



Outils de développement pour Linux embarqué

Pierre FICHEUX
pierre.ficheux@openwide.fr

Thomas MONJALON
thomas.monjalon@openwide.fr

Mars 2011

- Présentation d'Open Wide
- Rappels sur l'architecture d'un OS Linux embarqué
- Briques de base: noyau Linux, Busybox, GCC
- Environnements de « production »: Buildroot, OpenEmbedded, OpenWrt
- Mise au point: GDB/KGDB
- Instrumentation: LTTng, Ftrace
- QEMU



Présentation Open Wide

- SSII/SSLL créée en septembre 2001 avec Thales et Schneider
- Indépendant depuis 2009
- Environ 90 salariés sur Paris et Lyon
- Industrialisation de composants open source
- Quatre activités :
 - OW : système d'information
 - OW outsourcing: hébergement
 - OW ingénierie: informatique industrielle
 - OW technologies: composants Java

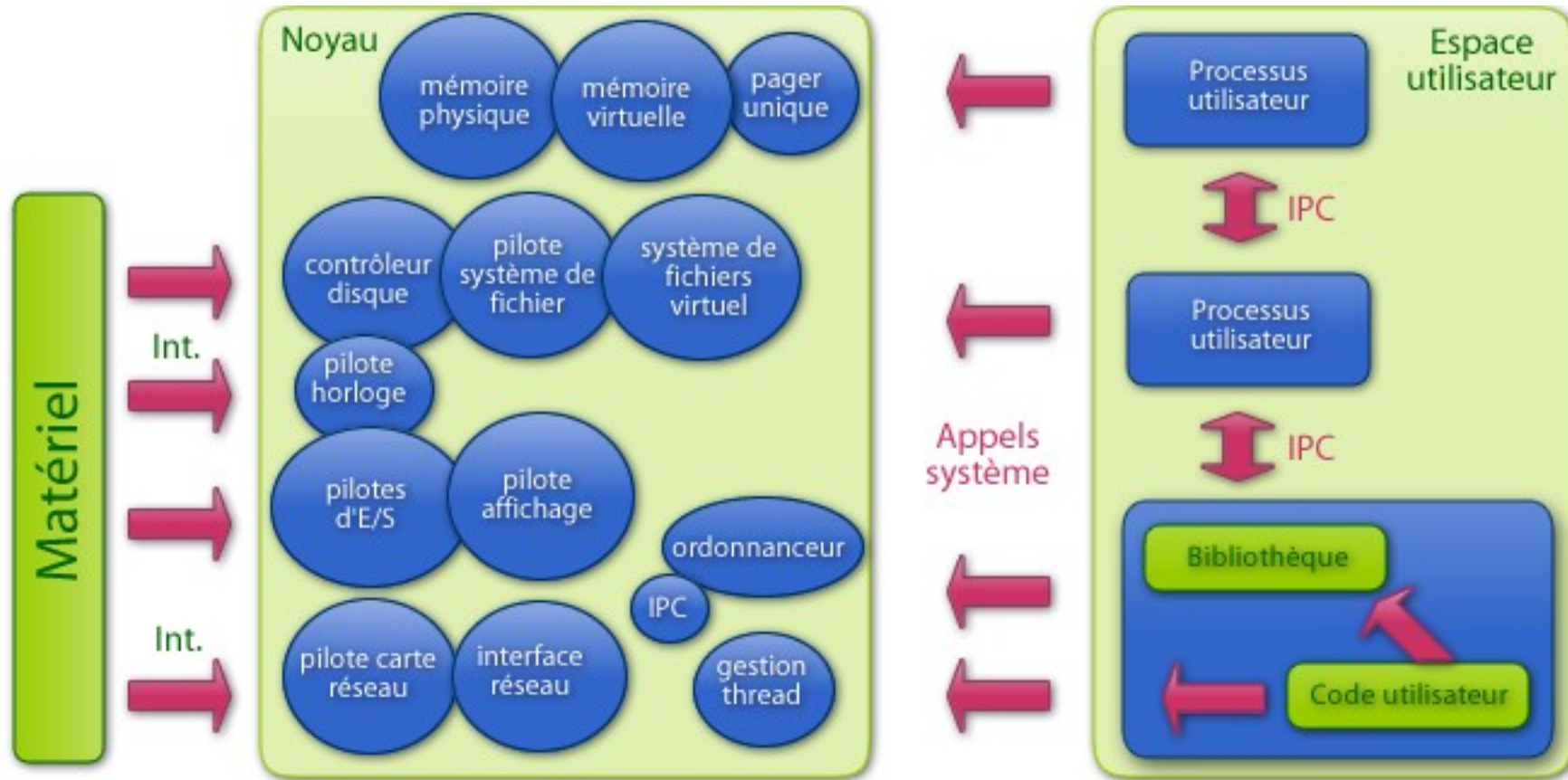




Rappels, généralités



Architecture de Linux 1/2





Architecture de Linux 2/2

- Les éléments fondamentaux :
 - Le noyau: en théorie, interface unique avec le matériel, API de programmation spécifique (modules Linux)
 - Le «root filesystem» (root-fs): les commandes et fichiers système, communs à (presque) toutes les versions d'UNIX (API standard => POSIX)
 - Un système Linux est *obligatoirement* l'association noyau + root-fs
- Embarquer Linux = optimiser le noyau + construire un root-fs léger

- La structure de Linux est complexe :
