COURS
LINUX EMBARQUÉ

Gilles BLANC — EISTI novembre-décembre 2016

LICENCE

Creative Commons
Paternité - Pas d'Utilisation Commerciale - Partage des Conditions Initiales à l'Identique 3.0 France

Vous êtes libre :

- de reproduire, distribuer et communiquer cette création au public,
- de modifier cette création.

Selon les conditions suivantes :

Paternité. Vous devez citer le nom de l'auteur original.


Pas d'Utilisation Commerciale. Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.


Partage des Conditions Initiales à l'Identique. Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

À chaque réutilisation ou distribution, vous devez faire apparaître clairement aux autres les conditions contractuelles de mise à disposition de cette création.

Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits.

Ce qui précède n'affecte en rien vos droits en tant qu'utilisateur (exceptions au droit d'auteur : copies réservées à l'usage privé du copiste, courtes citations, parodie...)

Le texte intégral de la licence applicable à ce document se trouve à l'adresse : http://creativecommons.org/licenses/by-nc-sa/3.0/legalcode

POINTS COMMUNS ?

- Appareils électroniques
- Logiciel embarqué
- Linux embarqué

GÉNÉRALITÉS SYSTÈMES EMBARQUÉS

- « Embedded systems »
  - Systèmes enfouis
    - Systèmes informatiques où on ne le soupçonne pas
    - Généralisation usuelle : systèmes « ni bureautique, ni serveur ».
- 1ère Caractérisation : ressources limitées
  - Adaptations spécifiques
  - Approche adaptée
- Architectures différentes
  - ARM, PPC, MIPS, x86, m68K/ColdFire
  - Distinction avec MMU (Linux)/sans MMU (µClinux)

ARCHITECTURES MATÉRIELLES

- Matériel industriel
  - gammes de température, résistance aux poussières, etc.
  - durci (casing)
- Architecture microprocesseur
  - héritage du grand public (PC, et avant PowerPC/RISC)
  - spécifique mobilité/CE (ARM, MIPS, SH4)
  - très spécifique : en voie de totale disparition (processeurs Thomson pour satellites par exemple)
- CPU + RAM + { SSD, Flash, eMMC }
- carte (PCB avec fonction CPU) :
  - de développement
  - tout intégré/industrielle
  - sur module & porteuse
  - totalement personnalisée (et made in China)

DISPARITÉ DES ARCHITECTURES

- PC/x86
  - actuellement, essentiellement Atom ou i3/i5/i7
  - anciennement : de l'AMD, du PC104
  - point fort de (rétro)compatibilité, même architecture que l'environnement de développement PC
- ARM
  - le gros du marché grâce au Consumer Electronic (téléphones, tablettes, etc.), entrée dans le «pur» industriel récente
  - du M0 (concurrent du PIC) au Cortex-A9 multicoeur surpuissant
- Mips (routeurs/modems), SH4 (STB/tv), PowerQUICC II/III (indus/calculs) : très gros déclin

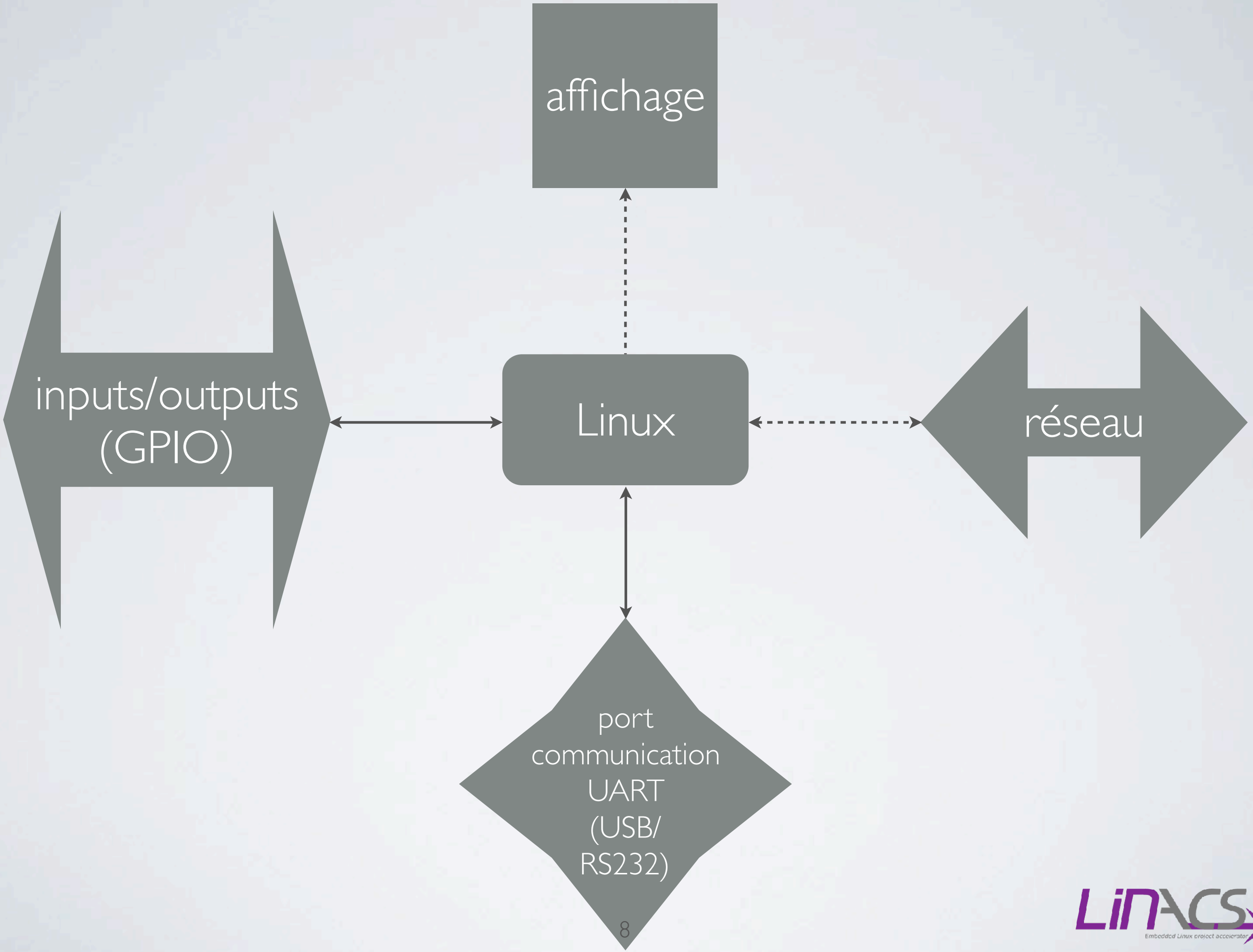
CONTRAINTES SYSTÈMES EMBARQUÉS

- Peu de ressources
  - Quantité de RAM faible
  - Pas toujours de disque dur (pas de swap)

  Donc on ne peut pas toujours compiler depuis le système
- Processeur généralement différent de la station de travail
  - Architecture différente
  - I/O spécifiques (embarqué : souvent encore programmation directe)
  - Périphériques spécifiques (Flash par exemple)

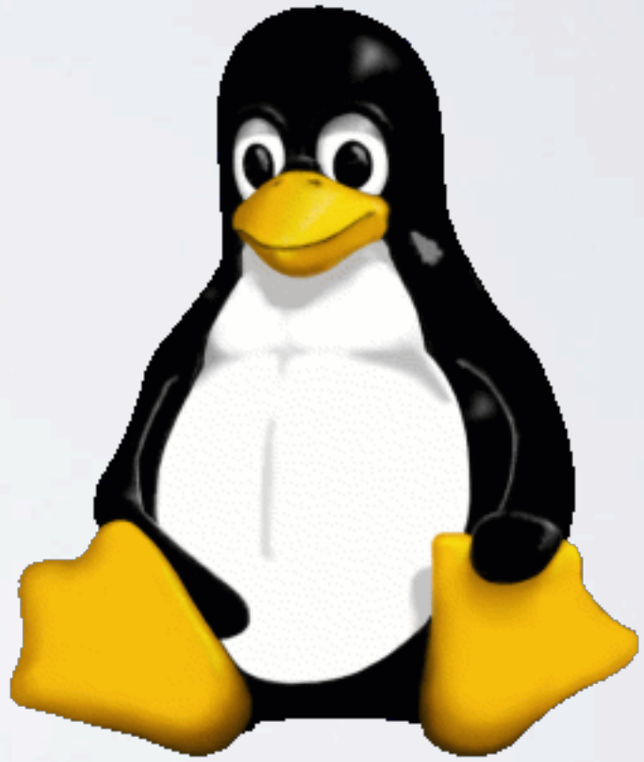
  Donc on ne peut pas développer et tester depuis la station
  (sauf à passer par une émulation logicielle)

  Solution : compilation croisée.


affichage
inputs/outputs
(GPIO)
Linux
réseau
port
communication
UART
(USB/
RS232)

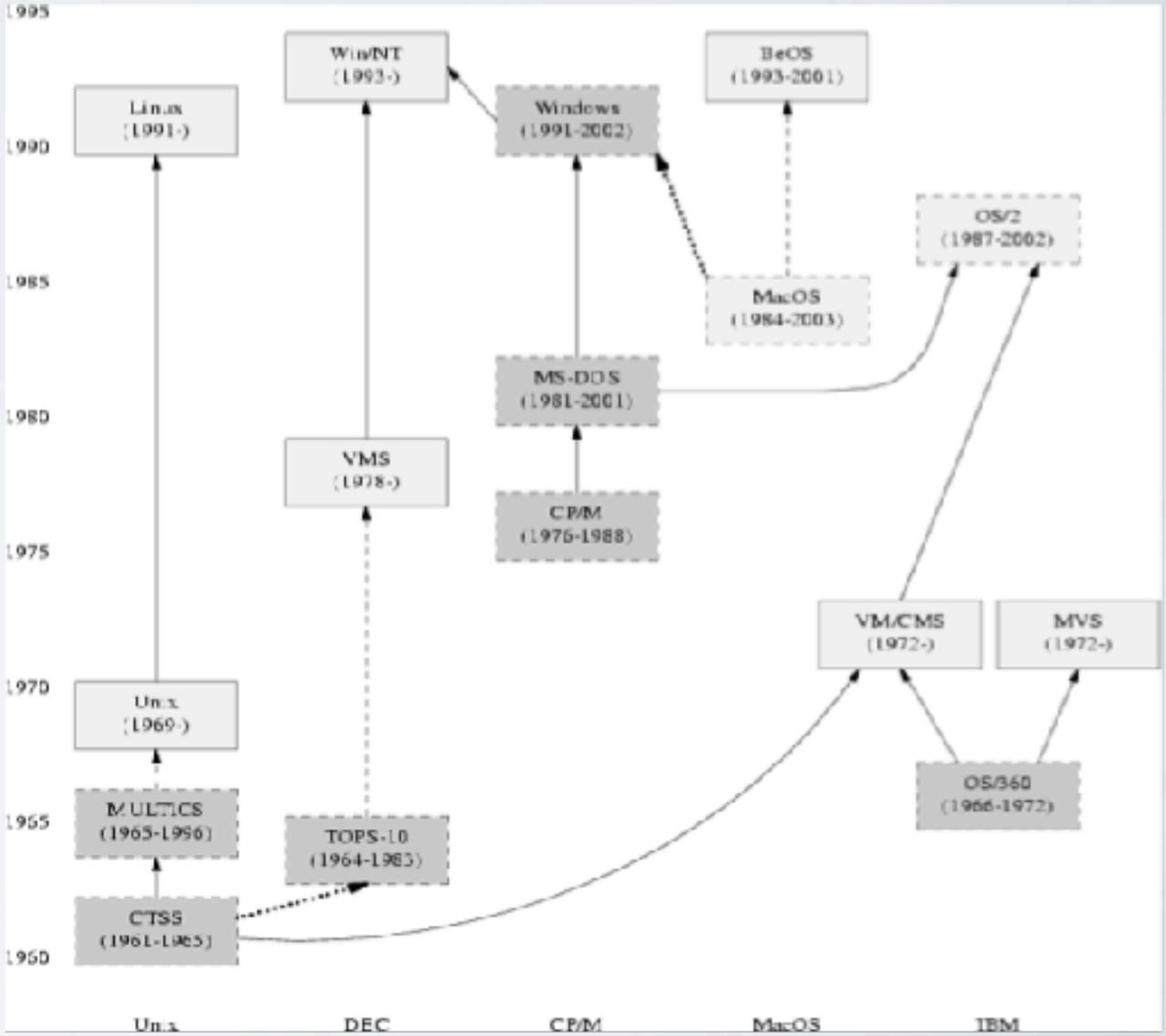
QU'EST-CE QUE LINUX ?

- un **noyau**, un **système d'exploitation**
- à destination du PC, puis d'autres architectures
- un projet communautaire
- un système **libre**
- un héritage ***NIX**



HISTORIQUE

- « Linux is evolution, not intelligent design », Linus Torvalds
- 1991: première annonce sur usenet (Freax)
- 1992: v0.96 GPLed pleinement fonctionnelle
- 1996: v2.0
- 1999: entrée en bourse de RedHat
- 2001: v2.4
- 2003: v2.6
- http://en.wikipedia.org/wiki/History_of_Linux



NOYAU

espace utilisateur

Application

Application

Application

...

Bibliothèques

Appels système

noyau Linux

systèmes de fichiers

protocoles

mémoire virtuelle

processus

modèle de périphériques, modules

gestion réseau

gestion mémoire

gestion tâches (ordonnanceur)

pilotes caractère

pilotes bloc

pilotes USB

pilotes réseau

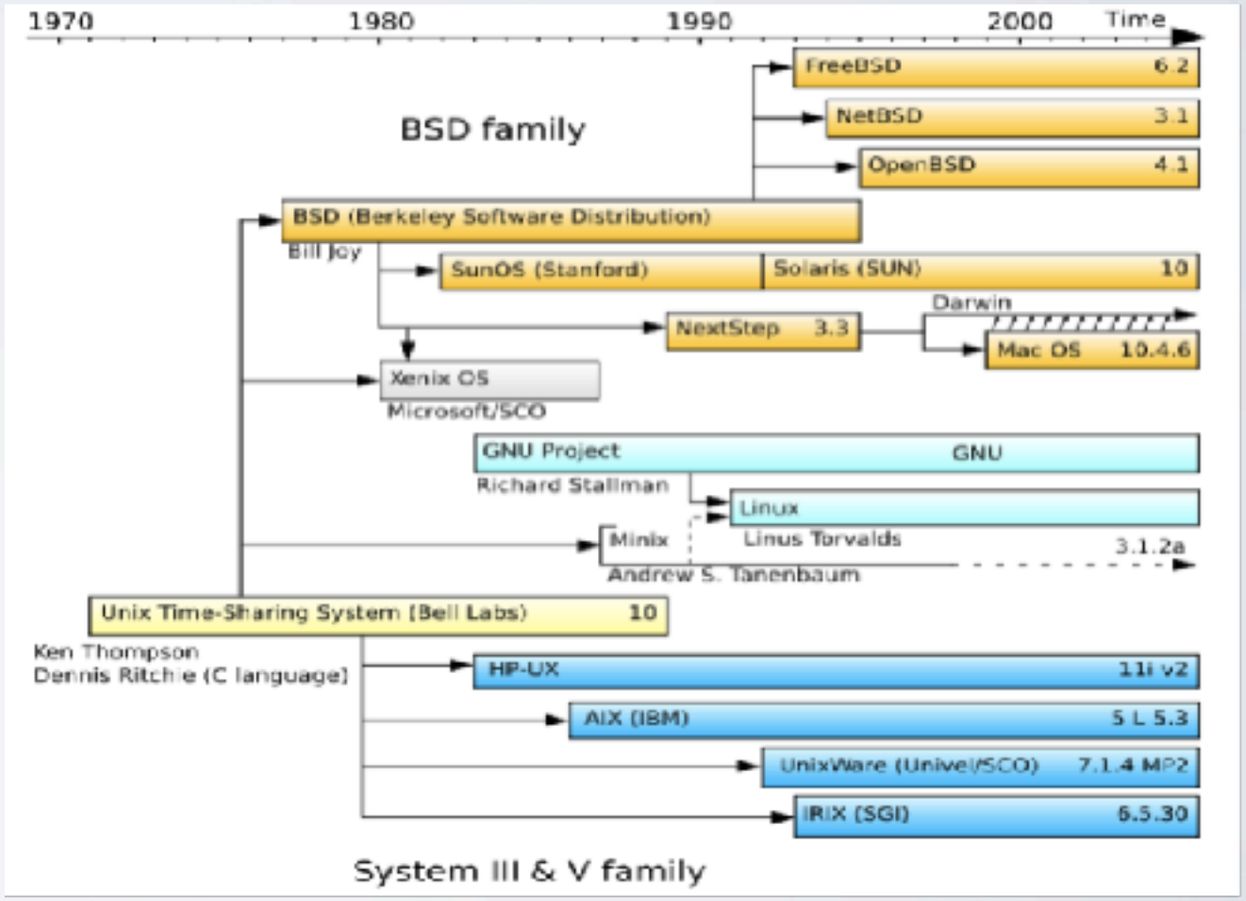
interruptions IRQ

IRQ

matériel

SYSTÈME D'EXPLOITATION

- héritage des UNIX
  - tout est fichier
    - arborescence : /, /dev, /etc, ...
    - noyau séparé du système
- distributions
- GNU/Linux : noyau Linux + système GNU



LE LOGICIEL LIBRE

- une philosophie
- origine : le projet GNU (http://gnu.org)
    - 84/85 : lancement du projet GNU par RMS, puis FSF
    - fin 80's : projet d'OS finalisé (gcc, binutils, glibC, emacs), à l'exception du noyau
    - juin 91 : licence GPLv2
- 4 libertés :

0. la liberté d'exécuter le programme, pour tous les usages,

1. la liberté d'étudier le fonctionnement du programme et de l'adapter à ses besoins,

2. la liberté de redistribuer des copies du programme (ce qui implique la possibilité aussi bien de donner que de vendre des copies),

3. la liberté d'améliorer le programme et de distribuer ces améliorations au public, pour en faire profiter toute la communauté.

LOGICIEL LIBRE

UTILISER
dans n'importe quel but (liberté 0)

APPRENDRE ADAPTER
le logiciel à ses besoins (liberté 1)

REDISTRIBUER
des exemplaires au public (liberté 2)

PUBLIER LES AMÉLIORATIONS
pour qu'elles profitent à toute la communauté (liberté 3)

UNE FAÇON ÉTHIQUE DE COMPRENDRE LE LOGICIEL
dans son développement, sa commercialisation, sa distribution et son utilisation

VALEURS
Éthique, créativité, efficacité, science, non-discrimination, vie privée, securité, transparence, coopération, solidarité et surtout liberté

est
exprime
permettent de
nécessite
requiert
distribue
pour
pour

LICENCES LIBRES
GPL (Copyleft), Apache, BSD, Creative Commons (partage à l'identique)

DISTRIBUTIONS
Debian, Ubuntu, OpenBSD, Trisquel, OpenSolaris, gNewSense
et OpenSUSE, Fedora, uClinux...

CODE SOURCE
instructions

produisent et corrigent
compilé
avec
réunis dans
respectent

CONTENUS LIBRES
polices typographiques, traductions, localisations, modèles, sons, images, FAQs, guides, manuels

PROGRAMMES EXÉCUTABLES
Moodle, Audacity, Blender, GIMP, Inkscape, Firefox, Scribus, LibreOffice
100% libre Icecat

STANDARDS OUVERTS
UTF-8 (IETF)
HTML, XML (W3C)
ODF (OASIS et ISO)

DÉVELOPPEURS
Fondations comme la FSF, et projets comme :
Enlightenment, Linux kernel, KDE, MySQL, Window Maker, PHP, GNU, Gnome, X.org, Sugar, Apache

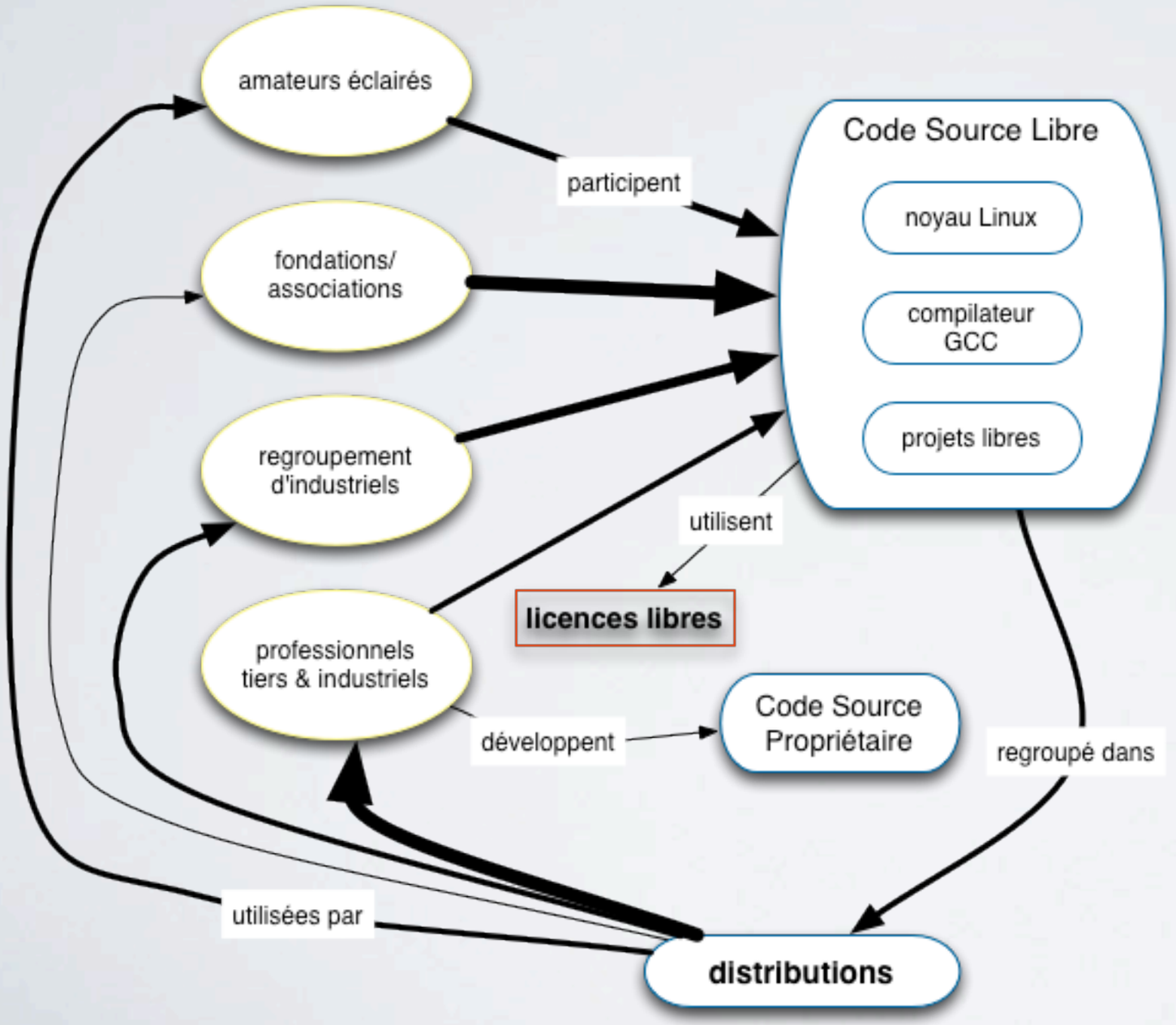
collaborent avec
choisissent

UTILISATEURS
ONG : Wikipédia, Greenpeace
INSTITUTIONS : UNESCO, NASA
ENTREPRISES : Google, IBM
PAYS : Espagne, France, La Chine, Brésil, Allemagne
UNIVERSITÉS : MIT, Berkeley
89% du TOP500 des superordinateurs

reçoivent et donnent

SUPPORT
ÉVÈNEMENTS : RMLL, FOSDEM **ASSOCIATIONS :** April, Aful, Framasoft
GUL : Abul, Linux62, Pariux, **BLOGOSPHÈRE :** Phoronix.com, blog.cfset.org
WIKIS : EmacsWiki, wiki.debian.org **FORUMS :** lea-linux.org, LinuxQuestions
IRC (Chat) : OFTC, freenode **LISTES DE DIFFUSION :** debian-i18n, blag-users
RENCONTRES : Solutions linux, aKademy, GHM (Gnu Hackers Meeting)

