le noyau

4. Compilation d'un noyau

Device Tree

- DTS : Device Tree Source
  - fichier texte décrivant le matériel du SoC, de la carte de dev, de ses extensions
  - description sous forme d'arborescence
  - se trouvent dans *arch/arm/boot/dts*
- DTB : Device Tree Blob
  - version compilée
  - le bootloader passe le DTB au noyau
  - DTB patchable dynamiquement par des *overlays*

Device Tree

- Exemple spi dans *bcm2711-rpi-4-b.dts*

```
&spi0 {
	pinctrl-names = "default";
	pinctrl-0 = <&spi0_pins &spi0_cs_pins>;
	cs-gpios = <&gpio 8 1>, <&gpio 7 1>;

	spidev0: spidev@0{
		compatible = "spidev";
		reg = <0>;   /* CE0 */
		#address-cells = <1>;
		#size-cells = <0>;
		spi-max-frequency = <125000000>;
	};

	spidev1: spidev@1{
		compatible = "joypinote_keypad";
		reg = <1>;   /* CE1 */
		#address-cells = <1>;
		#size-cells = <0>;
		spi-max-frequency = <125000000>;
	  poll-interval = <50>;
	};
```

4. Compilation d'un noyau

Compilation

- Compilation :
  - `make ARCH=<arch> CROSS_COMPILE=<préfixe de la chaîne de développement croisé> <cible>`
  - `<cible>` : bzImage, modules, dtbs
- Exemple :
  - `make ARCH=arm CROSS_COMPILE=arm-linux-`
  - sous entend que la chaîne de développement croisée pour arm est installée (dans le *path*) et que les exécutables s'appellent arm-linux-gcc, arm-linux-ar, arm-linux-ld, ...

4. Compilation d'un noyau

Installation

- Installation des modules :
  - `make (INSTALL_MOD_PATH=…) modules_install`
  - les modules sont installés dans le répertoire /lib/modules/<version_du_noyau>
  - les modules créés ne sont utilisables que sur ce noyau
- Installation du nouveau noyau :
  - `make (INSTALL_PATH=…) install`

4. Compilation d'un noyau

Autres cibles du Makefile

- `make clean :`
  - supprime tous les fichiers intermédiaires
- `make distclean :`
  - remet les sources dans leur configuration d'origine
  - supprime les fichiers intermédiaires mais aussi
    - les fichiers résultats (noyau et modules)
    - le fichier de configuration .config
- `make oldconfig :`
  - utilise configuration d’une version précédente du noyau

5. Création d'un système de fichiers

5. Création d'un système de fichiers

Répertoires indispensables

- /dev
  - contient les fichiers spéciaux de périphérique
- /bin
  - contient les fichiers exécutables
- /lib
  - bibliothèques nécessaires aux exécutables
  - modules noyau si nécessaires

5. Création d'un système de fichiers

Répertoires optionnels

- /proc
  - point de montage du système de fichier virtuel *procfs*
  - donne des informations sur les processus en cours de fonctionnement et plus généralement sur l'état du noyau
  - ce répertoire est vide tant que *procfs* n'est pas monté
- /etc
  - contient les fichiers de configuration du système

5. Création d'un système de fichiers

Les fichiers périphériques

- Création manuelle :
  - `mknod /home/user/rootfs/dev/périph_car c majeur mineur`
  - `mknod /home/user/rootfs/dev/périph_bloc b majeur mineur`
  - `cp [–dpR|-a] /dev/periph /home/user/rootfs/dev/periph`
    - d: copier les liens symboliques en tant que tels
    - p: conserver les propriétaire, groupe et droits d'accés
    - R: copier correctement les fichiers spéciaux
- Création automatique : démon udev

5. Création d'un système de fichiers

Les fichiers périphériques indispensables

- /dev/console: gère les entrées/sorties standard
  - pas de console => pas de clavier ni d'affichage…
- Liaison série :
  - /dev/tty0 et /dev/tty1: pseudo terminal
  - /dev/ttyS0: liaison série