 - 2 espaces de mémoire
 - Multi plateforme
- Cela rend difficile la disponibilité de certains outils par rapport à d'autres environnements RTOS (OS21, VxWorks, ...)
- L'approche open source ne simplifie par forcément les choses :
 - Parfois peu de documentation
 - Foisonnement d'outils spécialisés
 - Nécessite un effort de mise en place



Briques de base

- Première version 0.01 en septembre 1991 (environ 50 Ko de code compressé)
- Dernière version: 2.6.38 (75 Mo de code compressé)
- 26 architectures officiellement supportées
- Qualité du support variable => nécessité de « patch » externes pour certaines architectures (ARM, PowerPC, SH4, ...) et surtout certaines cartes
- Difficulté de portage => pas de *normalisation* pour l'ajout d'une carte pour certaines architectures (ex: ARM)
- Nombreuses versions de constructeurs hors du « mainline »

- GNU/Linux basé sur « coreutils » est trop volumineux pour l'embarqué

```
$ ls -l /bin/bash
```

```
-rwxr-xr-x 1 root root 877480 21 mai 2010 /bin/bash
```

- Busybox remplace la majorité des commandes Linux par des versions « réduites »

```
$ ls -l /bin/busybox
```

```
-rwsr-xr-x 1 root root 670856 15 mars 09:45 /bin/busybox
```

- 95 % des OS Linux embarqués utilisent Busybox
- Simple, léger, portable
- Diffusé sous licence GPLv2
- Pas vraiment un « outil » mais un composant incontournable !