ÉCOSYSTÈME LIBRE EMBARQUÉ



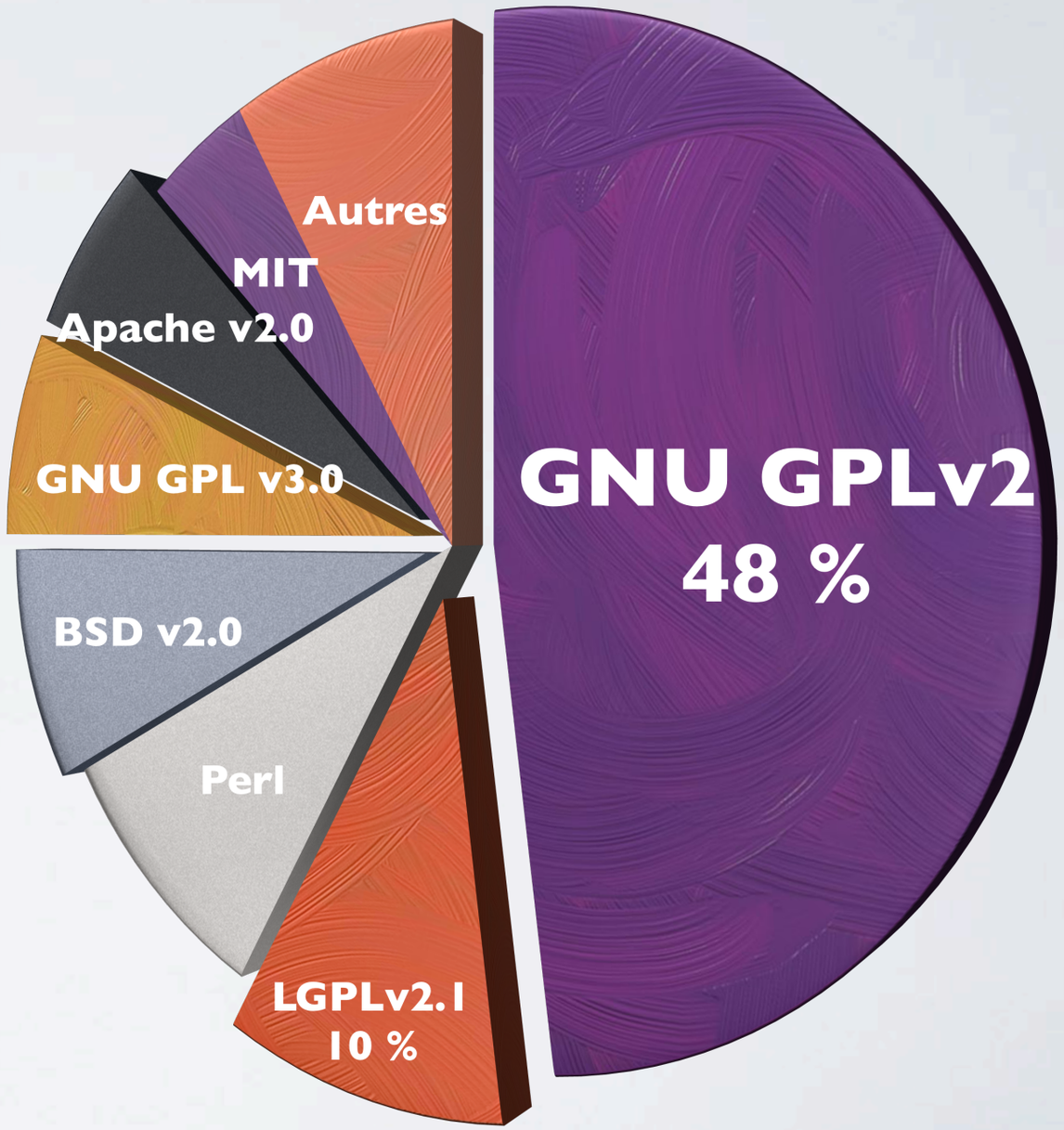
- écosystème complexe
- mêle du libre et du propriétaire
- ceux qui participent le plus ne sont pas ceux qui utilisent le plus

LES LICENCES

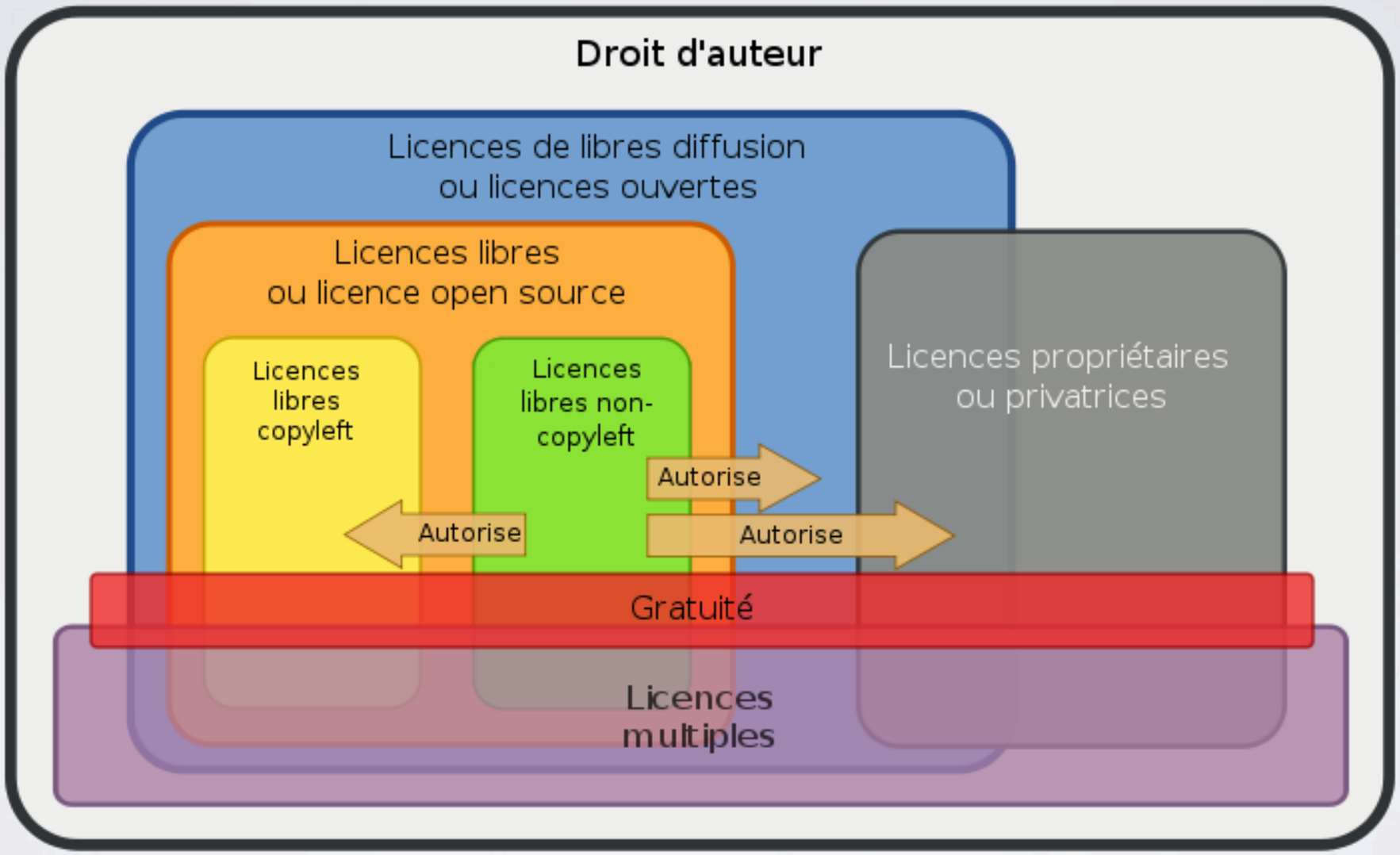
- un contrat : contre paiement ou non
- entre un/plusieurs auteur(s) du logiciel et un « utilisateur »
- le contrat se forme lors de l'usage = avoir en possession l'exécutable ou le code source, exécuter, compiler, décompiler, etc.
- cession de droits au choix : exécution, étude (code source, décompilation), modification, etc.
- potentiellement sous conditions : droit d'exécuter mais de modifier seulement contre paiement, etc.

LES LICENCES LIBRES

- s'applique au code source ; garantit les 4 libertés (différent de l'OpenSource)
- plusieurs types
  - copyleft
    - GNU GPL, GNU LGPL, CeCILL
    - implique de laisser le code source sous la même licence, avec cession de droits automatiques : le logiciel reste libre
- permissives
  - BSD, Apache
  - permet de rajouter n'importe quelle licence au code source, y compris propriétaire + pas d'obligation de redistribution du code (fermeture du code source)



CLASSIFICATION DES LICENCES



http://fr.wikipedia.org/wiki/Fichier:Classification_des_licences.svg

DROITS ET DEVOIRS

- Droits : modifier, exécuter, redistribuer, redistribuer la version modifiée

- Licences de type GNU GPL :

  - devoir de redistribution à l'utilisateur (**au client**)

  - GPLv3 : pensée pour des problématiques embarquées
    ➜ interdiction de TiVoization (DRM sur firmware)

- Licences permissives : possibilité de « fermer » le code

SURVOL JURIDIQUE (1)

- Licences impliquées :
  - GPLv2
    - Liberté d'utiliser, de modifier, de vendre
    - La liberté se transmet au client
      - Donc tout code embarquant du code GPL est nécessairement GPL.
    - Cas du kernel : tout driver lié statiquement est GPL
      - Driver hors GPL : nécessairement en module externe.
        En toute rigueur, il faudrait même prendre garde à ne pas inclure de fichier GPL dans le driver (en particulier des fichiers include '.h')
  - LGPL (LesserGPL)
    - Licence créée pour la libc
    - Permet de lier (dynamiquement) du code non-GPL (voire propriétaire) à du code LGPL

SURVOL JURIDIQUE (2)

- Licences (suite)
  - BSD / Apache (licences permissives)
    - Autorisation de modifier, d'utiliser, de revendre
    - Autorisation de fermer le code issu du projet
    - Obligation : citation des auteurs d'origine
- Licences particulières
  - Ecos : GPL mais autorise de plus la liaison statique avec une application propriétaire sans impliquer la mise en GPL dudit code.

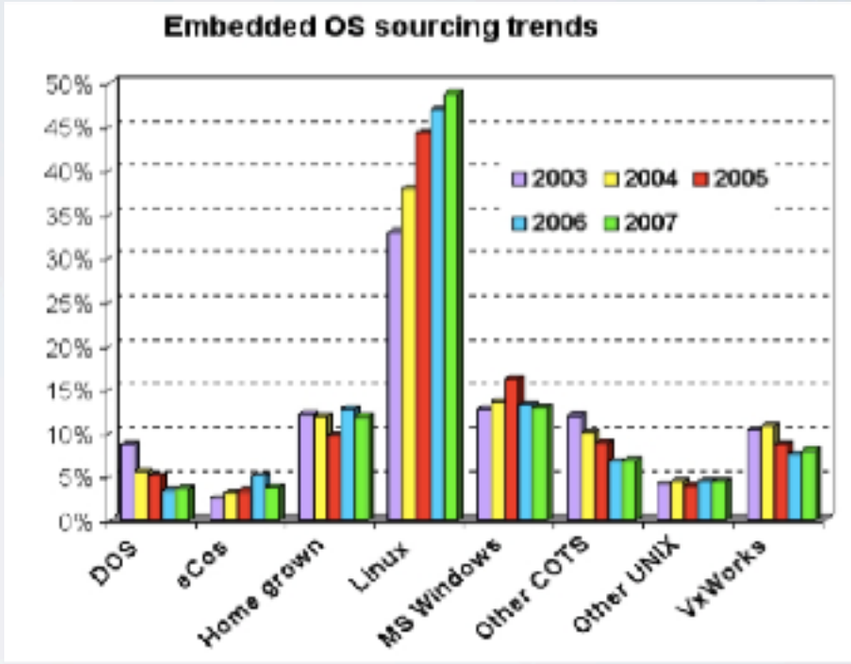
LIBRE EMBARQUÉ

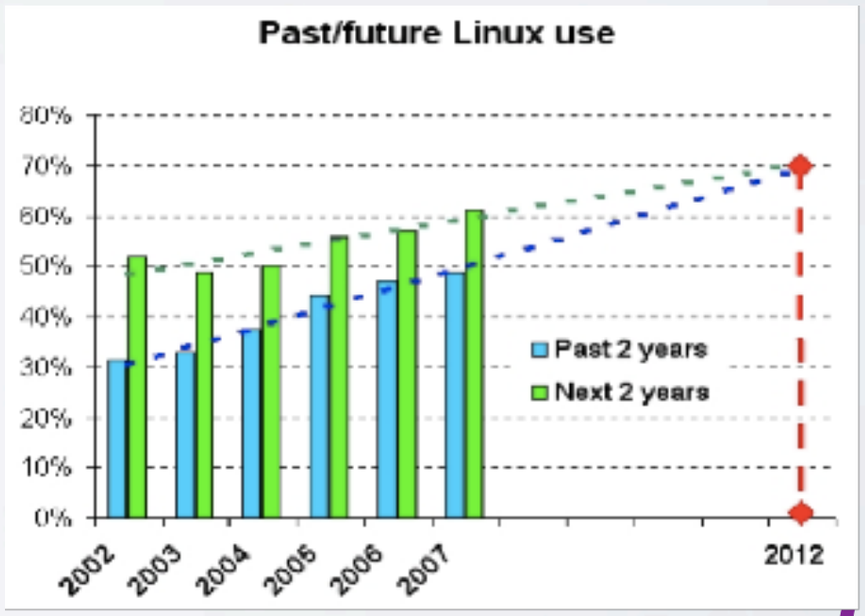
- au-delà de Linux : un mouvement
  - bootloader : u-boot, grub, ...
  - compilateur : GCC, LLVM
  - outils divers de développement/intégration : Eclipse, GPS, SVN, Git...
  - logiciels
  - polices de caractères
  - environnements complets : GPE, Android, MeeGo, Tizen, ...
  - OS embarqués : eCos, RTMS, FreeRTOS, Pok, ...
- au-delà du libre : les formats ouverts
  - communication : XMPP, UPnP
  - image : png/mng, svg
  - vidéo/son : ogg, WebM
  - compression : gzip, bzip2, LZMA, 7z
- matériel libre ! (OpenHardware)
- Certification ! (OpenDO, Opees, Babylone)

LINUX EMBARQUÉ : QUOI ET POURQUOI ??

LINUX EMBARQUÉ

- premier portage (DEC Alpha) en 1995 ; architectures suivantes : PPC, m68k, ARM, MIPS
- démarrage du projet « LinuxCE » en 1996
- produits depuis ~2000
- éclosion réelle avec les version 2.4 (2004)
- explosion avec la version 2.6 (2004/2005)
- tournant en 2007
- intégration de µClinux
- acteur incontournable à présent (3 projet / 4 !)





EMBARQUÉ & TEMPS RÉEL

- Définition du TR : déterminisme temporel
- Des mondes qui se recoupent
- TR « mou » vs « dur » : Linux TR mou par défaut
- Solutions TR dur :
  - Linux, par modification du code : RTLinux, MontaVista, RTAI, Concurrent Computer
  - Par paravirtualisation : Xenomai/ADEOS, Wind River Hypervisor, PikeOS (Sysgo), Integrity (GreenHills), ...

SYSTÈMES EMBARQUÉS OUVERTS

- Linux : système généraliste
  - Portable sur de nombreuses architectures (~70), 32bits
  - Large disponibilité de pilotes
  - N'est pas temps-réel dur
- RTLinux, RTAI, Xenomai
  - Noyau temps réel dont le système Linux est une tâche non temps-réel (donc moins prioritaire)
- RedHawk Linux, MontaVista
  - Patches au kernel Linux, rendant le kernel Linux temps réel.
    Notons qu'une partie des patches de MontaVista sont déjà intégrés au noyau courant (patches low-latency).
- Ecos : Système léger, temps réel dur
  - Équivalents commerciaux : Nucleus, QNX.
  - Ne propose pas tout-à-fait le même modèle d'exécution (pas de *shell*, pas nécessairement de partition '/', licence adaptée).

POURQUOI LE LIBRE DANS L'EMBARQUÉ ?

- Nombreux avantages pour l'industriel :
  - technicité (à la Torvalds)
    - amélioration continue
    - GPL/LGPL : amélioration « forcée »
    - qualité du code
    - interactions/documentation
  - pérennité/éthique (à la RMS)
    - image positive (développement durable)
    - aucun contrôle absolu, garantie d'existence sur le long terme
  - couts (question entrepreuriale)
    - pas de royalties (problème majeur dans l'embarqué)
    - sources gratuites
- mais libre n'est pas gratuit !

➜ couts cachés : couts de réalisation, couts de formation, couts de migration, etc.

INCONVÉNIENTS DU LIBRE DANS L'EMBARQUÉ

- manque d'interlocuteurs identifiés : problème technique (assistance) et contractuel (responsabilité)
- manque de contrôle : dépendance à la communauté
- contraintes juridiques (propriété intellectuelle) : comprendre les licences libres
- problème de compétences : expertise et expérience rares
- difficulté d'estimations en temps et en couts (cf. effet IKEA !)
- beaucoup d'aléas à estimer
- de nouveaux paradigmes à maîtriser pour une industrie à forte inertie

POURQUOI LINUX EMBARQUÉ ?

- Libre : GPLv2 ➜ forte évolutivité, modifiable/adaptable
  - cible/marché très étendu :
    du très embarqué au Consumer Electronic
  - temps réel : « mou » ou « dur »
- écosystème existant : GNU/Linux
- disponibilité de développeurs formés
- nombreux points forts techniques

POINTS FORTS DE GNU/ LINUX POUR L'EMBARQUÉ

- mérites du système
  - faible empreinte mémoire par rapport aux capacités (notamment pour les drivers)
  - efficace, optimisé, fortes performances
  - très stable : uptime
  - espace utilisateur (userland) efficace

- réalisation
  - beaucoup d'outils disponibles, de bibliothèques (briques élémentaires très fonctionnelles)
  - support de nombreuses technologies, natives ou sous VM, notamment Java (et HTML)
  - développement userfriendly (hôte ~ cible)

- architecture interne du noyau
  - modulaire, paramétrable très finement
  - énorme support matériel
  - portabilité : ~75 architectures, 32 bits, MMU ou non
  - TCP/IP (serveurs !), bluetooth, multimédia, ... : de base
  - ordonnanceur de complexité O(1) puis O(log n), préemptible (à partir de 2.6)

DISTRIBUTIONS : DÉFINITION

- Une distribution Linux, c'est :
  - un ensemble de logiciels dont le noyau Linux ;
  - une procédure de préparation du système ;
  - une procédure de fabrication de l'arborescence du système ;
  - une procédure d'installation ;
  - **un outil de gestion des paquets binaires et dépendances ;**
  - **un outil de gestion de paquets source et dépendances :**
    - des distributions Linux gèrent les logiciel par « paquetages », plusieurs systèmes concurrents de gestion de paquets existent : *rpm* (RedHat Package Management), *deb* (Debian), *portage* (Gentoo, gestion orientée code source),
    - dans l'embarqué, il peut s'agir de **ipk**, compatible deb ou parfois de **rpm** ;
- Dans l'embarqué :
  - constitue le système (« racine », « rootfs ») ;
  - base de la création du firmware.

LES DISTRIBUTIONS

- Pourquoi ?
  - dépendances applications/bibliothèques, bibliothèques/bibliothèques, parfois applications/applications
  - notion de « paquets » : application et/ou bibliothèques et (potentiellement) fichiers de configuration
  - mise à jour = évolution du système
  - mises à jour de sécurité !
- Construction du système (système de compilation / construction de paquets)
- Patches... Rapidement ingérable « à la main »

DEUX MODÈLES

- *communautaire*
  - projets libres, animés par une communauté (amateurs éclairés et professionnels, parfois fondations ad-hoc)
  - **OpenEmbedded-Core / Yocto**, **Ångström**, **Buildroot**, **OpenWRT**, etc.
  - **Linaro** : à travers une fondation chapeautée par ARM
  - **Android** (cas particulier : Google) ; **Tizen** (Linux Foundation), **FirefoxOS** (Mozilla Foundation)
  - autres distrib' Linux « classiques » réadaptées : **emdebian**, **Ubuntu Core**, **Fedora ARM**, etc. *Attention : Distributions binaires*
- *commercial*
  - sur étagère (COTS), adaptable (service vendu)
  - **WindRiver**, **Sysgo**, **MontaVista**, etc.
- articulations
  - commercial s'appuie quasiment toujours sur du libre communautaire
  - relations commercial/communautaire (participation, intégration, etc.)
  - groupements d'industriels : OHA, Genivi, Babylone

DISTRIBUTIONS SOURCES vs BINAIRES

- Binaires
  - rapide, tout est déjà compilé (croisé)
  - idéal pour les POC
- mais manque de maîtrise
  - top-down
  - ajout de paquets seulement « à chaud » (lorsque le système fonctionne sur la cible)
- concerne : les distributions « classiques » réadaptées (Ubuntu, etc.)
- BSP pour embarqué > 80 Mo (souvent ~250 Mo), Ubuntu/ Fedora/etc : 1-2Go

- Sources
  - lent à mettre en place, tout doit être compilé, après une phase de configuration
  - pour la mise en production : maitrise totale de la génération
    - bottom-up
    - ajout des paquets « à froid » sur l'image du firmware à intégrer (toujours possibilité de le faire « à chaud »)
- Buildroot, OE-Core/Yocto, OpenWRT, T2, OpenBricks
- à partir de 2 Mo

- Hybrides
  - très bon compromis : distributions sources pré-compilées
  - permet la rapidité du binaire et la flexibilité des sources
  - point de départ du développement de sa propre configuration de distribution sources
  - par exemple : Angström, OpenWRT
  - de 2,5 Mo à 250 Mo

ENVIRONNEMENTS PRÊTS À L'EMPLOI

- idée : simplifier la création d'un système Linux embarqué
- télécharge les sources à jour, les compile (croisé)
- gère les problématiques de compatibilité, de licences, de création de firmware (limité), etc.
- le projet **Linaro** pour ARM : compilateur, débugger, compatibilité noyau (gestion de puissance), etc.