5. Création d'un système de fichiers

Les fichiers périphériques optionnels

- /dev/null :
  - trou noir. Ignore tout ce qui lui est envoyé
- /dev/zero :
  - fontaine blanche. Renvoie des zéros tant qu'elle est lue
- Disques :
  - /dev/ram[0...]: ramdisks
  - /dev/sd*: disques SCSI ou SATA ou clés USB
  - /dev/hd*: disques PATA (« IDE »)

5. Création d'un système de fichiers

Exécutables indispensables

- /sbin/init ou /etc/init ou /bin/init ou /bin/sh
  - seul programme dont le nom est fixé en dur dans le noyau. Peut être directement l'application embarquée
- Un shell :
  - bash ou une version plus légère adaptée à l'embarqué (ash, busybox,…)
  - de nombreux programmes standard utilisent /bin/sh comme shell: ajouter un lien symbolique de /bin/sh vers le shell que vous avez choisi
- L'application embarquée!

5. Création d'un système de fichiers

Exécutables optionnels

- Commandes de manipulation de fichiers :
  - ls, cp, mv, mkdir, rm
- Commandes de manipulation des systèmes de fichiers :
  - mount, umount, du, df
- Commandes de manipulation des processus :
  - ps, kill
- Commandes de manipulation des modules :
  - insmod, rmmod, lsmod, modprobe

5. Création d'un système de fichiers

Librairies

- `ldd <exécutable>` donne la liste des librairies dynamiques utilisées par un exécutable

- Tous les programmes ont besoin de la librairie C standard (entrées/sorties de base, fonctions « wrapper » des appels systèmes, …)
- ld-linux.so est la librairie qui permet le chargement des autres librairies dynamiques. C'est la seule qui doit impérativement se trouver dans le chemin indiqué par ldd (généralement /lib). Ce chemin est inscrit en dur dans l'exécutable

5. Création d'un système de fichiers

Librairies

- Enregistrement des librairies :
  - /etc/ld.so.conf doit contenir la liste des répertoires contenant les librairies
  - (<arch>-linux-)ldconfig –r /home/user/rootfs crée le fichier /home/user/rootfs/etc/ld.so.cache
- Suppression des symboles inutiles :
  - (<arch>-linux-)strip –s libxxx.so
- Remarque :
  - .so signifie « shared object »

5. Création d'un système de fichiers

Scripts d'initialisation

- /etc/inittab
  - fichier « parsé » par le programme init pour déterminer les services à lancer au démarrage
- /etc/rcS.d/*
  - scripts lancés à chaque démarrage
- /etc/rc*x*.d/*
  - scripts lancés en fonction du runlevel *x*

5. Création d'un système de fichiers

Fichiers de configuration

- /etc/passwd définition des utilisateurs
- /etc/shadow les mots de passe hachés des utilisateurs
- /etc/group définition des groupes
- /etc/gshadow mots de passe hachés des groupes
- /etc/fstab options de montage des systèmes de fichier
- /etc/crontab tâches à effectuer régulièrement

5. Création d'un système de fichiers

Types de systèmes de fichiers pour l'embarqué

- JFFS2 (Journalized Flash File System v2) :
  - fiable :
    - spécialement conçu pour limiter l'usure des flashs
    - journalisé
  - lent
- UBIFS (Unsorted Block Images File System) :
  - plus performant que JFFS2 (montage plus rapide, vitesse écriture, accès aux fichiers volumineux)
  - compression zlib à la volée