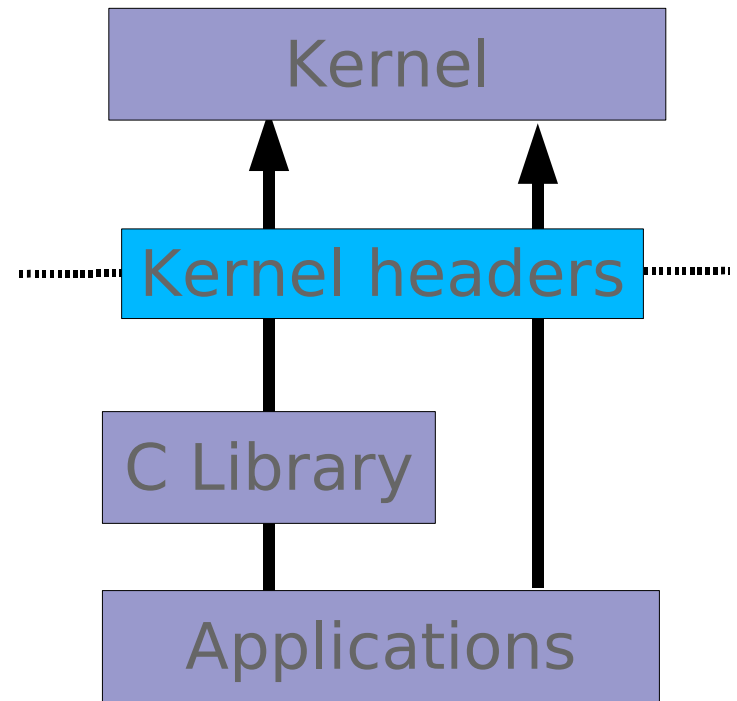


GCC (toolchain) 1/5

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



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



- Numéro des *system calls*, dans `<asm/unistd.h>`

```
#define __NR_exit      1
#define __NR_fork     2
#define __NR_read     3
```

- Définition de constantes dans `<asm/generic-fcntl.h>`, inclus par `<asm/fcntl.h>`, inclus par `<linux/fcntl.h>`

```
#define O_RDWR      00000002
```

- Structures de données dans `<asm/stat.h>`

```
struct stat {
    unsigned long  st_dev;
    unsigned long  st_ino;
    [...]
};
```



- Utiliser un compilateur binaire :
 - ELDK: <http://www.denx.de/wiki/DULG/ELDK>
 - Code Sourcery:
<http://www.codesourcery.com/sgpp/lite/arm/portal/release644>
- Installation simple
- Support (payant) possible
- Configuration connue => support par les forums
- Par contre:
 - Version des composants figées
 - Choix libC limité



- Construire un compilateur
 - Crosstool => obsolète
 - Crosstool-NG => assez complexe à prendre en main
 - Buildroot / OpenEmbedded (voir plus loin) => uClibc
- Aucun ou n'est « plug and play »
- La mise au point peut prendre des jours, voire plus !



Outils de création de distribution



Création d'une distribution

- Utiliser un produit d'éditeur (Wind River, MV, ...) => €€€
- Adapter une distribution Linux classique
 - Limité au niveau matériel
 - Empreinte mémoire importante
- Créer la distribution « from scratch »
 - Complexe
 - Difficile à industrialiser: gestion des dépendances et des évolutions
- Utiliser un outil de génération : Buildroot, OpenEmbedded, OpenWrt, LTIB

Principes des outils de génération

- Un « moteur » crée la distribution à partir des sources des composants adaptés en appliquant des « patch »
- Ne fournit pas les sources: uniquement les patch et les règles de production prenant en compte les dépendances :-)
- Peut produire la chaîne croisée
- Produit la distribution sous diverses formes
 - Image du bootloader
 - Noyau Linux (zImage, uImage)
 - Image de root-filesystem en EXT2, JFFS2, CRAMFS, TAR, CPIO, ...



Quelque outils disponibles

- OpenEmbedded
 - Moteur écrit en Python
 - Très puissant mais lourd
 - Basé sur des fichiers de configuration
- Buildroot
 - Au départ un démonstrateur pour uClibc
 - Désormais un véritable outil, bien maintenu !
- OpenWrt
 - Dérivé de BR
 - Orienté vers les IAD (Internet Access Device)
- Autres: LTIB, PTXdist, ...