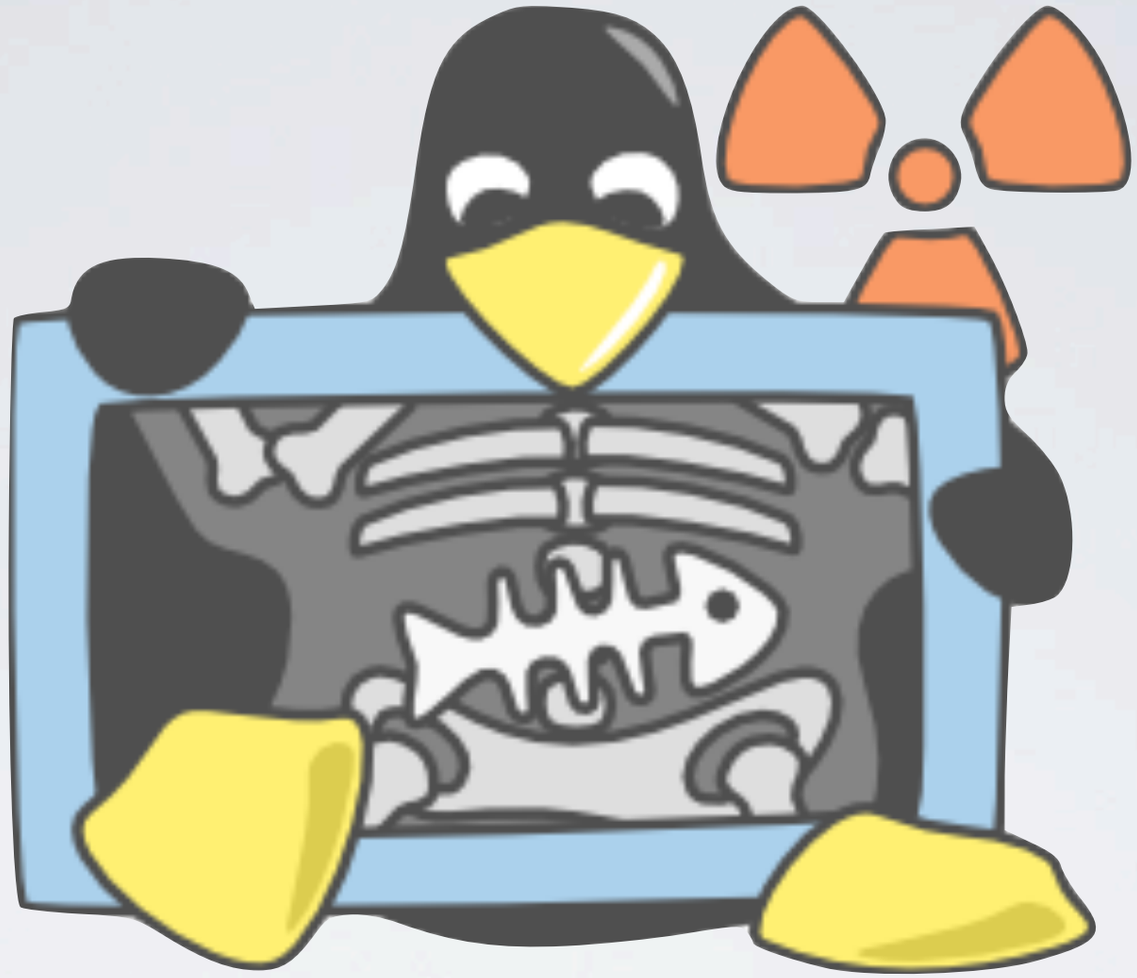
- **Buildroot**
  - fichiers Makefile en arborescence
  - rapide et efficace mais limité par la simplification
  - peut coincer (implique alors de toute refaire sous OE)
- **OpenEmbedded / OE-Core (Yocto Project)**
  - arborescence plus complexe
  - assez long à mettre en place mais très puissant
  - pérenne sur le long terme
  - migrations de projets entiers sous Yocto
- **OpenWRT**
  - spécialisé pour les routeurs et appareils réseau
  - utilise Buildroot-NG
  - très grande quantité de paquets

TP : BUILDROOT

- Télécharger tarball Buildroot
- Installer compilateur + qemu
- Décompresser et se placer dans le répertoire Buildroot
  - $ make qemu_x86_defconfig
  - $ make (~30 minutes)
- Tester le système :
  $ qemu-system-i386... (cf « board/qemu/x86/readme.txt »)

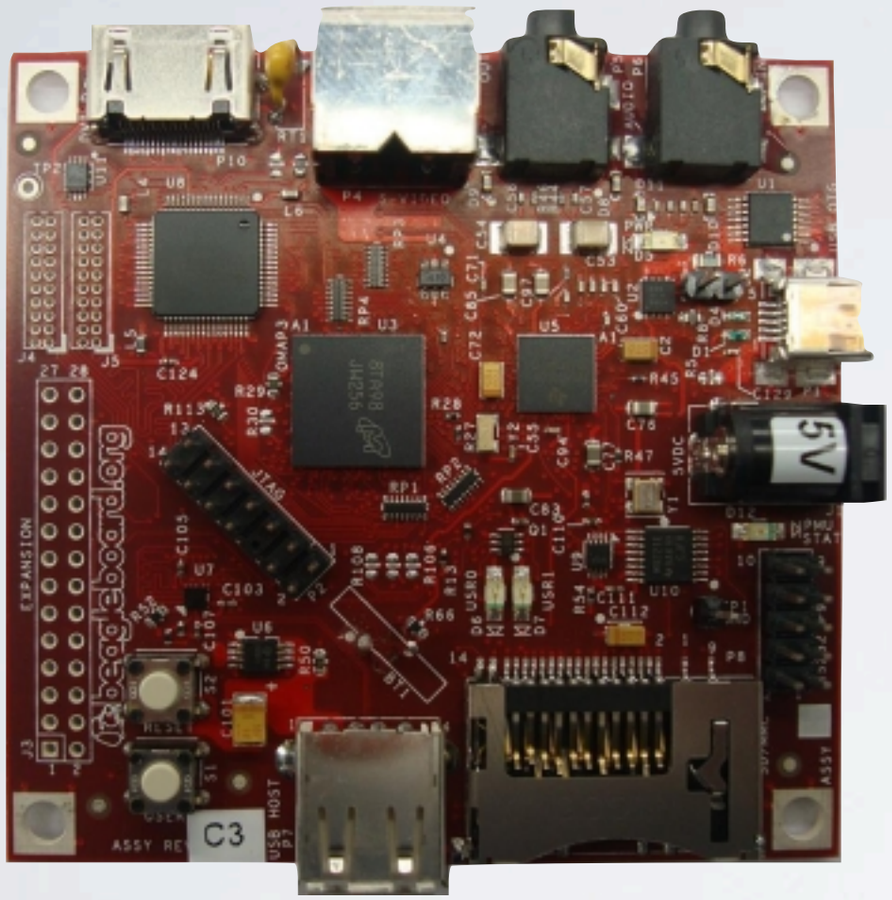
ENVIRONNEMENTS PRÉINTÉGRÉS/GRAPHIQUES

- But : base déjà existante, rajouter seulement sa valeur ajoutée
- **Ångström** : distribution complète, modulable par génération en ligne (Narcissus), très utilisée pour les POC (à présent migré sous Yocto)
- **Android** : le grand gagnant, mais environnement complet tout Java (sur noyau Linux) incompatible avec une distribution Linux « standard », liant pour l'avenir (apparition cependant de systèmes hybrides)
- **MeeGo** : base Linux standard, écosystème cohérent, porté par de grands industriels avant d'être abandonné brutalement au profit de Tizen, rebooté dans un nouveau projet (**Mer**)
- **Tizen** : peu de recul encore, v2.4 vient de sortir ; utilise HTML5 et EFL
- Gnome Mobile (**Maemo**, **OpenMoko**, **Sugar**) : solutions plus ou moins mourantes
- **GPE/OPIE** (handelds) : anciennes solutions abandonnées mais toujours utilisées sur de petits projets
- **Sato** (**Yocto/Poky**) : géré par la Linux Foundation, généré par le moteur de compilation OE-Core
- **Qt4 embedded/Qt5** & **QML** : bibliothèque graphique, très dynamique, mais pas vraiment un environnement

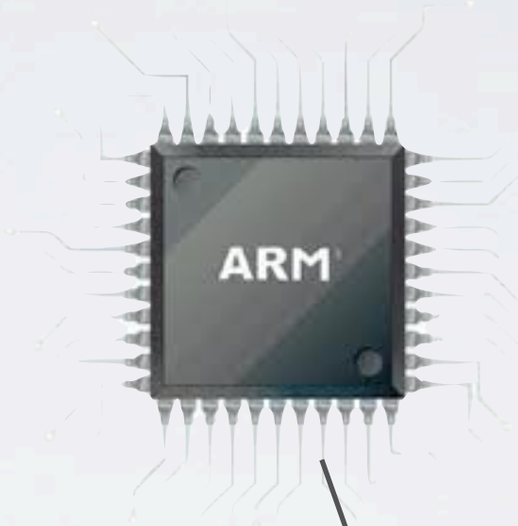


QU'EST-CE QUE LINUX EMBARQUÉ ?

ARCHITECTURE MATÉRIELLE



CPU

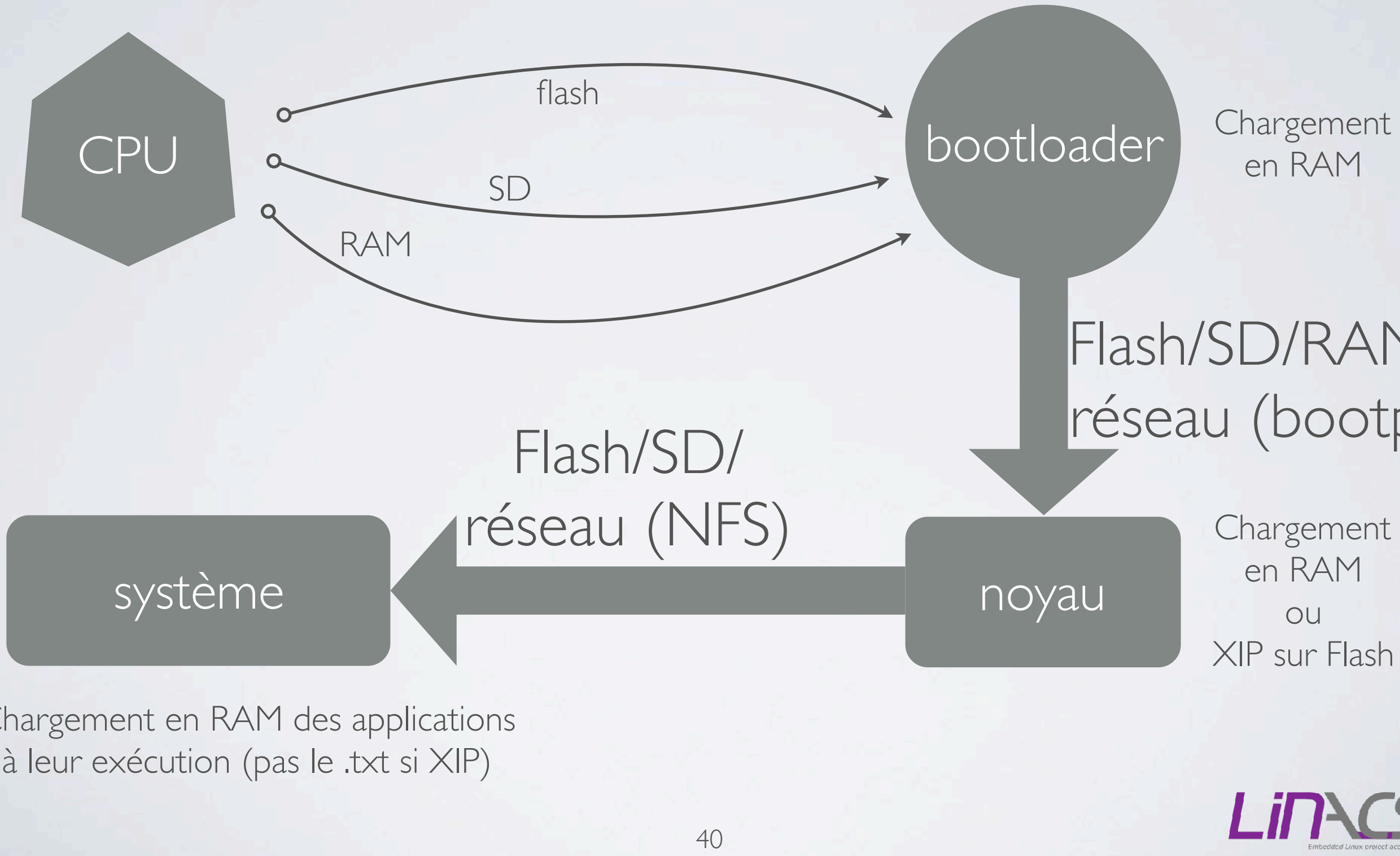


RAM

SD/MMC

Flash

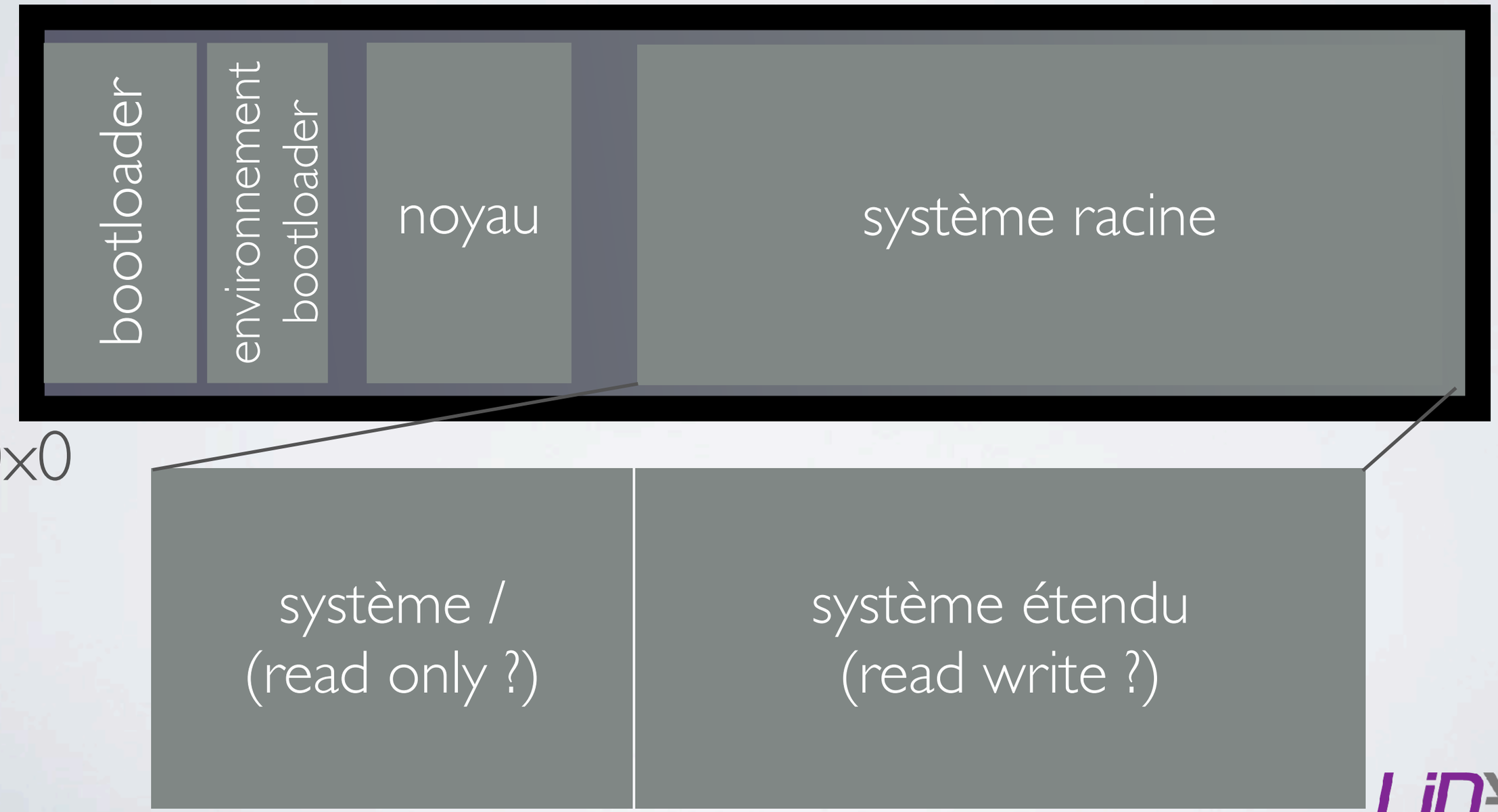
DÉMARRAGE


CPU
flash
SD
RAM
bootloader
Chargement
en RAM
Flash/SD/RAM/
réseau (bootp)
Flash/SD/
réseau (NFS)
système
noyau
Chargement
en RAM
ou
XIP sur Flash
Chargement en RAM des applications
à leur exécution (pas le .txt si XIP)

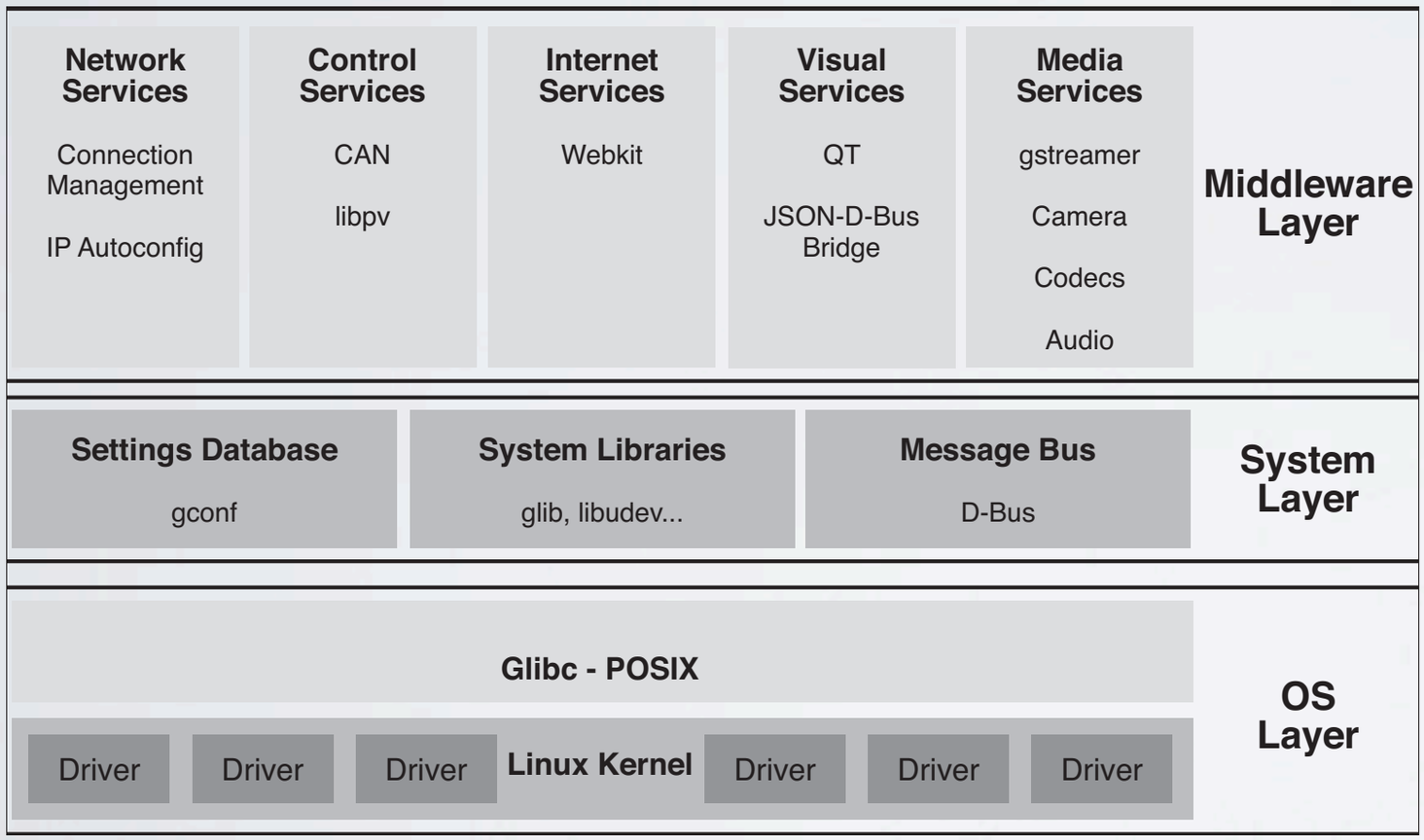
ÉLÉMENTS D'UN PROJET LINUX EMBARQUÉ

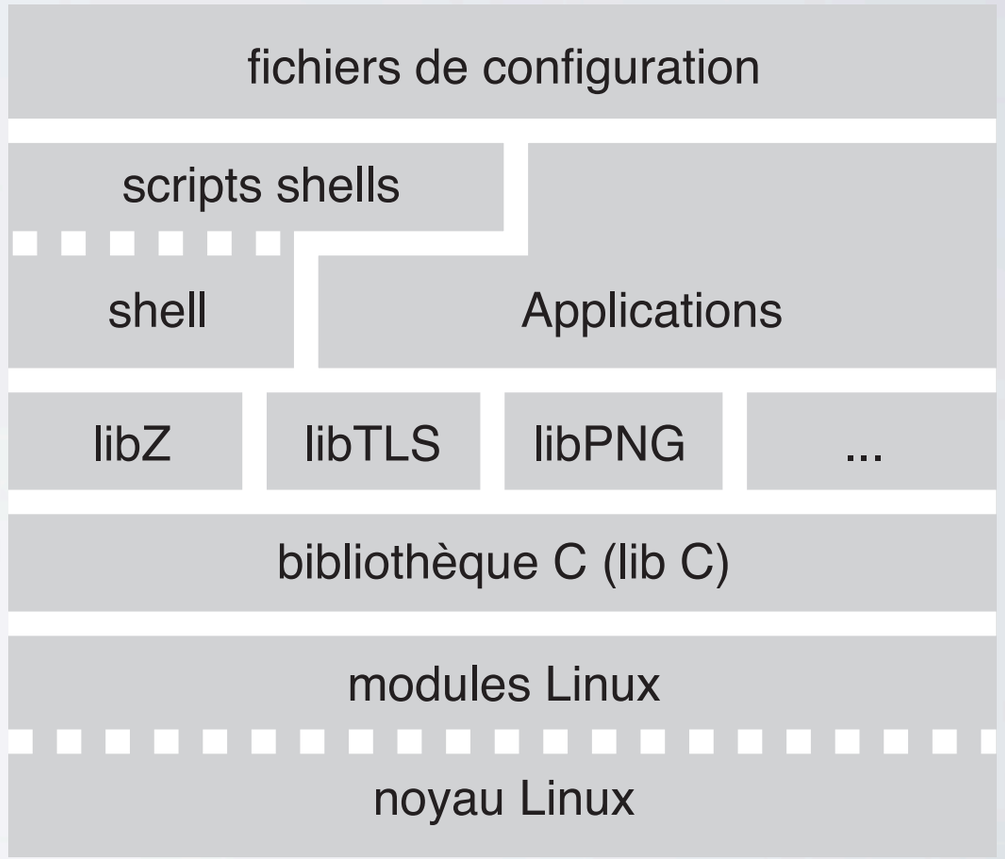
- **Bootloader**
  - Initialisation du hard, démarrage du kernel et chargement des mises à jour système.
- **Outils**
  - Tout pour générer et déboguer les programmes de la cible.
- **Linux Kernel**
  - Accès au matériel et services systèmes. Choix d'un kernel « vanilla » ou patché.
- ***Root file system***
  - Contient les utilitaires de base et la configuration, voire les toolkits graphiques etc.
- **Applications**
  - La part visible de l'utilisateur, les fonctions et la personnalisation.
- **Maintenance**
  - Le projet sera-t-il amené à être porté, complété, étendu ?