5. Création d'un système de fichiers

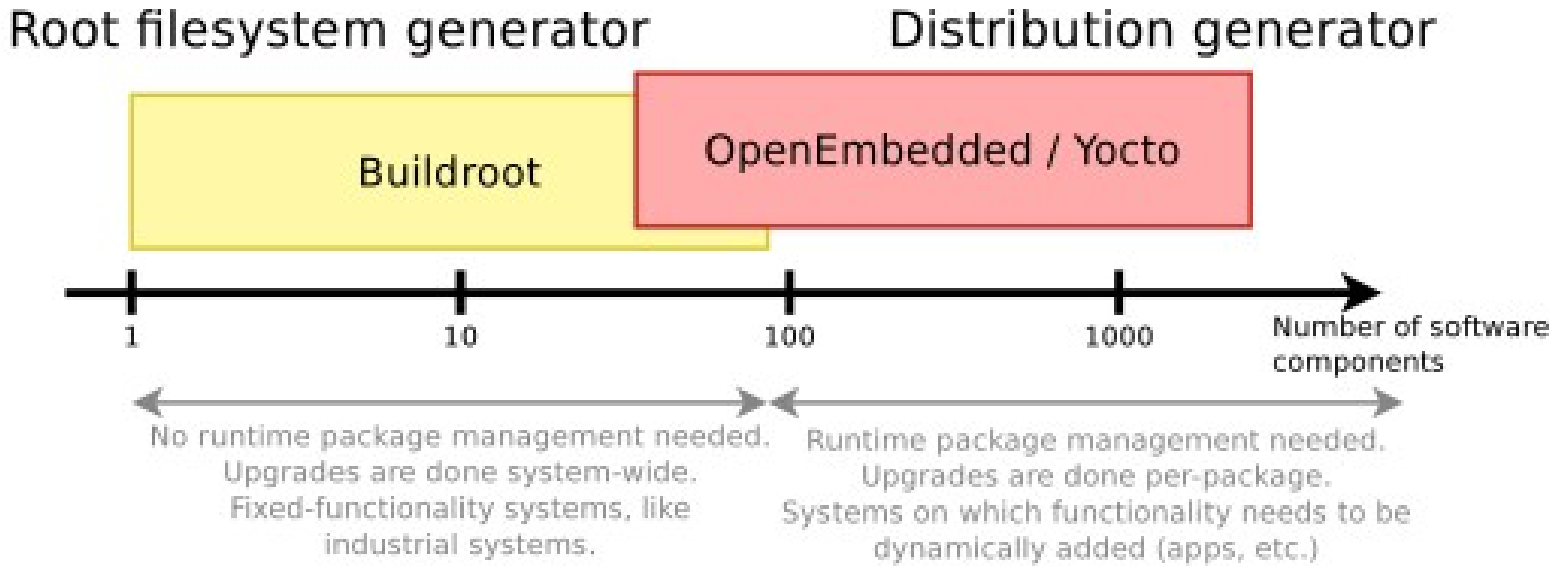
Types de systèmes de fichiers pour l'embarqué

- Initrd :
  - peut être inclus dans l'image du noyau
  - parfois étape indispensable
- Ext2 :
  - très rapide, faible empreinte
  - peu fiable (usure, pas de journalisation)
    - pas de problème en lecture seule
- FAT :
  - peu fiable, lent
  - compatible avec Windows