- Lié au projet uClibc (micro-C-libc) : libC plus légère que la Glibc
- But initial: produire des images de test de uClibc
- Moteur basé sur des fichiers *Makefile* et des scripts-shell
- Par défaut, utilise Busybox
- Outil de configuration utilise le langage *Kconfig*
- Produit également la chaîne de compilation (uniquement basée sur uClibc !)
- Pas de version « officielle » avant 2009



Buildroot aujourd'hui

- Repris en 2009 par Peter Korsgaard et Thomas Petazzoni
- Une version officielle tous les 3 mois: 2009.02, ..., 2011.02
- Projet géré sous Git
- Documentation améliorée
- Plus de 300 composants adaptés
- Support CPU x86, ARM, PowerPC, SH4, ...
- Il est assez « simple » d'ajouter un support de carte...si l'on connaît bien le Makefile et Kconfig



Using Buildroot « in a nutshell »

- Télé-charger depuis <http://buildroot.uclibc.org>
- Extraire l'archive
- Configurer par `make menuconfig`
- Compiler par `make`
- Le résultat est dans le répertoire `output/images`
 - Bootloader (U-Boot)
 - Noyau Linux
 - Image(s) du root-filesystem
- La chaîne de compilation dans `output/staging`

Configuration de Buildroot

- Type de CPU + variante, ex: arm puis arm920t
- Options de la cible
- Choix de la chaîne de compilation: interne (uClibc) ou externe (Glibc, Elibc, ...), Crosstool-NG
- Composants de la distribution (répertoire package)
- Formats des images du root-filesystem
 - JFFS2, CRAMFS, SQUASHFS (flash)
 - EXT2, CPIO (ramdisk)
 - TAR (NFS Root)
- Noyau Linux et bootloader (non compilés par défaut)

- Une « généralisation » de l'approche utilisée dans BR
- Utilise un moteur écrit en Python (bitbake) et un ensemble de règles utilisant un principe d'héritage => « recipe » (recette)
- Pas d'interface de configuration
- Processus lourd => plusieurs heures pour la première compilation (environ 30mn pour BR)
- TRES puissant, recommandé dans le cas où l'on gère un grand nombre de configurations
- Gère la notion de paquet binaire, contrairement à BR



Mise au point / instrumentation

Mise au point ou instrumentation ?

- Mise au point: résolution d'un problème de fonctionnement
 - Valgrind: pour problèmes de mémoire
 - GDB / KGDB
 - strace / ltrace
 - Analyse de crash (LKCD / Ksymoops)
- Instrumentation: analyse préventive, profiling
 - Traces en espace noyau: LTTng, Ftrace, etc.
 - Profiling OProfile (noyau/utilisateur)
 - Instrumentation « permanente » du système ?