ORGANISATION PHYSIQUE DU SYSTÈME SUR FLASH



ORGANISATION LOGIQUE DU SYSTÈME GNU/LINUX

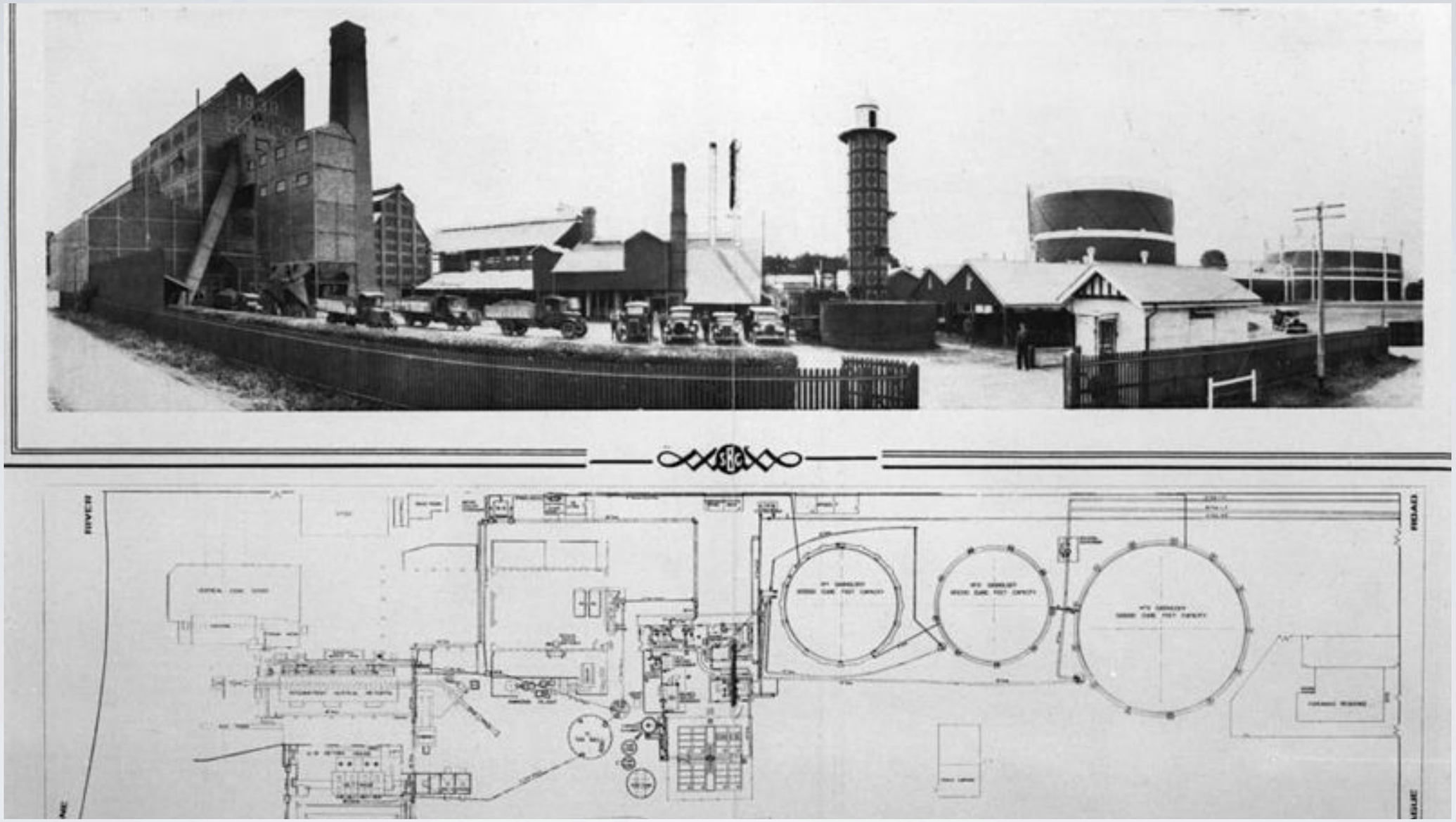




*Vue d'un système Linux embarqué industriel par Pengutronix (***http://www.pengutronix.de/ development/web/index_en.html***).*

MODALITÉS D'ORGANISATION

- sur la flash
  - en lecture seule
    - système racine non modifiable à chaud
    - sécurité accrue
  - en lecture-écriture
    - tout ou partie du système modifiable
    - espace de stockage
- en RAM
  - partie volatile
    - espace de stockage
    - potentiellement une image du système à modifier (par exemple le paramétrage)
- mise en oeuvre via les systèmes de fichiers ; détermine l'organisation du système et les politiques de firmware !



http://commons.wikimedia.org/wiki/
File:StateLibQld_1_54544_Panoramic_view_of_the_Gas_Works_on_Beesley_Street,_South_Brisbane_1935.jpg

RÉALISATION

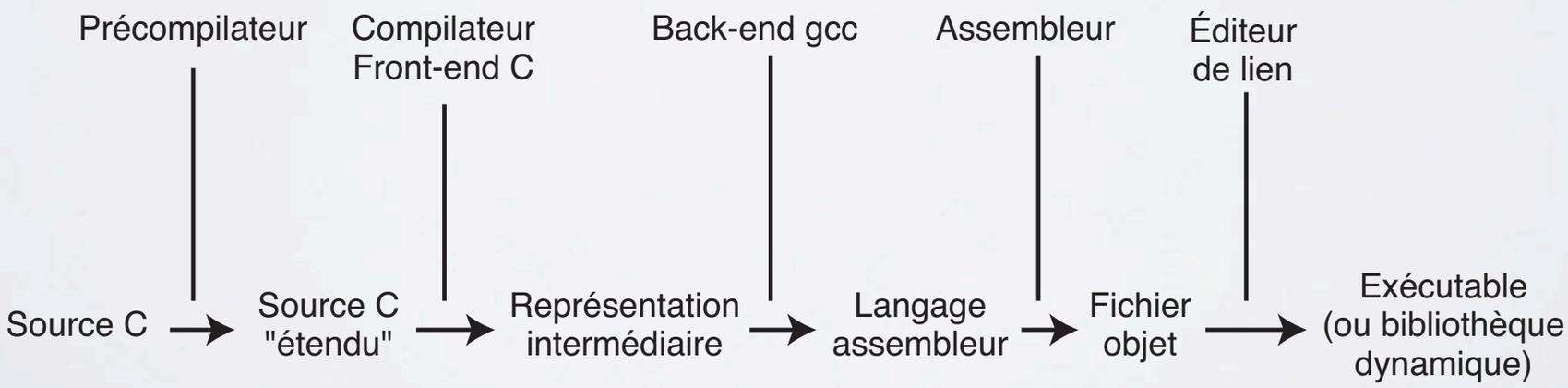
PRÉREQUIS : LA COMPILATION CROISÉE

➡ principe : générer à partir d'un code sur l'hôte un exécutable pour une cible

➡ GCC croisé à construire (à la main ou en automatisé), ou à récupérer

• Compilateur (rappel) :

- Préprocesseur
- Compilateur « proprement dit »
- Assembleur
- Éditeur de liens



GÉNÉRATION D'UN COMPILATEUR CROISÉ GNU

- Packages nécessaires :
  - GNU binutils (readelf, objdump, objcopy, strings, etc)
  - GCC
- Préparation
  - Déterminer et préparer un répertoire pour l'installation de la chaine (ex: /opt/gcc-arch)
  - Préparer un répertoire pour la compilation de la chaine (il est déconseillé de recompiler gcc dans les sources)
- Méthode
  - Générer les binutils croisés :
  - make clean && ./configure --prefix=/opt/gcc-arch --target=arch-unknown-linux
  - make && make install
  - Placer les binutils croisés dans le $PATH
  - Générer gcc croisé

COMPILATEUR CROISÉ : CROSSTOOL-NG

- L'outil (script) crosstool-ng permet la génération automatique à partir d'une configuration (version de compilateur et d'outils) de la chaîne
  - Par défaut, recherche par wget
  - Configuration pour l'architecture croisée
- Il permet en outre de poursuivre par une libc (complète ou légère µClibc)
- Capable aussi de compiler un noyau Linux
- (plus puissant que crosstool, abandonné)

COMPILATEUR CROISÉ : RÉCUPÉRATION

- Le plus simple (et conseillé) est de récupérer (téléchargement ou compilation automatisée) un compilateur croisé :
  - dans le SDK fourni avec la carte de dev (binaire et sources)
  - via l'outil de génération de la distribution (exemple : dans Yocto, la compilation automatisée du compilateur croisé est assurée après la compilation d'un autre compilateur natif certifié/testé/supporté)
  - en téléchargement sur CodeSourcery/Mentor (binaire)
  - en téléchargement sur le projet Linaro (sources & binaires pour ARM)

PROGRAMMATION C/SYSTÈME SOUS LINUX & TEMPS-RÉEL

petite incise

MINI-SOMMAIRE PROGRAMMATION C/SYSTÈME

- Les processus Linux
- Les outils de développement
- Les appels systèmes classiques
- La gestion des threads sous Linux
- Communiquer sur le réseau
- Gestion de la mémoire
- Gestion des bibliothèques
- Construire des applications sous Linux
- La sécurité des applications

LINACS
Embedded Linux project accelerator

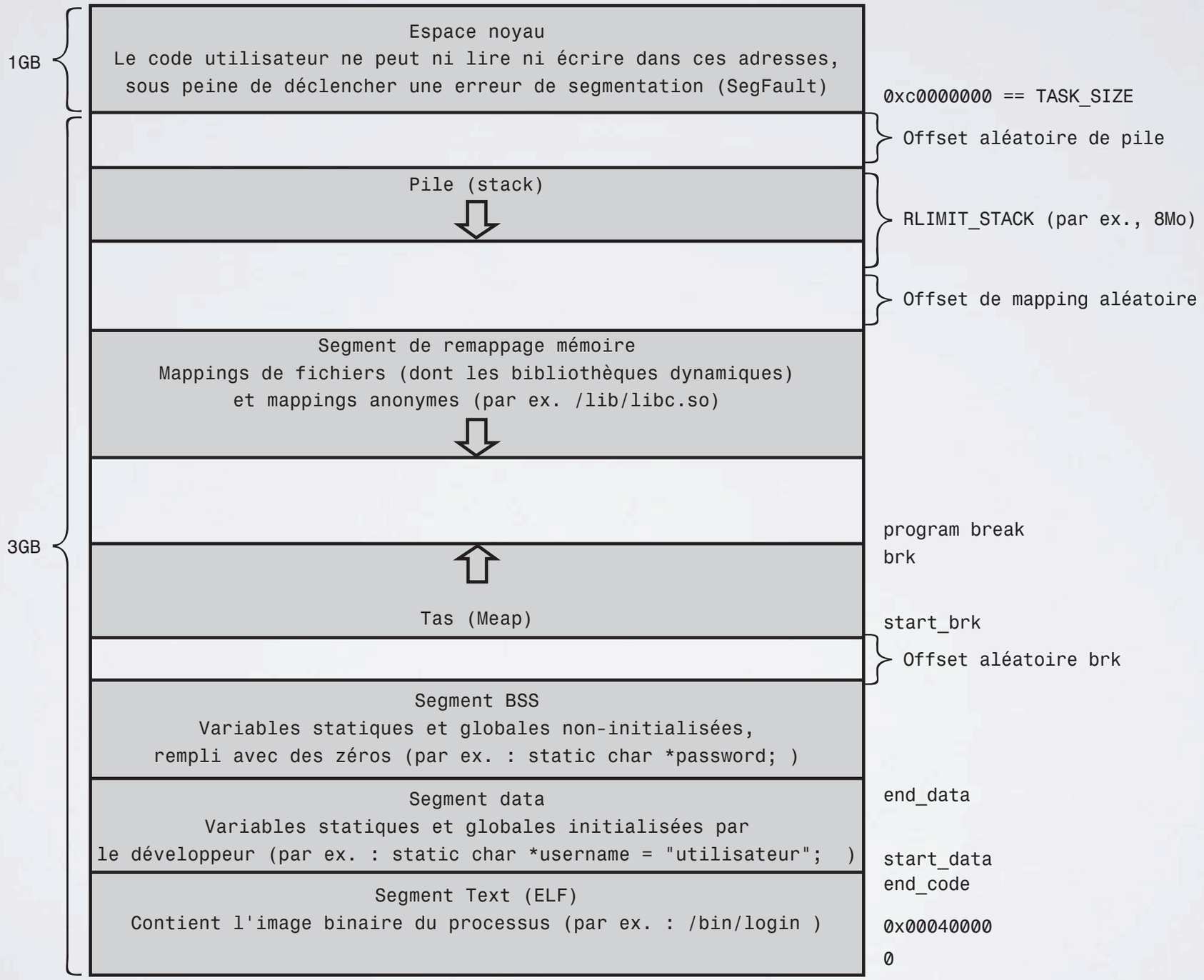
LES PROCESSUS LINUX (I)

- Un processus est un fil d'exécution
- L'exécution d'une application crée un processus
- Un processus peut comporter plusieurs threads (ou fils d'exécution), partageant le même tas mais ayant des piles différentes
- Sous Linux, le format d'une application binaire est ELF32
- lancement par /lib/ld-linux.so.2 (/lib/ld-linux.so.3 sous ARM EABI)
- L'exécutable est relocalisé en mémoire virtuelle aux adresses basses
- La pile est en adresses hautes(elle « descend »), le tas est au milieu,ainsi que les remapping de bibliothèques.
- La mémoire virtuelle est de 4Go
- http://duartes.org/gustavo/blog/category/linux

MÉMOIRE VIRTUELLE & MMU

- notion de mémoire réelle et de mémoire virtuelle, à travers un composant matériel du CPU, la Memory Management Unit
- un microcontrôleur (PIC, Coldfire, ARM M0...) n'a pas de MMU :
  - adressage direct en mémoire réelle
  - pas de protection (un programme peut en faire planter un autre, voire tout le système)
- un CPU de type x86, ARM Cortex, etc. a une MMU :
  - adressage en mémoire virtuelle, chaque processus est séparé
  - problématique de « translation » des adresses (tables/caches matériels et logiciels)
  - espaces protégés : un programme qui plante n'a pas d'impact pour les autres (bac à sable)

EXEMPLE : PROCESSUS LINUX



LES PROCESSUS LINUX (2)

- Les processus reposent sur des bibliothèques partagées ou dites dynamiques
- Une « shared library » est un ensemble de fonctions pouvant être appelées par un processus externe, et réunies sous la forme de fichiers en « .so » (cf /lib et /usr/lib)
- Elles ne sont chargées qu'une seule fois pour tous les processus
- Elles sont extrêmement nombreuses, leurs réutilisations par plusieurs applications sont à la base du concept de « non-réinvention de roue »
- Attention aux licences GPL/LPGL (quasiment toujours en LGPL : pas de restriction d'utilisation dynamique)
- On compte :
  - Au moins la libC (/lib/libc.so.6)
  - libxml, libjpg, libz, libncurses, etc

LES OUTILS DE DÉVELOPPEMENT (1)

- Compilation et édition de liens
- Utilisation de GCC
  - gcc <options> <fichiers> -o <binaire>
    - -L<chemin> : répertoire où trouver les bibliothèques dynamiques
    - -l<biblio> : nom de la bibliothèque à lier (sans le préfixe « lib » : -lthelib vers libthelib.so)
    - -o bin : résultat de la compilation
    - -c obj : résultat intermédiaire (fichier « .o »)
    - -O : optimisation
    - -DMACRO[=<value>] : définition de macro
    - -shared, -fPIC : création de bibliothèque dynamique
    - -static: utilisation de bibliothèque statique
    - -f<option> : de nombreuses options disponibles
  - Binutils : nm, ldd, etc

LES OUTILS DE DÉVELOPPEMENT (2)

- Débuggage
- Programme gdb
  - Utilise ptrace sur processus, ou analyse post-mortem sur coredump
  - En ligne de commande ou graphique
  - Commandes (raccourci) :
    - run (r)
    - break (b)
    - list (l)
    - step (s) / next (n) / cont (c)
    - print (p)
    - backtrace (bt)
    - quit (q)
- Les profileurs : gprof, strace, callgrind (kcachegrind)
- Mise en forme : lcint / indent

LES APPELS SYSTÈME CLASSIQUES (1)

- Conformité de Linux à posix.1
  - Read, write, etc
- Gestion des processus :
  - Chaque nouveau processus est issu d'une duplication du père suivi d'une exécution remplaçant son code
  - fork et exec
- Bonne terminaison de programme :
  - Éviter les zombies
  - wait et waitpid
- Gestion de l'environnement :
  - Environnement propre à chaque processus, hérité du père
  - getenv
  - setenv

LES APPELS SYSTÈME CLASSIQUES (2)

- Les Inter Process Communications (IPC)
  - Mailbox : msgget(), msgsnd(), msgrcv()
  - Mémoire partagée : shmget(), shmat(), shmdt()
  - Sémaphore : semget(), semop()
- Les pipes classiques et les pipes nommés (named pipe)
  - Classique : pipe
    - Tube de communication entre deux processus (en shell : |)
    - La sortie standard du premier est redirigé dans l'entrée du tube (créé par le père ou par lui-même), la sortie du tube est « branchée » à l'entrée standard du second (processus fils ou processus « frère » du précédent)
  - Nommé : mkfifo/mknod
    - Fichier spécial géré par le noyau (créé sur le FS)
    - Un process écrit, un autre lit (vide le tube des données lues)

LA GESTION DES THREADS SOUS LINUX (1)

- Conformité de Linux à posix 1.c
- Principe et implémentation :
  - Partage du tas mais fils d'exécution distincts au sein du même processus
  - Sous Linux : pthreads (#include "pthread.h")
- Avantages et inconvénients par rapport au fork :
  - Plus léger, accès aux ressources communes simplifiées
  - Problèmes de synchronisation, d'accès aux ressources
- Création de thread :
  - int pthread_create(pthread_t *thread, pthread_attr_t *attributs,void * (*fonction) (void * argument), void * argument);
  - Crée sur la référence d'une variable thread, avec quelques attributs, un nouveau thread en appelant une fonction acceptant un argument en pointeur (peut être NULL)

LA GESTION DES THREADS SOUS LINUX (2)

- Vie du thread :
  - pthread_exit() : sortir du thread (mais pas de l'application)
  - pthread_join() : attendre le retour d'un thread
- Les différents attributs
  - pthread_attr_init(pthread_attr_t * attributs)
  - phtread_attr_getXXX() / phtread_attr_setXXX() : adresses, politiques d'ordonnancement, etc ; puis passage à pthread_create()
- Gestion de l'exclusion mutuelle (mutex)
  - Statique: pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
  - Dynamique: pthread_mutex_init(pthread_mutex_t * mutex,
    const pthread_mutexattr_t * attributs)
  - pthread_mutex_lock, unlock, trylock
  - Même fonctions en « rwlock » au lieu de « mutex » : gère les accès en lecture/écriture (plusieurs lecteurs, un seul écrivain)

LA GESTION DES THREADS SOUS LINUX (3)

- Gestion des données et des signaux des threads
  - Possibilité de partager des données privées (créé dans un thread donné) à travers les threads par un système de clés :
    - pthread_key_create() / pthread_key_delete()
    - pthread_setspecific() pour associer une clé à un pointeur
    - pthread_getspecific() pour récupérer les données associées à la clé
- Signaux :
  - Déconseillé mais parfois inévitable
  - pthread_sigmask() / pthread_kill()

COMMUNIQUER SUR LE RÉSEAU (1)

- Les appels système fondamentaux
  - int socket(int domaine, int type, int protocole);
  - int bind(int sock, struct sockaddr * adresse, socklen_t longueur);
  - int listen(int sock, int nb_en_attente);
  - int accept(int sock, struct sockaddr * adresse, socklen_t * longueur);
  - int connect(int sock, struct sockaddr * adresse, socklen_t longueur);
  - int shutdown(int sock, int mode);
- Les sockets DGRAM : implémentation d'un client-serveur en UDP (mode non connecté)
  - recvfrom/sendto ou recv/send
- Les sockets STREAM : implémentation d'un client-serveur en TCP (connecté)

COMMUNIQUER SUR LE RÉSEAU (2)

- Les options importantes
  - getsockopt/setsockopt
    - Broadcast, keepalive, reuseaddr, etc
- Utiliser les services d'inetd
  - Transformation du serveur en démon
  - Intégration au système

GESTION DE LA MÉMOIRE (I)

- Rappel sur les variantes de malloc
  - Allocation de mémoire dans le tas
  - malloc(), basé sur brk(), sbrk() : algo de réservation de mémoire, retourne un pointeur
  - calloc() : allocation avec pré-remplissage
  - realloc() : ré-allocation (modification de la taille de la mémoire réservée)
  - free() : libération de la mémoire pointée
  - Alloca() : allocation dynamique sur la pile
- Configuration de l'algorithme de malloc : mallopt()
- Suivi avec mtrace
  - mtrace(), muntrace() et application mtrace
  - $ export MALLOC_TRACE="trace.out"
- Verrouillage de pages mémoire (protection contre swap) : m[un]lock[all]

GESTION DE LA MÉMOIRE (2)