6. Distributions Linux embarqué

6. Distributions Linux embarqué

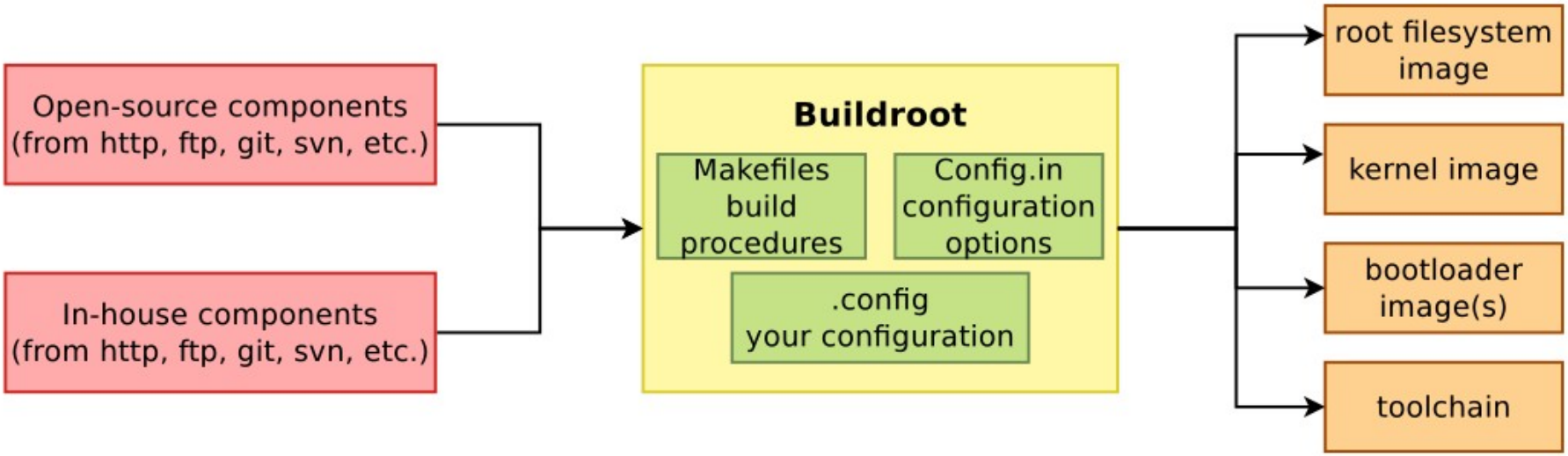
Yocto vs Buildroot vs others

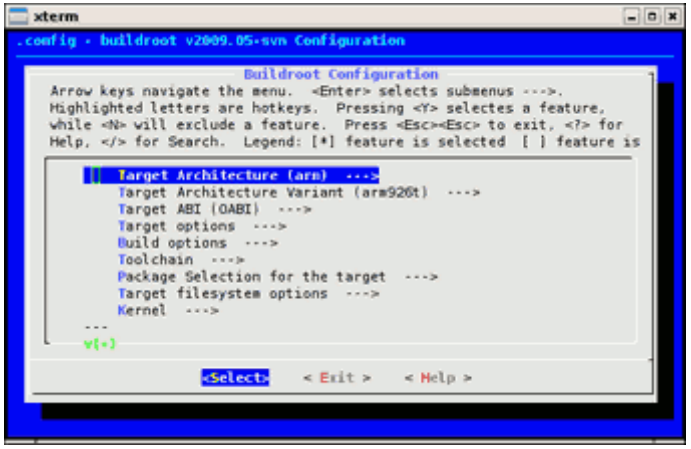


- LTIB (Linux Target Image Builder)
- ELDK (Embedded Linux Development Kit)
- ELBE (Embedded Linux Build Environment)

6. Distributions Linux embarqué

Buildroot





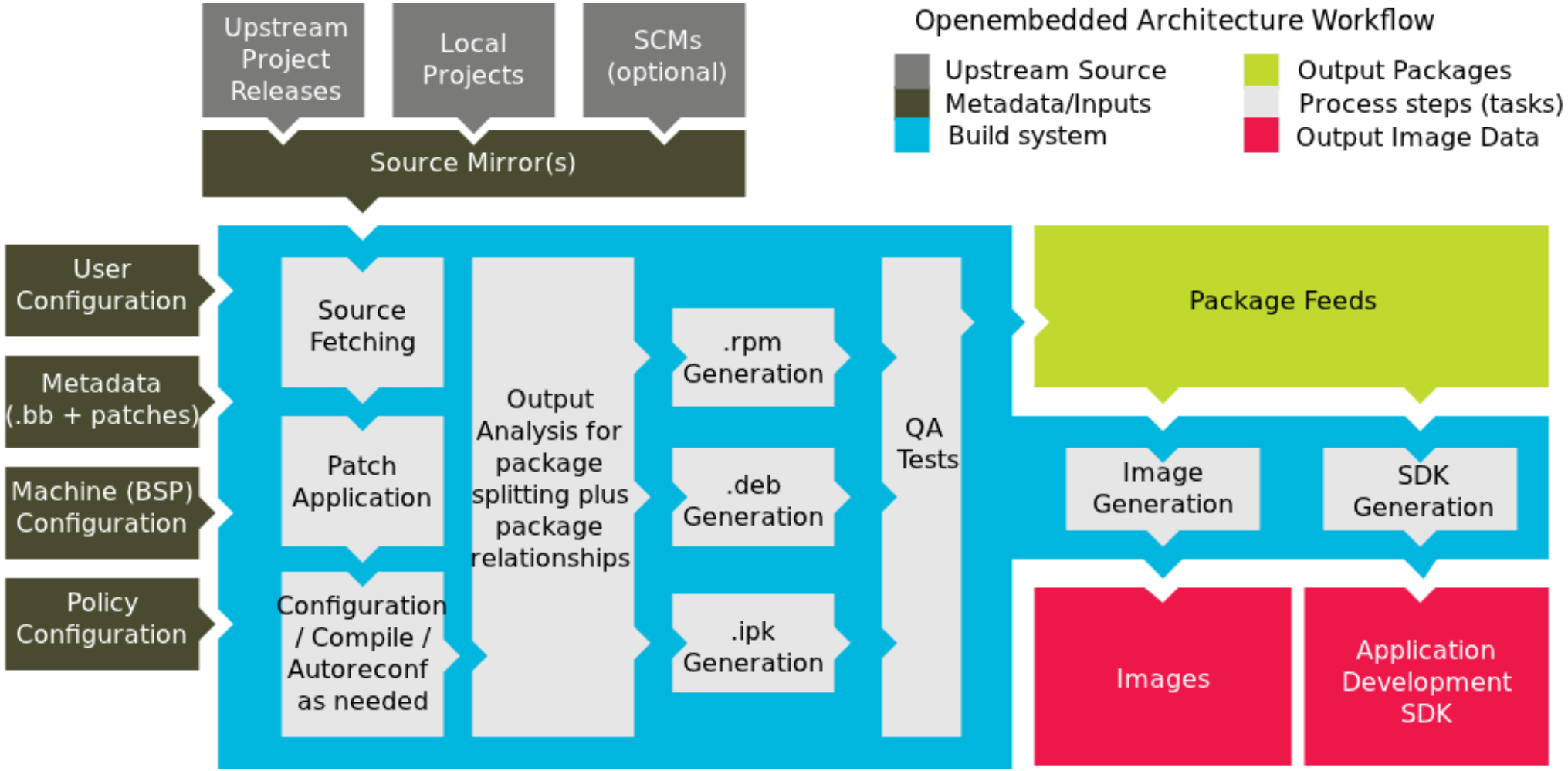
- Chaîne de compilation croisée
- Système de fichiers
- Image noyau
- Image bootloader
- Arch : x86, ARM, MIPS, PowerPC, etc