- Electricfence : bibliothèque de gestion de mémoire pour meilleure détection des fuites
- Gestion des fuites mémoire et outils associés
  - export MALLOC_CHECK_=1
  - Valgrind
- Routines avancées de traitement des blocs mémoire
  - Projection : mmap
  - Copie de blocs : memcpy(), mempcpy(), memccpy()
  - Autres : memmv(), memcmp()

GESTION DE BIBLIOTHÈQUES

- Bibliothèques statique
  - Fichier « .a », collection (archive ar) d'objets « .o », lié à l'intérieur du programme lors de la compilation
  - Utilisation :
    gcc -c code.c -o code.o
    ar cr libcode code
    gcc -static main.c -L. -lcode -o programme
- Bibliothèque partagée/dynamique
  - fichier « .so », symboles (fonctions et variables globales)
  - Utilisation :
    gcc -shared -fPIC -o libcode.so code.c
    gcc main.c -o programme -L. -lcode

CONSTRUIRE DES APPLICATIONS POUR LINUX

- Gestion des dépendances
  - automake/autoconf
  - cmake
- Construction
  - http://fr.wikipedia.org/wiki/Make
  - $@, $?, $<, etc : http://ljk.imag.fr/membres/Jean-Guillaume.Dumas/Enseignements/CPP_M1MAI/gnu_make.pdf

SÉCURITÉ

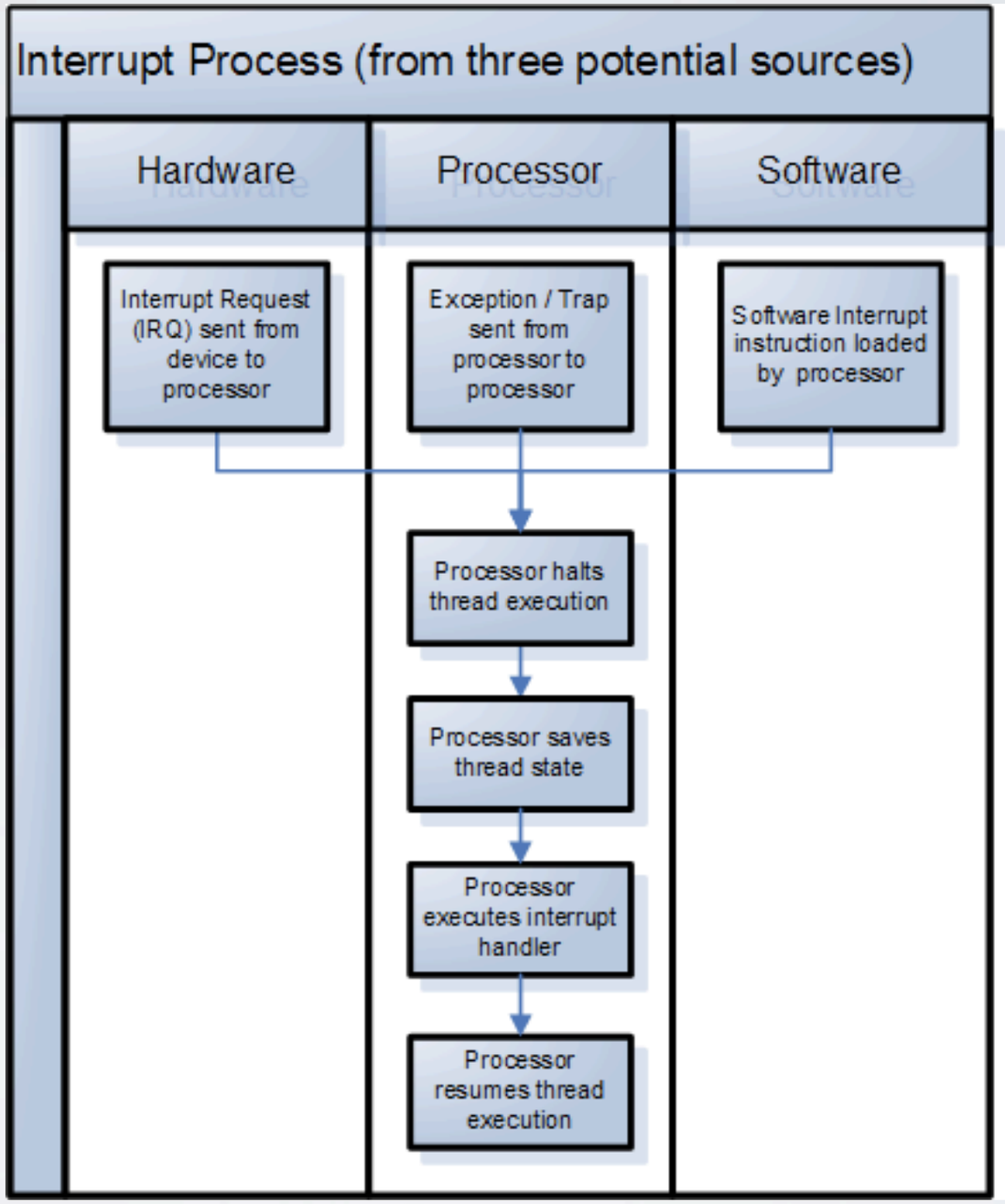
- Bonnes pratiques de codage
  - Buffer overflow et shellcode
    - Tableaux sur la pile
    - Débordement par injection
    - Shellcode : exécution d'un shell à un retour de fonction (injection)
- Faire attention au nombre de caractères lus
- Gets (« N'utilisez jamais gets() »)
- Scanf : très attention aussi (éviter)
- Protection de pile
  - -fstack-protector[-all] : canaris mis en place par GCC (lier -lspp)
  - Dans le noyau (PAX)
- Protection noyau : chroot, ulimit, droits, etc (blindage par AppArmor, SELinux, ou encore : RSBAC)

TEMPS RÉEL

INTERRUPTIONS, LATENCE

- deux types d'interruptions :
  - matérielle (IRQ, FIQ, processeur) : un périphérique alerte le CPU qu'il se passe quelque chose (et qu'il faut venir lire son buffer de données — *device interrupt*) ou le processeur lui-même a rencontré une erreur (accès mémoire interdit, division par zéro, etc. — *hardware trap*)
  - logicielle (via une interruption CPU dédiée) : « appel système », d'une routine bas niveau (lecture, écriture, mapping mémoire...) par un programme
- latence
  - temps de traitement entre le moment de la réception de l'interruption (niveau matériel), sa détection par le noyau (à travers un système de masque puis de handler de traitement), la remontée de l'information au driver/programme concerné, et enfin le traitement adéquat attendu
  - sur un système, trop de latence donne l'impression de « lag » (réactions qui arrivent trop tard, ex : lecture audio qui saccade)
  - système temps réel si la latence est toujours (de manière déterministe) inférieure au temps attendu de traitement (échéance)

INTERRUPT SOURCES AND PROCESSOR HANDLING



Stephen Charles Thompson
http://en.wikipedia.org/wiki/File:Interrupt_Process.PNG

LE TEMPS-RÉEL

- réponse aux évènements en un temps strictement déterminé
  - délai de latence ou temps de réponse de chaque tâche inférieur à son échéance — ou sa période si la tâche est périodique
- temps-réel « mou » : pas de garanti totale du déterminisme du système, comportement observé usuellement (mais si charge excessive ou évènements inattendus — ex : stress du système par des accès importants à un disque dur —, le temps de réponse peut varier)
- temps-réel « dur » : garantie stricte de conformité à des modèles comportementaux prédéfinis (cependant, les problématiques sont nombreuses et le système n'est pas forcément plus stable au final !!)
- pas de notion de « temps court »
  - un système bancaire est temps-réel sur une journée !
  - mais généralement, le temps de latence attendu d'un noyau temps réel (type VxWorks) est de l'ordre de 10 µs (en dessous, FPGA/ASIC obligatoire)

MULTI-TÂCHES

- problématique principale des noyaux : faire tourner plusieurs tâches « simultanément »
- par « tâche », on entend processus et thread (dans le cadre d'un processus multi-threadé)
- pipe de tâches (par coeur du CPU) : découpage du flux d'exécution des tâches, quelques instructions exécutées par tâche, les unes après les autres
- détermination de l'ordre d'exécution de ces bouts de tâches = ordonnancement
- donne l'impression de simultanéité (exemple de la lecture vidéo : image et son « en même temps » : en réalité, séquencé de manière assez rapide pour en donner l'impression, càd 25 images/s & 48kHz), ou garantie de temps-réel (ne pas confondre !)

ORDONNANCEMENT

- l'ordonnanceur/scheduler est une partie logicielle du noyau qui calcule par un ou plusieurs algorithmes l'ordre des tâches à exécuter
- cet algorithme a lui-même une complexité qui dépend souvent du nombre de tâches à gérer (par exemple : O(n), O(log(n))), ou pas (O(1)). De cela dépend la capacité de l'OS à gérer correctement la charge et sa stabilité générale.
- l'ordonnanceur est déclenché par une interruption d'horloge (fréquence de réordonnancement : le « tick »), et/ou par les interruptions du système lui-même (système « tickless »)
- un système qui doit attendre qu'une tâche rende la main (yield) est dit coopératif (très dangereux : en cas de plantage, plus rien ne répond)
- un système qui peut interrompre une tâche à tout moment est dit préemptif
  - une tâche qui ne souhaite pas se voir interrompre pendant une phase critique (notion d'atomicité) peut user des sémaphores
  - un noyau qui peut interrompre ses propres tâches internes (traitement d'interruptions et d'appels système) est dit totalement préemptif (très rare : Solaris, Linux)

POLITIQUES D'ORDONNANCEMENT

- temps partagé (pour desktop, dynamique) : prise en compte des interruptions afin de donner une impression de fluidité à l'utilisateur (son qui ne saccade pas, clavier et souris qui répondent immédiatement), au détriment des tâches de calcul ou des accès aux périphériques blocs (écriture sur disque dur)

- temps-réel (« dur ») : ordre d'exécution des tâches calculée en fonction du strict respect des échéances

- temps-réel « mou » : système où l'ordonnancement par temps partagé est assez robuste pour donner l'impression d'un temps-réel (mais aucune garantie intrinsèque qu'une ou plusieurs échéances ne seront pas respectées de temps à autre). Cas du Complete Fair Scheduling (CFS) de Linux.

CHANGEMENTS DE CONTEXTES

- lorsqu'une nouvelle tâche est exécutée (par choix de l'ordonnanceur), le contexte de l'ancien processus en cours est sauvegardé, pour être restauré lorsqu'il sera de nouveau exécuté
  - sauvegarde des registres
  - sauvegarde de la mémoire virtuelle
  - particulièrement coûteux en temps !
- méthode logicielles et matérielles pour simplifier/optimiser ces changements de contexte

PRIORITÉ

- les tâches peuvent avoir différentes priorités
- la politique d'ordonnancement tient compte de ces priorités
- cependant, cela peut produire des déconvenues :
  - famine : si des tâches de haute priorité préemptent sans cesse une tâche de moins haute priorité, celle-ci peut ne jamais être exécutée (solution : héritage de priorité, ie augmentation de la priorité au fur et à mesure de l'attente, jusqu'à devenir très prioritaire)
  - inversion de priorité : mêlé à des problématiques de verrous, une tâche de plus grande priorité peut se retrouver sous une tâche moins prioritaire (cf étude de cas plus loin)

PORTABILITÉ

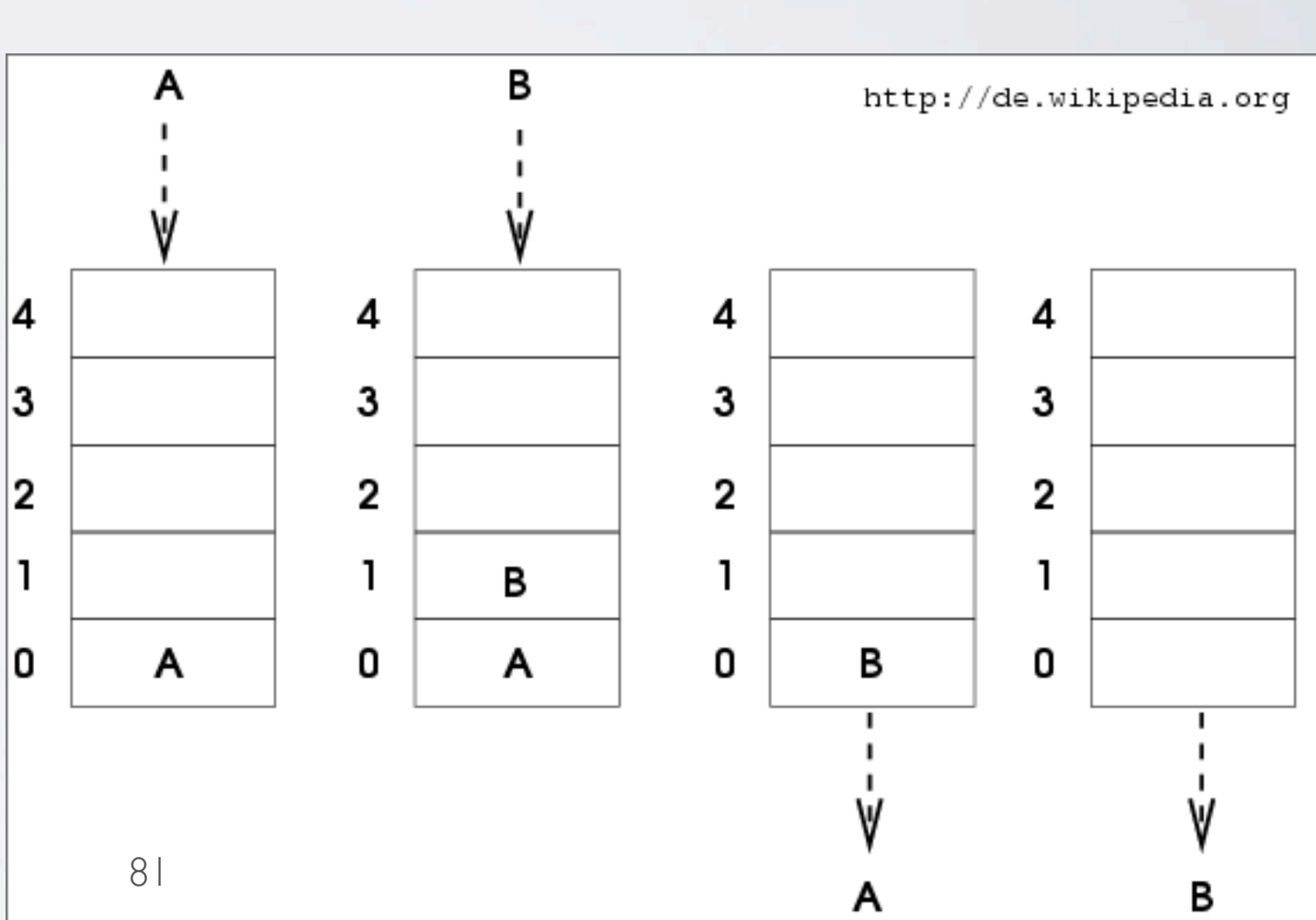
- souvent précaire : chaque noyau tend à avoir sa propre API
- POSIX.1b, Real-time extensions (IEEE Std 1003.1b-1993)
    - Priority Scheduling
    - Real-Time Signals
    - Clocks and Timers
    - Semaphores
    - Message Passing
    - Shared Memory
    - Asynch and Synch I/O
    - Memory Locking Interface

ORDONNANCEMENTS TEMPS-RÉEL

- chaque tâche doit s'exécuter en un certain temps connu d'avance, afin d'effectuer une action qui doit s'achever avant une échéance elle aussi connue (à savoir la fin de la période suivante si la tâche est périodique ; chaque tâche possède sa propre période)
- différents algorithmes (principaux) :
  - FIFO (First in, First Out)
  - Round-Robin (RR)
  - Shortest Job First (SJF)
  - Rate monotonic
  - EDF (Earliest Deadline First)

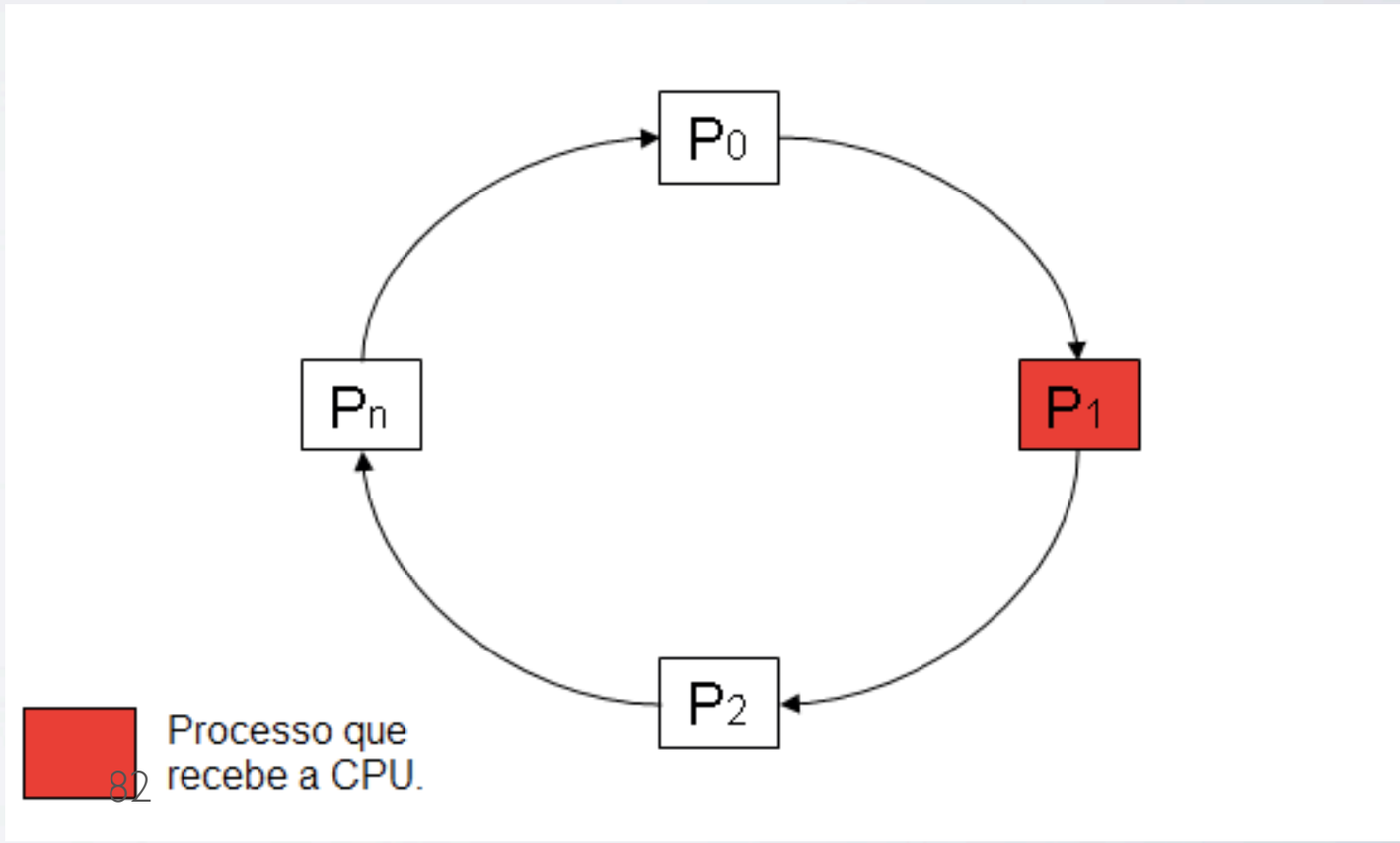
FIFO

- tâches exécutées dans l'ordre d'arrivée (file)
- interruption si :
  - I/O
  - tâche de plus haute priorité
  - yield
- gros inconvénient si petite tâche en attente derrière d'autres plus longues



ROUND-ROBIN

- tourniquet
- comme FIFO mais :
  - quantum de temps
  - préemptivité des tâches
  - grosse amélioration



SHORTEST JOB FIRST

- plus court processus en premier (temps d'exécution total le plus court)
- version préemptive
  - SRTF (Shortest Remaining Time First)
    - si une tâche plus courte que celle en cours est insérée dans la file des processus, elle interrompt la tâche courante et prend sa place
- avantage : algorithme des plus rentables pour la réduction du temps passé dans la file d'attente des processus
- inconvénient : nécessite une évaluation précise du temps d'exécution de tous les processus en attente de traitement
- de fait, peu utilisé

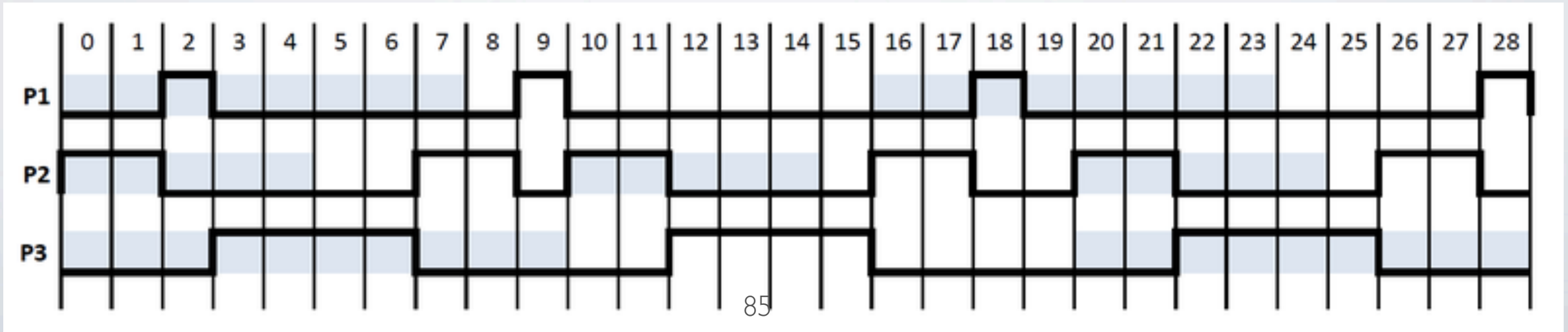
EDF

- échéance proche = préparation en premier
- préemptif, à priorité dynamique
  - plus l'échéance d'une tâche est proche, plus sa priorité est grande
  - au plus vite le travail doit être réalisé, au plus il a de chances d'être exécuté
- le meilleur rendement en théorie
- problèmes
  - difficile à implémenter, il faut savoir combien de temps il reste à chaque tâche
  - peut même être dangereux si occupation du processeur à plus de 100% (imprédictibilité)
  - peu utilisé en pratique

EDF : EXEMPLE

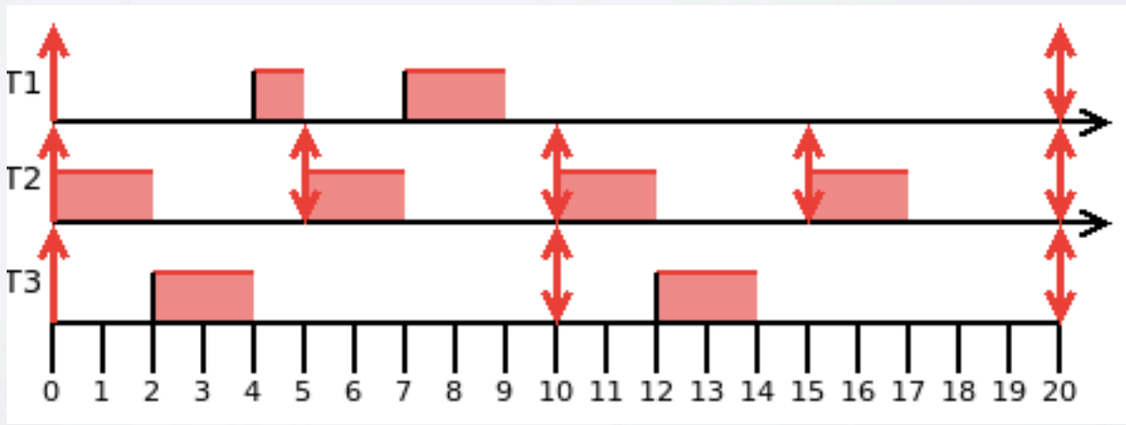
Process Timing Data

Process	Execution Time	Period
P1	1	8
P2	2	5
P3	4	10



RATE MONOTONIC

- ordonnancement à taux monotone
- priorité la plus forte à la tâche qui possède la plus petite période
- optimal dans le cadre d'un système de tâches périodiques, synchrones, indépendantes et à échéance sur requête avec un ordonnanceur préemptif (condition suffisante d'ordonnançabilité = charge processeur à 69,3 %)
- tâche modélisée par quadruplet (r, C, D, T)
  - r = date de réveil (algorithme optimal si toutes tâches simultanées, ie r = 0)
  - C = durée d'exécution ou pire temps d'exécution de la tâche
  - D = échéance relative de la tâche (à échéance sur requête, D = T)
  - T = Période
- Exemple :
  - Tp1 = (C = 3 , T = 20)
  - Tp2 = (C = 2 , T = 5)
  - Tp3 = (C = 2 , T = 10)



MUTEX & SÉMAPHORES

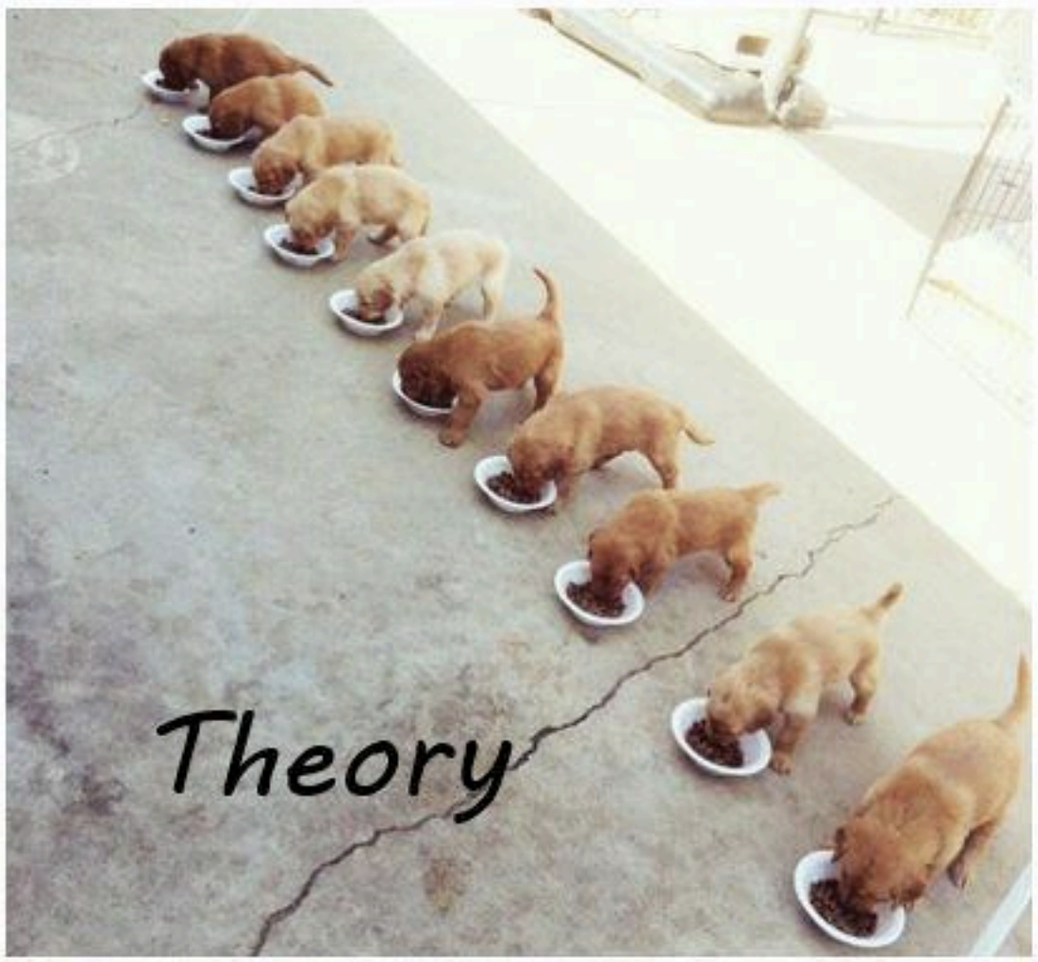
- but : rendre atomique une partie de la tâche logicielle
  - dans un processus : éviter les race conditions qui mènent à des comportements indéterministes et difficilement reproductible, par exemple les accès concurrents à la mémoire (lecture et écriture simultanés)
  - dans le noyau : si une tâche demande à écrire à un endroit du disque dur, et une autre tâche à un autre endroit, mieux vaut ne pas perturber le déplacement de la tête
- implémentation
  - parmi les IPC entre processus
  - posix 1.c pour les threads de processus
  - dans le noyau Linux : les spinlocks
- inconvénients :
  - rend le système moins réactif car nuit (potentiellement) à la préemptivité
  - risque d'interblocages

RACE CONDITION : EXEMPLE

Thread1	Thread2		valeur d'entier
			0
lit la valeur		←	0
incrémente la valeur			0
écrit la valeur		→	1
	lit la valeur	←	1
	incrémente la valeur		1
	écrit la valeur	→	2

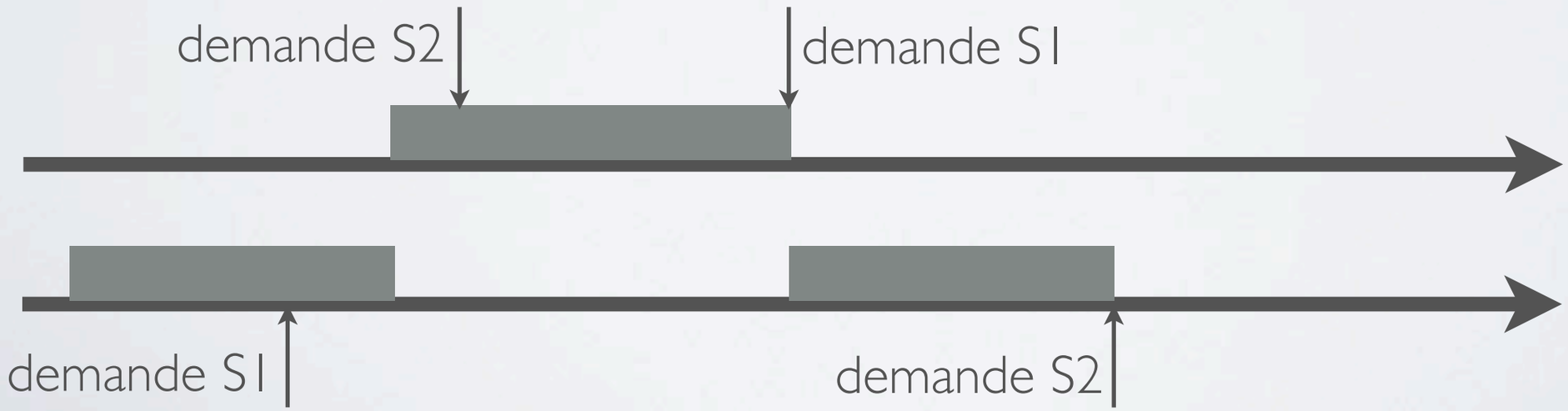
Thread1	Thread2		valeur d'entier
			0
lit la valeur		←	0
	lit la valeur	←	0
incrémente la valeur			0
	incrémente la valeur		0
écrit la valeur		→	1
	écrit la valeur	→	1

Multithreaded programming



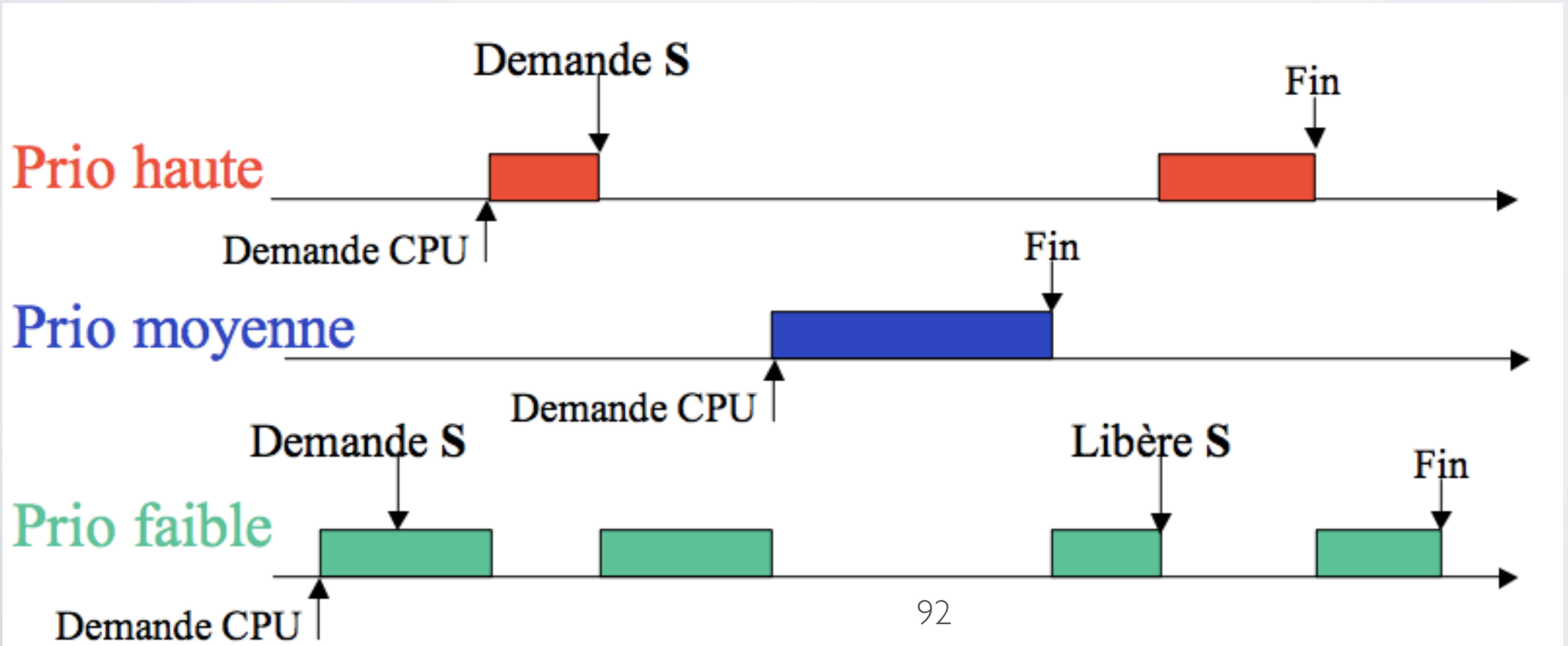
INTERBLOCAGES

- une tâche T1 prend le sémaphore A, s'exécute puis est interrompue
- une seconde tâche T2 prend le sémaphore B, s'exécute puis est interrompue
- T1 attend le sémaphore B et est interrompue
- T2 attend le sémaphore A, qui n'est jamais relâché car T1 attendra indéfiniment le sémaphore B jamais relâché par T2



INVERSION DE PRIORITÉ

- cas où les exclusions mutuelles mènent à rendre prioritaire un processus moins prioritaire qu'un autre



EN BREF...

- le temps-réel est épouvatablement complexe
- implémenter un système temps-réel n'est pas forcément garantir sa robustesse : cf mission pathfinder sur Mars...
- la programmation multithreadée ramène à des problématiques similaires difficiles à résoudre : problème encore plus important avec la généralisation des processeurs à multicoeur
- pour aller plus loin : cours de Pierre-Yves Duval (cppm, 2002) http://www.in2p3.fr/actions/formation/InfoTempsReel/Ordonnancement_TR.pdf
- de plus en plus souvent : temps-réel dur évité le plus possible, ou utilisé pour une tâche que l'on veut déterministe, le reste des tâches du système tournant sur le temps CPU restant (ex : calculateur de bord sous Xenomai/Linux)

fin de l'incise

DÉMARRAGE *BOOTLOADER*

- Initialisation du CPU et de sa vitesse
- Initialisation RAM (démarrage des banques, registres de configuration etc.)
- Activation des caches du processeur
- Initialisation du pointeur de pile
- Initialisation de la zone de paramètres pour Linux (important, car les paramètres de *boot* informent le noyau du système de fichiers '/', de la taille des pages, de la taille de la RAM...)
- POST (Power On Self Test) : autotest du matériel
- Support de la gestion d'énergie (power/resume)
- Initialisation du port série
- Chargement éventuel et branchement au point d'entrée du *kernel*.

DÉMARRAGE (SUITE)

- Linux ré-effectue les initialisations du *bootloader*, à l'exception :
  - des paramètres de *boot* : utilisation de ceux indiqués par le *bootloader*, sauf si d'autres ont été compilés « en dur »,
  - des GPIOs : fonctionnement comme un « masque ».
- Suivant l'option de démarrage (initrd/initramfs ou non) le noyau est chargé seul ou avec une image de disque
  - Dans le cas initramfs, le noyau monte ce ramdisk en '/' et exécute /init
  - Sinon, le paramètre « root= » est utilisé et le système de fichiers pointé est monté sur / (en lecture seule/*read-only*, sauf si paramètre « rw » indiqué)

DÉMARRAGE (FIN)

- Le programme pointé par le paramètre « init= » (par défaut /sbin/init) est démarré :
  - il suit le script /etc/inittab
  - il est configuré (en conf system V) en *runlevel* (1 – single user, 2 – multiuser, 3 – net, 5 – X11)
  - il lance le script pointé, dans /etc/inittab, par la ligne commençant par 'si:' (ou 'sysinit:')
  - ce script parcourt généralement dans /etc/rc*.d (en passant le paramètre « start »). Les *runlevels* décident des services (*daemons*) à démarrer.
- Usuellement, on démarre des « getty » sur les terminaux
  - Mais on peut démarrer des binaires spécifiques (voir le paramètre d'inittab « respawn »)

LE BOOTLOADER U-BOOT

- Anciennement PPC-Boot ; licence GPL
- Support d'une grande variété d'architectures différentes dont PowerPC et ARM (SH, MIPS, mais pas x86)
- Configurable, avant compilation (architecture, commandes disponibles, etc)
- Bootloader avec commandes :
  - Configuration Ethernet (gère aussi le ping, le TFTP, le changement d'adresse MAC)
  - Commandes Flash
    - effacement, copie
    - protection
    - stockage de variables d'environnement
- Permet de scripter le démarrage
- Le plus largement utilisé à l'heure actuelle, très bien documenté, très puissant
- exploration de U-Boot

LE NOYAU

- Sources de référence (« vanilla kernel »)
  - www.kernel.org
  - La dernière version embarque le maximum de drivers ; c'est la version de référence pour reverser
    Version stable actuelle : 4.9 (décembre 2016)
- Configuration, compilation en compilation croisée
  - **Très** nombreuses options de configuration (~6600 !!)
  - Garder en référence le fichier généré (`.config`)
    `make newconfig` : raccourci pour le cas d'un changement de kernel
    `make <board>_config` : chargement d'une configuration par défaut pour une carte
- Compilation par :
  `make ARCH=<architectecture> CROSS_COMPILE=<prefixe_compilateur_croisé> uImage`
  (image zippée au format U-Boot).
  - Possibilité de générer une image « execute in place » (XIP) (attention au support Flash !)
- Le choix de l'architecture cible est déterminant
  - Pour ARM, chaque *board* est supportée dans un fichier spécifique
  - Depuis le noyau 3.10, système « Flattened device tree » devenu obligatoire : description plus abstraite et factorisée de la sous-architecture sous forme de structure dans un fichier dts, compilation modulaire dans un fichier séparé dtb (chargé en mémoire par le bootloader)

CONFIGURATION DU NOYAU

- Prérequis : bien connaître la cible.
- Identifier
  - la partie constante (dans la vie du produit)
  - la partie statique (les éléments non extractibles mais susceptibles d'évoluer)
  - la partie dynamique (USB par exemple)
- Modules
  - support et drivers utiles dès que le système n'est pas constant
  - fichiers externes « .ko » chargés lors de l'exécution du noyau, depuis l'espace utilisateur
- Choix du type de *boot* : initrd/initramfs [peuplé] ou non : dépend de la présence de modules à charger très tôt (généralement: non), prend alors plus de place et de temps au démarrage
- Choix des *drivers* statiques : pilotes nécessaires au boot

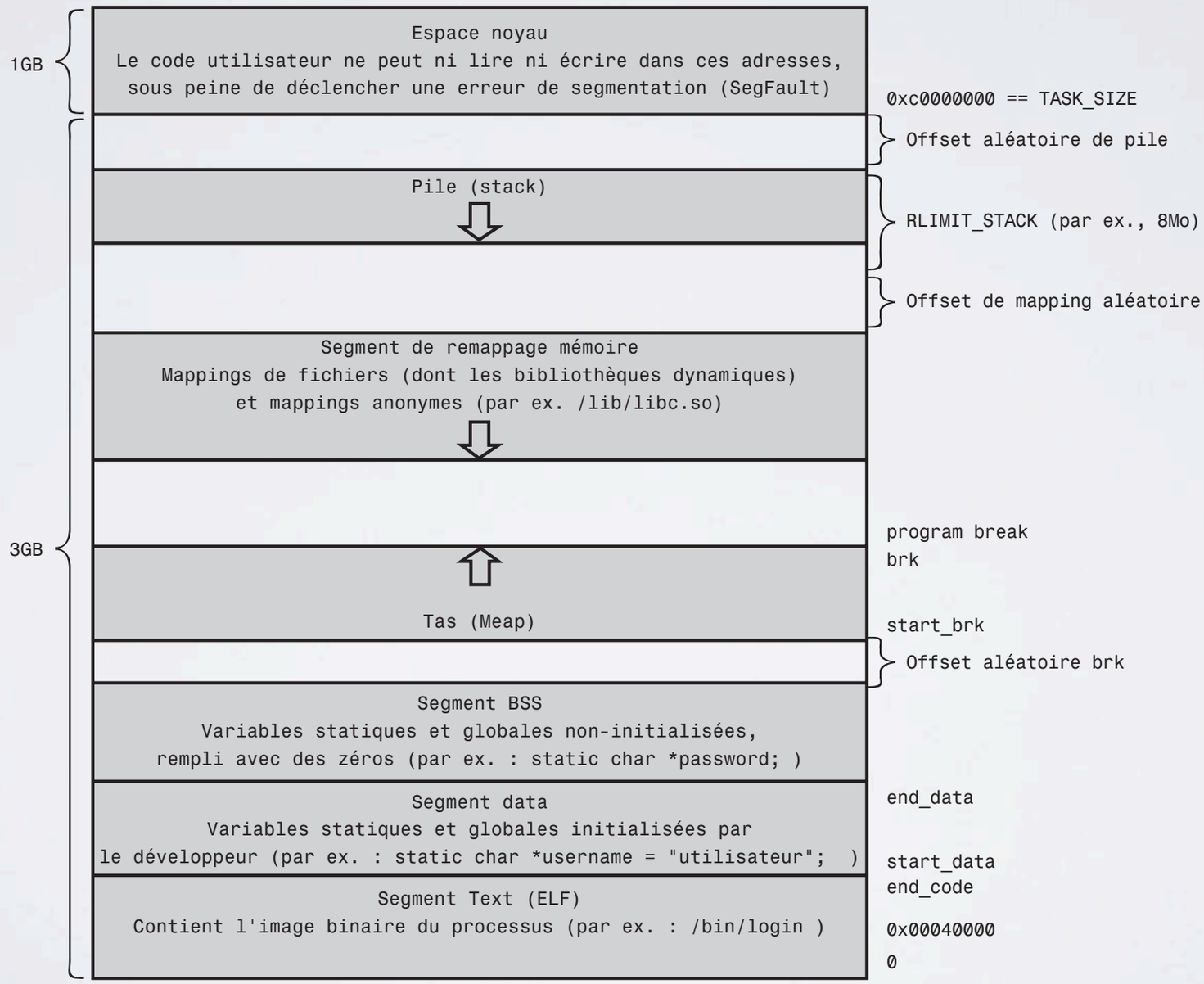
CONFIGURATION DU NOYAU (SANS RAMDISK)

- Embarquer les drivers correspondant au hard en natif si possible :
  - périphérique emportant '/' et son *filesystem*
  - penser au support de ramdisk, du clavier, du réseau, etc.
  - identifier les périphériques réellement dynamiques
  - invalider le support des périphériques non présents
- Prévoir le support, par modules, des périphériques appelés à changer
  - Gain de place en zone '.text' du fichier-*kernel*
  - Placement sur un support éventuellement compressé, non chargé inutilement en RAM en permanence
- Périphériques constants : modules ?
  - Augmente la complexité du système mais pratique pour les mises à jour (ajout de pilotes sans modifier le noyau)
  - Temps de chargement : plus long à charger, mais peut permettre de charger plus vite les fonctionnalités de base et de remettre à plus tard les fonctionnalités secondaires

CONFIGURATION DU NOYAU (AVEC RAMDISK)

- Prévoir le support du Ramdisk et de initramfs
  - Préparer le contenu de l'image
  - initramfs est un système de fichiers racine temporaire, monté au démarrage du noyau
    - si un script /init est contenu, celui-ci est exécuté ; c'est à lui de passer la main au système de fichier sur la flash, en effectuant les montages
- Il est compilé dans le noyau, ou alors peut être chargé extérieurement par le bootloader en RAM avant l'exécution du noyau (paramètre « initrd= »)
- initramfs a succédé à initrd, devenu totalement obsolète
- En réalité, il y a toujours un initramfs dans les noyaux 2.6.x, mais il est vide par défaut
- Ce système est utile pour des chargements avant le système nominal ; ce cas est rare dans l'embarqué ; on peut sinon l'utiliser comme moyen simple de créer un « blob » entièrement chargé en RAM

PROCESSUS LINUX



LE SYSTÈME

- Distributions : définition et problématiques (packaging)
- Le système de fichiers arborescent '/'
  - Choix du système de fichiers (FS) *[cf slide suivant]*
  - Contenu : /lib, /etc, /bin, /sbin, /usr, autres
  - Placer des données sur une ou plusieurs autres partitions ?
    - le but : sécurité, mises à jour, stockage (séparation logique)
    - politiques de montage : standard, extension en RAM, masquage, option *bind*, en *loopback*, ou encore FS unionfs
- inspection d'un système complet

SYSTÈMES DE FICHIERS

- Systèmes de fichiers sans disque
  - Ramdisk, Initramfs (associé au noyau au démarrage) : en RAM
  - **TmpFS** : en RAM dynamique, occupe la RAM au fur et à mesure de son peuplement, jusqu'à une taille limite
- Systèmes en Flash
  - Les FTL (Flash Translation Layer) : FTL – NFTL (NAND FTL) ; système **MTD** et **UBI**
  - FS usuels derrière un contrôleur (**vfat**, **ext2**), précautions (noatime), problèmes de synchronisation
  - Systèmes spécifiques
    - Compressé (**JFFS2**/**SquashFS**) ou non (**YAFFS2**)
    - RO (SquashFS/CramFS) ou RW (JFFS2/YAFFS2/**UBIFS**)
- Montage « **loopback** »
  - pour construire une image de système de fichiers
  - peut alors nécessiter une émulation du système MTD (flash) pour les FS spécialisés

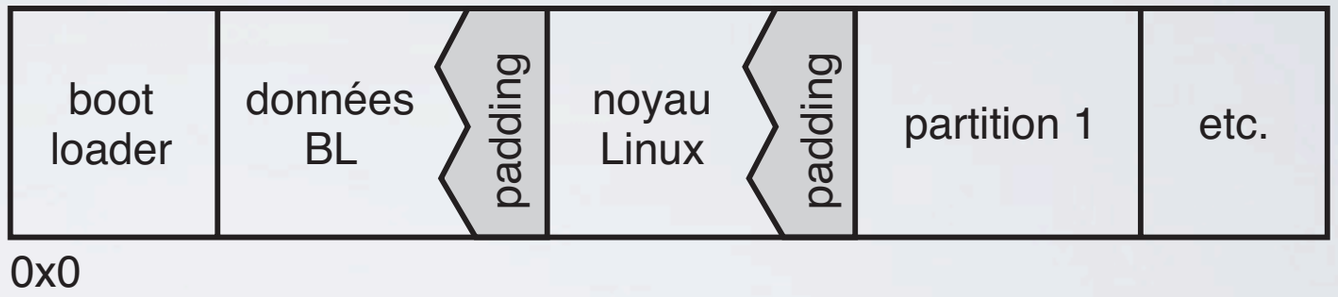
APARTÉ POINTS DE MONTAGE

« Monter » un support, c'est remplacer (virtuellement) le contenu d'un répertoire (généralement laissé vide) par le contenu du système de fichiers monté. Tous les systèmes de fichiers sont accessibles par un chemin dit « absolu », qui fait référence au système de fichier '/'.