6. Distributions Linux embarqué

Yocto Project

- Projet de la Linux Foundation (Intel, Microsoft, Meta, Google, ARM, etc, etc)
- Basé sur le système de build d'OpenEmbedded :
  - BitBake : système de construction de paquets. Permet d'automatiser la chaîne de compilation d'une distribution à partir de sources locales ou sur le réseau (svn, git, wget, …). Utilise des recettes en entrée (*.bb).
  - Openembedded-core : layer/couche de base, contenant les recettes principales
- Poky : nom de la distribution de référence, générée par Yocto

Yocto Project



6. Distributions Linux embarqué

Yocto Project

- Arborescence simplifiée
  - meta/ — layer open embedded core
    - conf/ — configuration du layer
      - machine/ — configurations des cibles
    - classes/ — classes partagées
    - recipes-xxxx/ — recettes classées par groupes
  - meta-poky/ — layer poky
  - meta-yocto-bsp/ — layer yocto
  - oe-init-build-env — script d'initialisation

6. Distributions Linux embarqué

Yocto Project

- Extensibilité :
  - ajout de customs layers
    - pour du logiciel : ex ROS
    - pour du matériel : ex Raspberry
  - les layers ont des priorités
    - gère les conflits si même recettes
  - possibilité d'étendre des recettes existantes
    - via des fichiers .bbappend

7. Séquences de démarrage

7. Séquences de démarrage

x86 legacy

- System startup
  - BIOS
- Stage 1 bootloader
  - MBR (master boot record)
- Stage 2 bootloader
  - Lilo, Grub, etc
- Kernel
- Init
- User prompt

7. Séquences de démarrage

Raspberry Pi 4 (ARM, BCM2711)

Load and run First stage bootloader (ROM)

immuable

Load and run Second stage bootloader (EEPROM)

détermine si boot sd-card, usb ou réseau

Load and run Firmware (start4.elf)

Load kernel.img, dtb, config.txt, cmdline.txt

Run Kernel

GPU

CPU

7. Séquences de démarrage

Raspberry Pi 4 (ARM, BCM2711)

- Le firmware peut se charger :
  - sur un support de stockage, par le réseau
- Le noyau et sa configuration peuvent se charger :
  - sur un support de stockage, par le réseau
- Le système de fichiers peut être :
  - sur un support de stockage, sur un partage réseau

Pour développer sur le noyau et le système de fichiers, le plus souple est l'utilisation du réseau.