Par exemple monter un *ramdisk* dans /tmp signifie que le contenu de /tmp sera rendu inaccessible dans l'arborescence du système et remplacé par le contenu du *ramdisk*.

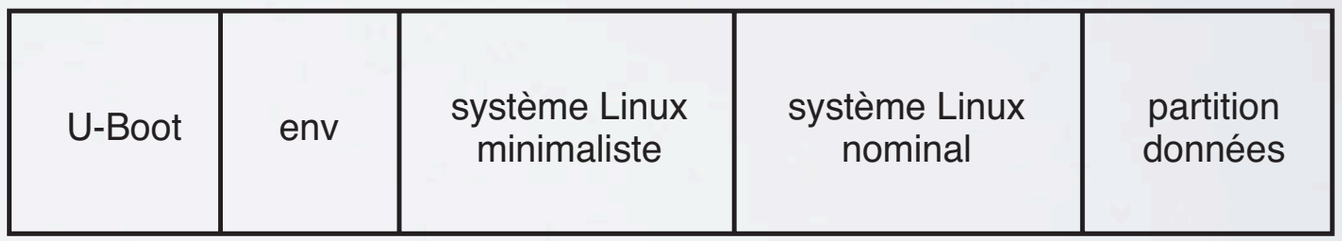
ARCHITECTURES

représentation standard
(avec *padding* explicite)

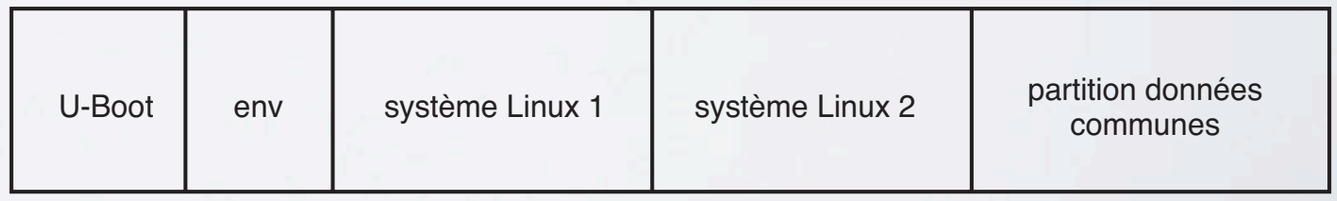


standard à données séparées

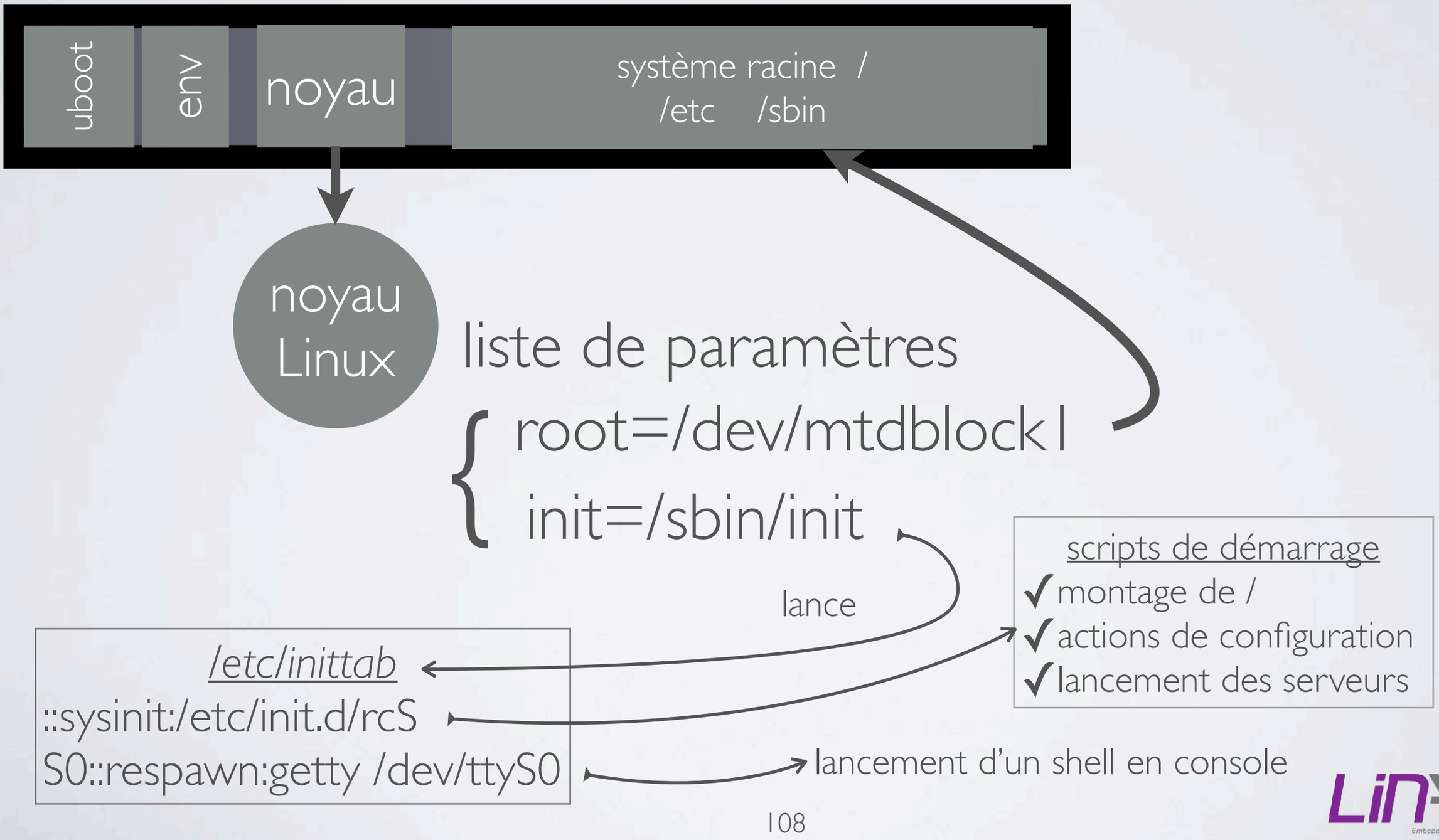
idem précédent
+ système de secours/mise à jour



bascule et données séparées communes



DÉMARRAGE


uboot
env
noyau
système racine /
/etc /sbin
noyau Linux
liste de paramètres
root=/dev/mtdblock1
init=/sbin/init
lance
scripts de démarrage
✓montage de /
✓actions de configuration
✓lancement des serveurs
/etc/inittab
::sysinit:/etc/init.d/rcS
S0::respawn:getty /dev/ttyS0
lancement d'un shell en console

CONSTRUCTION, OPTIMISATION D'UN SYSTÈME

- peut être plus ou moins (très) long suivant la solution choisie
  - penser au matériel, à ses ressources, aux contraintes (support, mises à jour)
  - choix qui engage pour longtemps
- plusieurs approches possibles :
  - à partir d'une distribution pour l'embarqué existante (commerciale ou libre) ou par construction totale via un *framework* spécialisé
  - top-down vs bottom-up
- *Frameworks*
  - Buildroot et OpenEmbedded-Core (Yocto Project)
  - la Busybox

CONSTRUCTION : À PARTIR DE L'EXISTANT

- Approche top-down : éliminer le superflu
  - partir d'une distribution proche de ce souhaité, « minimale »
  - supprimer des paquets, retailler en supprimant les fichiers
  - peut être très long, peu efficace, rédhibitoire
- Distributions intéressantes (pré-optimisées) : Damn Small Linux, Ångström
- Distributions plus standard : Fedora, Debian, Ubuntu (existe en version embarquée)
- Approche top-down alternative : on part d'un système existant dont on extrait les binaires utiles
- Intérêt limité : seulement x86

CONSTRUCTION : ÉLÉMENTS D'UN SYSTÈME

- Approche *bottom-up* : éléments nécessaires
  - un noyau
  - un système de fichier / contenant
    - la configuration (/etc),
    - les entrées nécessaires de /dev et/ou udevd (statique ou dynamique par devtmpfs — à présent préféré),
    - les programmes init, un shell compatible Bourne-shell et mount
    - le nécessaire pour héberger : /bin, /lib, /usr, /dev, /tmp, /var, /home
- Approche raisonnable pour un système proposant une fonctionnalité définie, peu volumineuse
  - On peut ajouter un outil de gestion de packages
  - Choix possible de projets « réducteurs d'empreinte » tels Busybox ou µClibc
- Exemple d'approche bottom-up packagée, documentée, à partir des sources : embedded gentoo, OpenEmbedded, Buildroot.

CONSTRUCTION : LA BUSYBOX

- Binaire unique contenant une multitude d'utilitaires
  - Basiques : cp, mv, ls, ln, etc.
  - Plus complexes : serveurs http, dhcp, vi, sed, etc.
  - Liens symboliques entre le nom des programmes classiques et le binaire /bin/busybox (résolution par nom d'appel)
  - Permet de gagner beaucoup de place, facile à mettre en place (une seule compilation)
- Compilation configurée
  - Choix des « sous-logiciels » incorporés
  - Configuration par interface en ncurses (console graphique)
  - Compilation croisée

CONSTRUCTION DU SYSTÈME DE FICHIERS RACINE 1/4

- Monté sur /
- Répertoires et volatilité
    - Peuvent être en lecture seule (penser à l'option « noatime » en lecture-écriture)
        - /sbin, /bin : exécutables fondamentaux
        - /lib : bibliothèques partagées
        - /usr : exécutables et bibliothèques « de second niveau »
        - /etc : configuration, constant sauf pour quelques fichiers (on utilisera des liens symboliques)
    - Un peu volatils, rémanence souhaitable (pourquoi pas sur Flash)
        - /home, /root, /user
        - si installation de logiciels tiers, /opt ou autres précédemment listés (peu recommandé)
    - Très volatils, aucune rémanence : en RAM
        - /tmp, /var
        - La structure de /var peut être régénérée au boot

CONSTRUCTION DU SYSTÈME DE FICHIERS RACINE 2/4

- Création du fichier conteneur ou *firmware*
  - création des répertoires /bin, /dev, /etc, /var, /lib, /sbin, /home, /proc, éventuellement /mnt, /usr, /usr/bin, /usr/sbin, /initrd
  - /tmp lien symbolique vers /var/tmp
  - pour x86, « /boot » n'est pas nécessaire si son contenu se trouve dans une partition séparée
  - dd, mke2fs + mount en *loopback* et copie des fichiers (x86)
  - Ou création via des outils spécialisés (mksquashfs, mkfs.jffs2) acceptant une arborescence :
    ```
    $ mkfs.jffs2 --pad=0x700000 -o rootfs.jffs2 -e 0x20000 -n -d/tmp/jffsroot/
    ```
  - Ou création via des outils + montage par émulation mtd et loopback
    voir: http://wiki.openmoko.org/wiki/Userspace_root_image
- Population des répertoires
  - /etc : fichiers de configuration et scripts de démarrage
  - /bin et /sbin : binaires d'origine ou busybox (impacte /lib)
  - /dev : console, accès ressources, partitions (statique ou dynamique par udevd)
  - /var, /home, /proc laissés vides (points de montage)

CONSTRUCTION DU SYSTÈME DE FICHIERS RACINE 3/4

- fichiers de configuration de /etc
  - fichier fstab : points de montage
    - rootfs : /
    - /proc (en procfs) ; /sys (en sysfs)
    - en tmpfs : /var et/ou /tmp
    - FS séparé de la partition racine : /home ou /user
- lien /etc/mtab ⟶ /proc/mounts (anciennes versions)
- Configuration réseau, configuration des interfaces (/etc/hosts, /etc/resolv.conf, etc.)
- passwd (et shadow) : autorisation des login root

- scripts dans /etc
  - Script de boot
    - montages
      - création de l'arborescence de /var
      - var/tmp, /var/log, /var/lock, /var/run
      - chmod a+rwX /var/tmp
- Si racine en RO, lien fichiers variables de /etc dans /var (on les y génère)

CONSTRUCTION DU SYSTÈME DE FICHIERS RACINE 4/4

- Le dossier /dev contient les fichiers spéciaux de périphériques (device)
  - type caractère ou bloc
  - identifié par numéros majeur et mineur
- problème : comment sont créés ces fichiers ?
  - autrefois : manuellement, à froid, en dur
  - à présent : usage de devtmpfs (monté sur /dev) et de udev (démon udevd)
    - compiler le noyau avec DEVTMPFS et DEVTMPFS_MOUNT
    - possibilité d'écrire des scripts udev pour déclencher des actions (par exemple montage automatique de périphériques externes détectés par udev)

INTERFACE UTILISATEUR

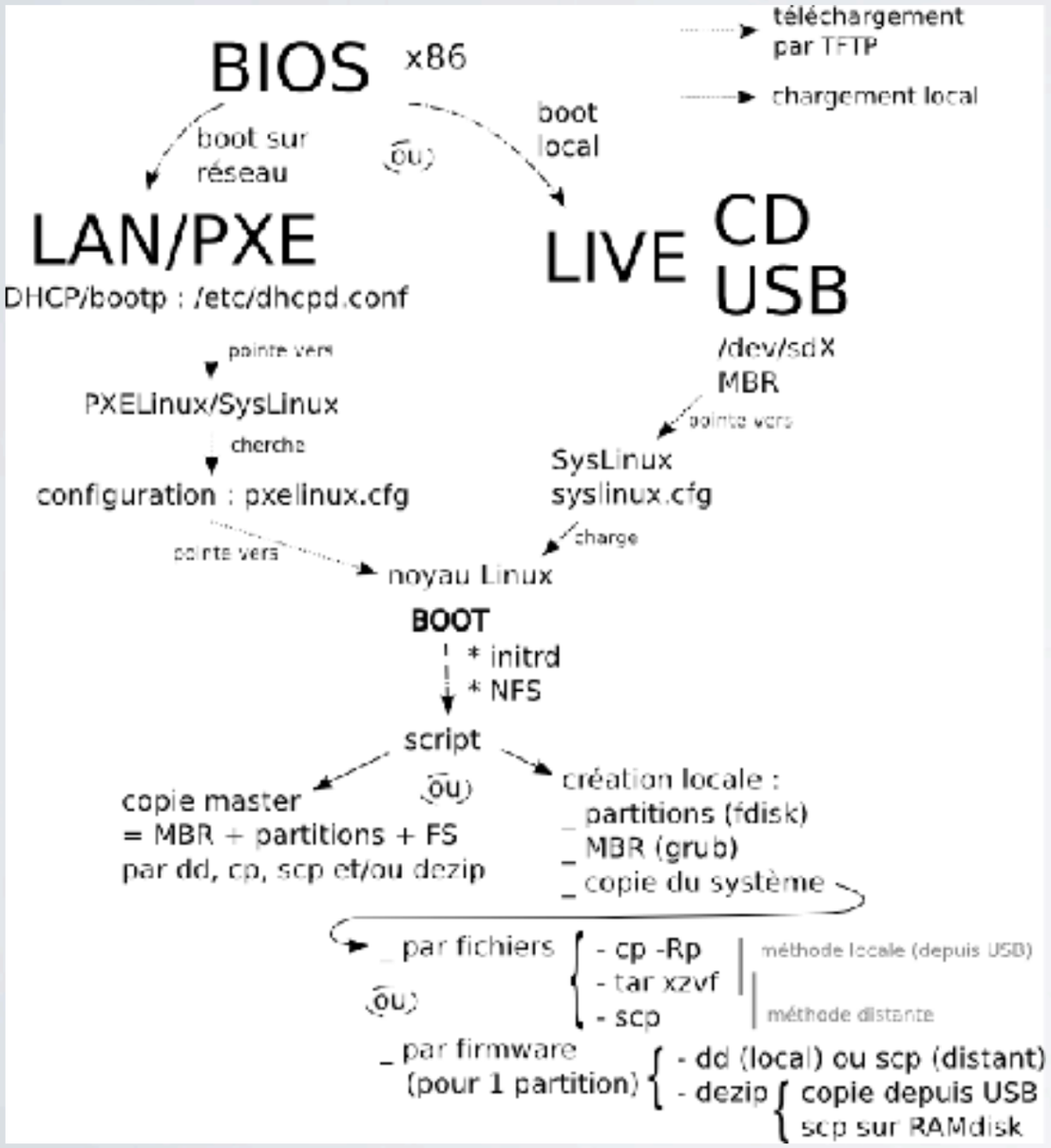
- Solution classique
  - Serveur X11 : Xorg
  - Frame-buffer compatible X : Xfb
  - Sinon : directement en framebuffer
- Bibliothèques graphiques
  - GTK : de moins en moins utilisé (version embarquée : Hildon)
  - Qt : de plus en plus utilisé
    - C++ ou Python
    - version Qt embedded
    - QML
- Environnements
  - Tout Java : Android (assez incompatible), GoogleTV
  - Packagé compatible : Maemo, MeeGo/Mer, ...
  - croisé HTML5 : Tizen
- Par serveur/client web
  - Comme serveur : utilisation de CGI (serveur thttpd, lighttpd, etc)
  - Comme client : Webkit
  - Firefox OS
  - Apparition de ChromeOS

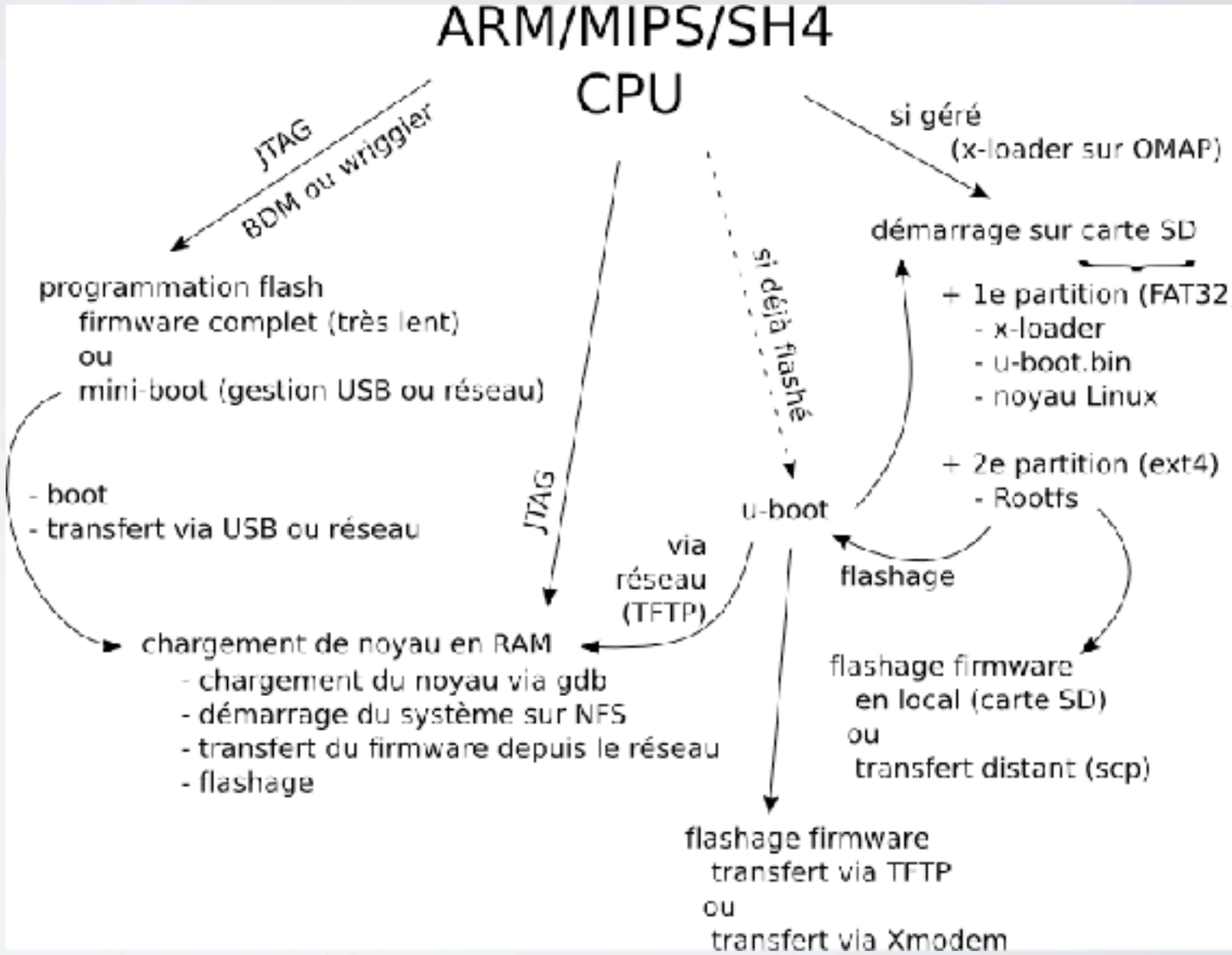
INTÉGRATION

MÉTHODE D'INTÉGRATION

- Cible vs PC de développement
- [rappel] 3 acteurs sur la cible : le *bootloader*, le noyau, le système
- Sur la cible
  - Démarrer un noyau Linux : local (flash ou SD si géré), ou transféré par TFTP (si *bootloader* sur Flash ou SD gérée), ou envoyé dans la RAM par J-TAG (si SD non gérée et pas de *bootloader* par exemple)
  - Démarrer le système Linux de la cible
  - Changer le *firmware* par une commande dd sur la bonne partition /dev/mtdblockX
- Changement de noyau sur la flash possible à chaud (sauf si utilisation de XIP)
- OU : programmation de la Flash par J-TAG
- À distance
  - TFTP pour noyau Linux, via le Boot loader
  - NFS en ligne de commande pour Linux
  - Serveur NFS sur la machine de développement, partage de la racine d'un système embarqué complet
  - Avantage : intégration fluide, possibilité de modifier le système à la volée, possibilité de flasher

INTÉGRATION INITIALE





INTÉGRATION CONTINUE PAR NFS

connexion réseau 10/100 (RJ45)

1. lancement serveur NFS
   /etc/exports:
   `<chemin_rootfs>`
   `*(rw,no_root_squash,async)`

2.démarrage de la carte
(u-boot et noyau présent)

système pour la cible
/
/etc
/lib
/bin
...

montage

3. noyau paramétré : root=/dev/nfs
   `nfsroot=<NFS_SERVER_IP>:<chemin_rootfs> rw`
   `ip=<client-ip>:::<netmask>::<device>:`

Chargement *dynamique* de fichiers
en RAM (lors d'un accès)

accès lecture, écriture, exécution

LINACS
Embedded Linux project accelerator

FIRMWARE

- Qu'est-ce qu'un *firmware* ?
  - une agglomération d'éléments
    - binaire (« blob ») :
      - bootloader et/ou noyau et/ou système(s) de fichiers
      - au choix, selon ce que l'on veut modifier, mettre à jour, flasher initialement, réinitialiser, etc.
    - ou fichiers :
      - une archive compressée
  - un système complet et/ou une configuration et/ou un ensemble de logiciels précis, etc.
    - potentiellement signé (vérification d'intégrité et d'authenticité)
- Politiques de mises-à-jour
  - impact sur l'architecture de la solution (à considérer dès la phase de spec !)
  - impact sur la gestion de projet
  - problématique de support et de cout

INTÉGRATION CONTINUE

- Intégrer régulièrement
  - méthode Agile
  - au fur et à mesure de la réalisation, par paliers
  - permet de vérifier que tout fonctionne correctement
- Ne pas attendre la création de *firmware*
  - long à la création et au chargement sur cible
  - mettre en place un environnement de test et de débuggage adapté

DEBUG PAR GDB

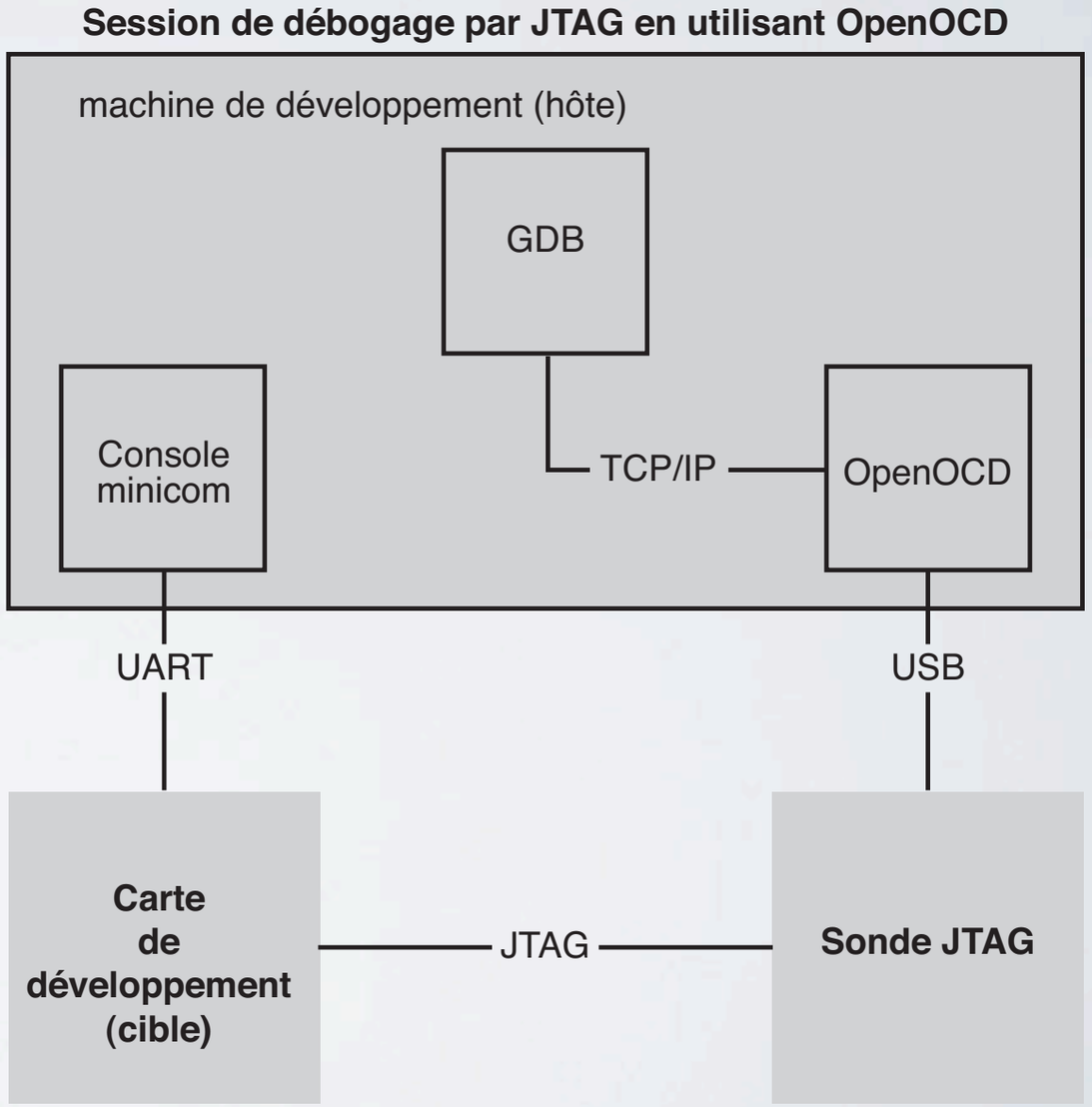
- Avantages
  - méthode familière
  - fonctionnalités (*breakpoints*, examen de variables, désassemblage...)
  - coût matériel modeste (un câble null-modem RS232)
- Inconvénients
  - un peu d'intrusion logicielle
  - temps-réel perturbé
- Principe
  - gdbserver sur la cible, gdb client sur la station qui analyse
  - connecter gdb client à gdbserver
  - attacher gdb du client au processus du serveur

MISE EN PLACE GDB/GDBSERVER

- Besoins du client
  - Table des symboles : compiler avec l'option -g
    - idem pour le noyau (option de configuration DEBUG) ; pas strictement nécessaire mais aide beaucoup.
      Parade : utiliser le « system.map », liste de symboles générée lors de la compilation ; analyser à la main ou charger manuellement dans gbd
  - Liaison série configurée au préalable
- Commande serveur
  - Débuggage d'un programme : `gdbserver /dev/ttyS0 <programme>`
  - Débugguge du kernel : plus complexe et pas forcément très stable — préférer le JTAG
- Commande client
  - `gdb <programme>`
  - `(gdb) target remote /dev/ttyS0`

JTAG ET DEBUG

- JTAG IEEE 1149.1
  - Pilotage depuis une station de développement Linux (ou Windows...)
    - solution commerciale (interfacée sous Eclipse le plus souvent) ou OpenOCD
- Debug possible par gdb (émulation gdbserver), lien série pour la console



JTAG IEEE 1149.1

- Avantages
  - Méthode standard, accès aux registres du CPU
  - Sonde matérielle, pas d'intrusion logicielle
  - Temps réel respecté au mieux
  - Nécessaire pour démarrer une carte neuve, vierge de tout logiciel
- Inconvénients
  - Signaux de sonde : faille de sécurité (intrusion possible)
  - Efficacité dépendante de la sonde
  - Breakpoints (hard) limités (mais softs possibles)
  - Coût matériel (sonde)
  - Outils périphériques coûteux (Code Composer), non Open-Source

DEBUG EN ENVIRONNEMENT VIRTUEL

- Avantages
  - Économe en matériel
  - Accélération du cycle du code au test (améliore l'intégration continue)
  - Environnement intégré tout simulé (permet de bouchonner plus facilement, par exemple)
- Inconvénients
  - Peut occasionner des développements d'émulation de périphériques
  - Périphériques émulés ou simulés (exemple : Flash), architecture CPU pas totalement correspondante, gestion difficile de plusieurs partitions
  - Temps réel non respecté
- Qemu en logiciel libre : architectures ARM, MIPS, sh4, x86

http://commons.wikimedia.org/wiki/File:Innards_of_an_AI-139a_mechanical_watch.jpg?uselang=fr

MENER UN PROJET

LES DIFFÉRENTES ACTIONS

- la spécification
  - réfléchir avant d'agir, mais ne pas aller trop loin (rester Agile !)
  - attention aux grands choix ayant un impact à long terme (maintenabilité, mises à jour) : choix des systèmes de fichiers et de type de déploiement des développements (patch vs packaging)
- la réalisation
  - le développement
  - l'intégration
- l'intégration (attention : ambiguïté du terme !)
- le test

COMPÉTENCES ET RÔLES

- Qui fait quoi dans l'équipe ?
  (pour un projet « simple »)
  - Un chef de projet (maître d'œuvre)
  - Un développeur kernel (au besoin)
  - Un concepteur et intégrateur de la solution
  - Un ou plusieurs développeur(s) de couches logicielles
  - De préférence l'un des trois premiers est un expert avec plusieurs années d'expérience
- Demander son DIF à temps !

tâches parallélisées

- Compilation / intégration : série
  - ➡1 personne (garder la logique du système/ homogénéité)
- Travaux noyau : parallèle (sur cible ou non ?)
  - ➡1 personne par module/tâche spécifique
- Intégration application graphique
  - ➡Par une ou plusieurs personnes
- Couche logicielle maison
  - ➡à voir (souvent le concepteur/intégrateur)

ÉVALUER LE PROJET

- Qu'est-ce qui existe déjà ?
  - *Out of the box*
  - À réadapter
- Qu'est-ce qui peut être intégré ?
  - Choix d'un *framework* ? (graphique, OE, buildroot, etc.)
  - Quelles implications sur l'architecture ?
- Qu'est-ce qui reste à développer ?
  - Sur le *framework* ?
  - Sur quoi s'appuyer ?

ESTIMER LE TEMPS

- En avant-vente puis au fur et à mesure de la réalisation
- « Combien de temps ça va te prendre ? »
- Définir le périmètre & specs ? 5 à 10 jours
- Compiler un système Linux ? 10 à 15 jours
- Intégrer un système Linux ? 5 jours
- Optimiser ? 5 jours
- Coder un module ? Heu... 10 à 40 jours
  (estimations non contractuelles à la louche)

ENVIRONNEMENT DE DÉVELOPPEMENT

- Machine puissante (compilation), avec beaucoup d'espace disque
- Sous Linux !! + accès Root
- Connexion Internet/réseau
- Dépôt de source (Git vs SVN)
- Forge (RedMine)
- Carte(s) de dev' (& associé : SD/MMC, lecteurs, alim, câbles série, adaptateur USB-série)
- Cible finale
- JTAG ?

METTRE EN PLACE UN POC

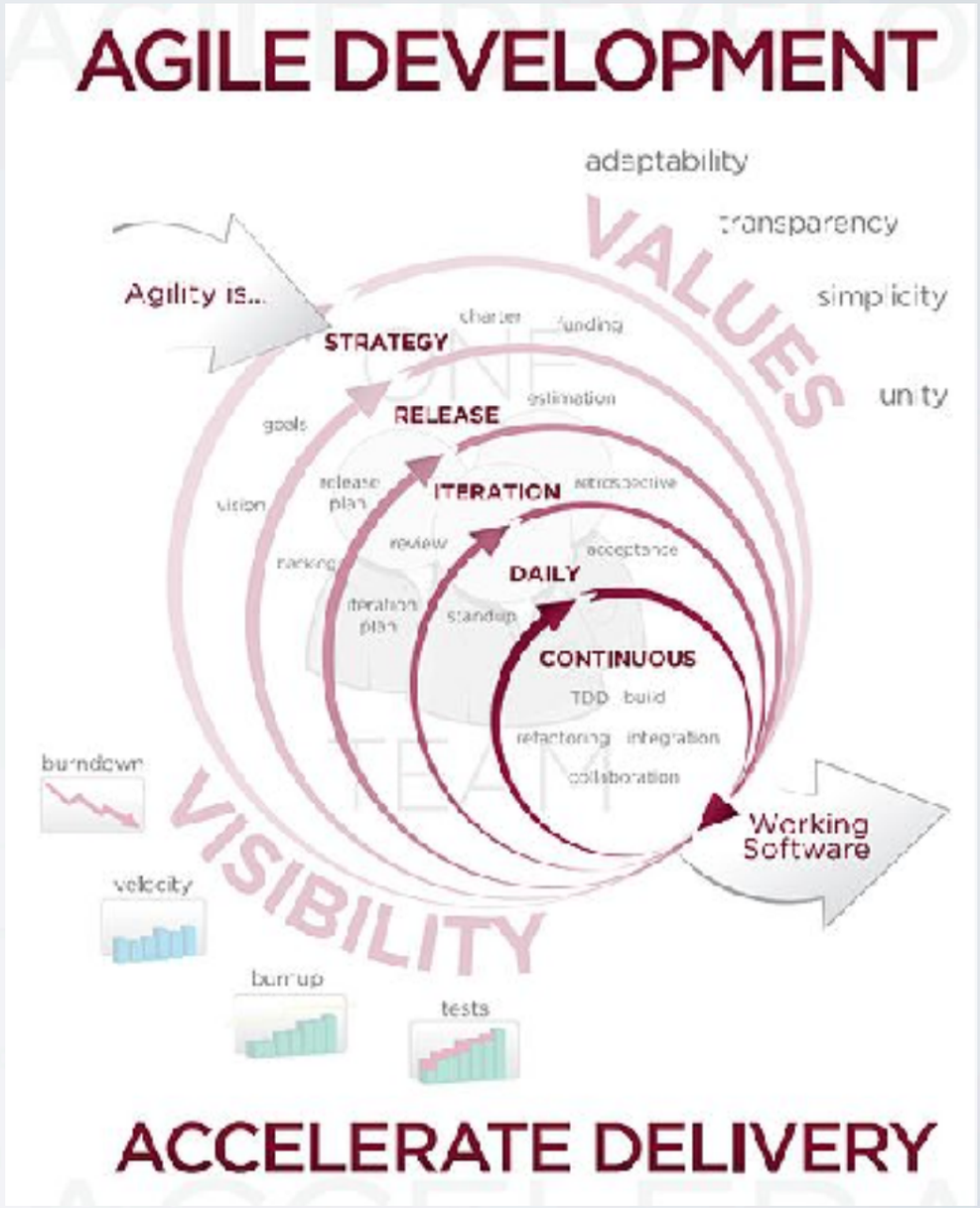
- Mise en place de *proof-of-concept*
  - Premier tour du périmètre
    - Permet d'estimer la charge de travail
    - Brouillon à réutiliser
- Utiliser une solution Linux embarqué prête à l'emploi (Angström)
  - Coder en script (shell, Python)
  - Sans optimisation aucune
  - Sous émulateur ? (Qemu)

LA SOUS-TRAITANCE

- Les SSII
  - répondent au cas par cas
  - souvent en prestation : pas de garantie de résultat
  - l'évolution tend vers le forfait pour les nouveaux projets
  - offre peu claire, peu de spécialistes *pure players*, beaucoup de compétences et de risques rattachés
- Les éditeurs
  - fournissent des systèmes sur étagère
  - peuvent adapter leurs propres solutions, mais ne proposent pas forcément des développements originaux lourds : business concentré sur leurs produits
- Les couts
  - dépend du mode d'intervention & du nombre de jours facturés
  - un ingénieur (junior) : 400-700 €/jour en SSII
  - un expert
    - 1000 à 1500 €/jour en SSII (dépend du nombre de jour et du fractionnement)
    - moins en indépendant (psychologie du client, économiquement regrétable)
  - une solution sur étagère : 5 000 à 20 000 €
  - autres (modules proprios, licences commerciales, etc.) : à marchander

ÊTRE AGILE

- développement itératif et amélioration continue
- flexibilité
- cycle adapté à la réussite projet
- valeurs



VALEURS COMMUNES DU LIBRE ET DE L'AGILE

- ✓ La cathédrale et le bazar, par ESR
- ✓ *release eraly, release often. And listen to your customers.*
- ✓ idée de cohésion/communautaire :
  - valeurs partagées
  - mode de fonctionnement collaboratif
  - plaisir
- ✓ KISS : *Keep It Simple, Stupid*

LEAN, AGILE ET LINUX EMBARQUÉ

- les particularités :
  ➤ adaptation (bespoke), OpenSource, problématique PI

- avantage sur le TTM :
  ➤ négociation PI/royalties, base d'existant, ROI long terme

- adaptation PDCM/Deming
  ➤ importance de l'intégration itérative

- risques : mauvaise estimation (temps & couts)/complexité, développement de modules, propriété intellectuelle…

DIFFICULTÉS

- ✤ côté développeurs : compréhension des méthodes !
- ✤ côté direction : application des méthodes "pré-mâchée" sans réflexion/adaptation ! (ex. de l'XP)

- ❖ attention à la doc, tout de même
- ❖ mise en place des tests

- ✦ garder le côté "artisanal"/ bespoke
- ✦ attention au Lean perçu comme "low cost" : doit rester dans une idée de "luxe"

- ➡ au-delà (en amont/ contexte) :
  - ✓ spécification projet
  - ✓ vente des projets
  - ✓ contrats/ engagements

INFORMATIQUE INDUSTRIELLE 2.0

- outils :
  - communication : IRC/gtalk
  - forge : Redmine (Git/SVN, Wiki, *bugtracking*)
  - Internet : wikis, forums/*mailing lists*, blogs
- communautés/collaboration (GENIVI/Linaro ; MeeGo, Android ; etc)
- remontée d'expérience, partage
- différents cas de management/*business model*
  Intel, Nokia, Google
  en France : Sagem, Philips, Free, Neuf, Orange/Soft@home
- encore beaucoup de progrès à faire :
  - communication (contradictions)
  - Organisation
  - SI (poste de travail, Internet, matériel)

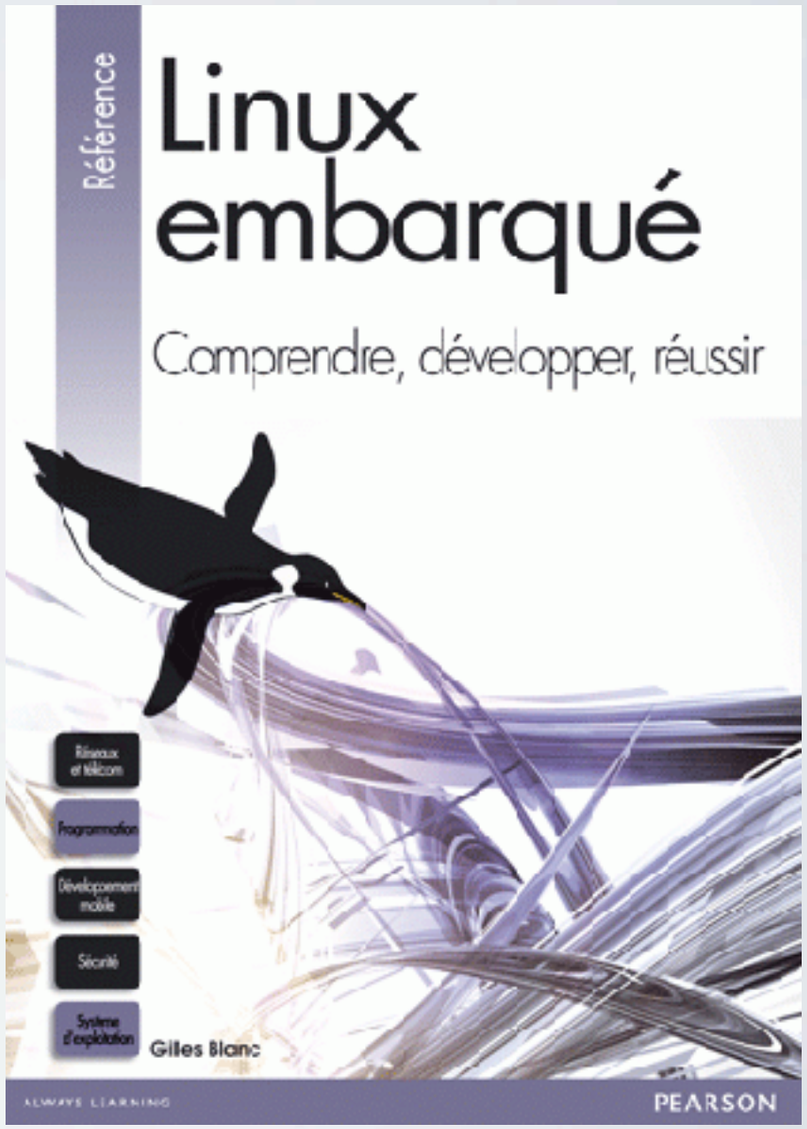
ALLER PLUS LOIN : LE LIVRE

- *Linux embarqué*
  comprendre, développer, réussir
  (éd. Pearson, déc 2011, 460 pp.)

- management (économie, juridique, stratégie, etc.)

- technique

- au-delà : plaidoyer



LiNACS
Embedded Linux project accelerator

MERCI DE VOTRE ATTENTION

LINACS

http://linacs-consulting.com

formation@linacs-consulting.com