7. Séquences de démarrage

A partir du second bootloader, par le réseau :

- Second bootloader: requête BOOTP (Bootstrap Protocol)
  - configuration IP
  - récupération de l'adresse du serveur TFTP (Trivial File Transfer Protocol)
  - téléchargement TFTP du firmware
- Firmware
  - téléchargement TFTP du noyau et de sa configuration
- Noyau:
  - requête DHCP, configuration IP
  - serveur et emplacement du système de fichiers via la cmdline (root=/dev/nfs nfsroot=192.168.0.1:/tftpboot/rootfs,vers=3)
  - montage du partage réseau NFS (Network File System)

8. Outils de développement

8. Outils de développement

Émulateur de terminal

- minicom, cutecom, gtkterm, putty, etc
- Régler les paramètres :
  - périphérique série utilisé (/dev/ttyS?)
  - bps, parité, nombre de bits de stop
  - coupure des lignes

Serveur DHCP

Dynamic Host Configuration Protocol

- Prend en charge BOOTP
- Permet à la cible de récupérer automatiquement :
  - son adresse IP et l'adresse du serveur
  - (le nom du noyau à charger)
  - (le nom du répertoire à utiliser comme système de fichier racine)
- Utile pour le noyau (et le système de fichiers)

Serveur DHCP

Exemple: udhcpd

```
# Sample udhcpd configuration file (/etc/udhcpd.conf)

# The start and end of the IP lease block

start        192.168.0.20       #default: 192.168.0.20
end          192.168.0.254      #default: 192.168.0.254

# The interface that udhcpd will use

interface    eth0               #default: eth0

# Currently supported options, for more info, see options.c
opt    tftp  192.168.0.1
```

Serveur TFTP

Trivial File Transfer Protocol

- Protocole de transfert de fichier basique utilisé pour télécharger le noyau sur la cible
- Attention, le bootloader du Raspberry Pi télécharge les fichiers via TFTP dans un sous dossier du nom de son numéro de série !

Serveur TFTP

Exemple: tftpd-hpa

```
# /etc/default/tftpd-hpa

TFTP_USERNAME="tftp"
TFTP_DIRECTORY="/tftpboot"
TFTP_ADDRESS=":69"
TFTP_OPTIONS="--secure"
```

Pour Raspberry Pi mettre les fichiers dans le dossier */tftpboot/serial_num/*

Serveur NFS

Network File System

- Permet à la carte de monter un répertoire de la machine de développement comme système de fichier racine
- Permet, pendant le développement, d'utiliser un système de fichier
  - de taille quasi infinie
  - modifiable dynamiquement

Serveur NFS

Exemple: nfs-kernel-server

```
#/etc/exports

/tftpboot/ 192.168.0.* (rw,sync,no_subtree_check,no_root_squash)
```

8. Outils de développement

Compilation / debugage

- Utiliser le compilateur croisé
  - `CC=arm-linux-gcc (--sysroot=/chemin/vers/chaîne)`
- Sur la cible :
  - `gdbserver :10000 /chemin/du/programme`
  - charge le programme et attend sur le port 10000
- Sur la machine de développement :
  - `gdb /chemin/local/du/programme`
  - `target remote <ip de la cible>:10000`
  - l'exécutable doit contenir les symboles de debug

8. Outils de développement

Compilation de paquetages

- Décompression de l'archive tar
  - `tar -xf paquetage.tar.gz/bz2/xz`
- Configuration et vérification des dépendances
  - `./configure help`
  - liste des options activables/désactivables
    - `--enable-OPTION / --disable-OPTION`
  - liste des options à inclure ou exclure
    - `--with-OPTION / --without-OPTION`

8. Outils de développement

Compilation de paquetages

- Exemple :

```
./configure \
--target=arm-linux \
--build=i386-pc-linux-gnu \
--prefix=/usr
```

- Construction (essentiellement compilation) :
  - `make`
- Installation :
  - `make DESTDIR=/répertoire/rootfs install`

Questions?