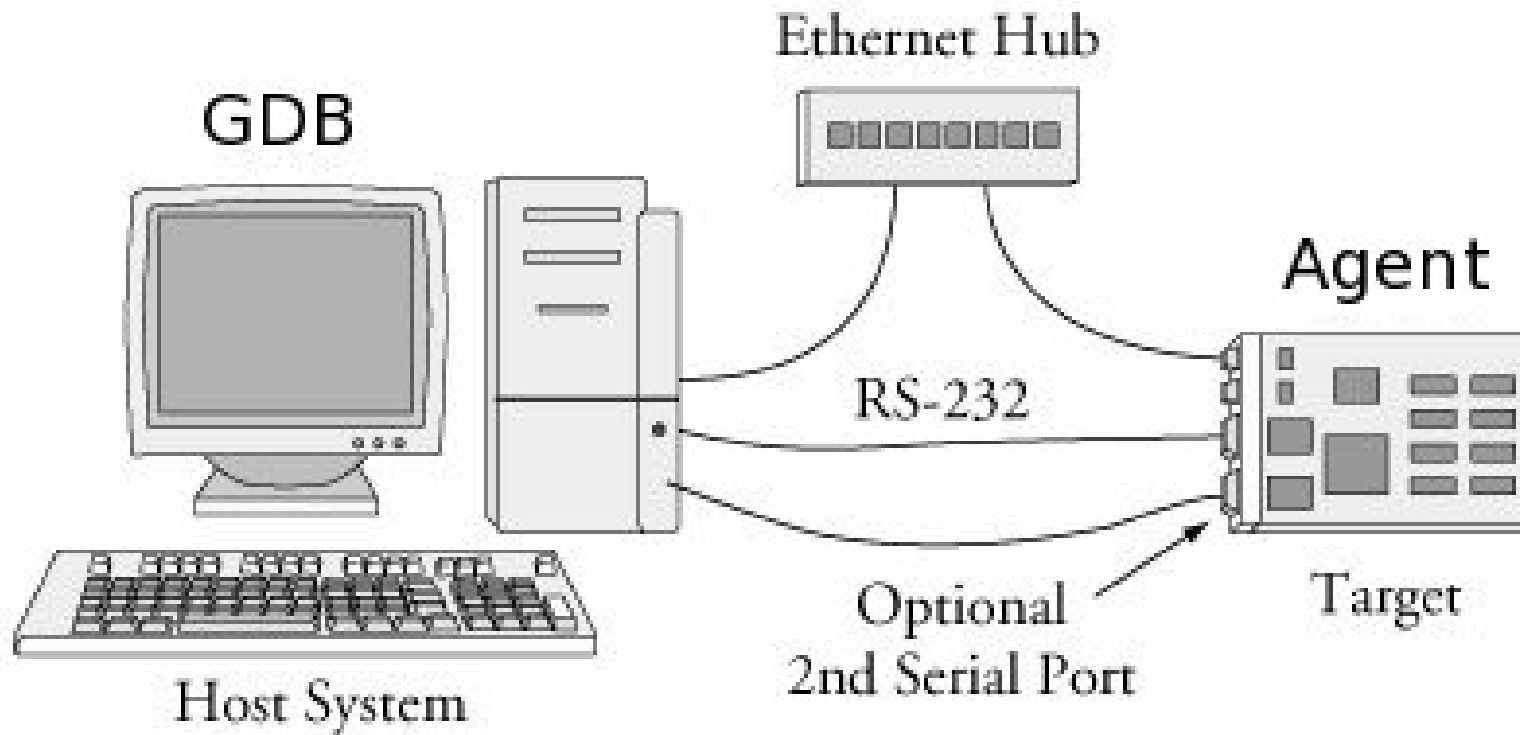
Mise au point en espace utilisateur

- Syslog: indispensable au suivi correct des traces d'un système. Version améliorée avec Rsyslog (serveur de traces, SQL, ...)
- Valgrind: exécution du programme dans une « machine virtuelle » => Pas de recompilation mais performances dégradées
 - Solutions propriétaires: Purify (IBM), Insure++ => recompilation
 - Voir démonstration dans Debug/Valgrind
- strace / ltrace: trace des appels systèmes et bibliothèques. Rustique mais efficace, filtrage des appels nécessaires.
- GDB

- Mode distant (remote) basé sur un protocole standard « remote protocol » développé pour GDB
- Utilisation d'un agent sur la cible et d'un débogueur croisé sur l'hôte
 - Agent gdbserver pour espace utilisateur
 - QEMU et/ou KGDB pour l'espace noyau
- D'autres agents intégrés dans des sondes JTAG (ex: BDI3000 Abatron => bdiGDB, OpenOCD)
- Ces outils peuvent être intégrés à des IDE (Eclipse, QtCreator)



Mise au point à distance, principe





Espace utilisateur : gdbserver

- Cas le plus simple pour le développement applicatif
- Fonctionne avec un lien Ethernet ou série RS-232 entre cible et hôte
- Exemple :

```
# gdbserver 192.168.3.109:9999 myprog ← cible  
Process myprog created; pid = 12810
```

```
$ arm-linux-gdb myprog ← hôte  
GNU gdb Red Hat Linux (6.7-2rh)
```

...

```
(gdb) target remote 192.168.3.50:9999  
Remote debugging using 192.168.3.50:9999
```



- Historiquement développé par LynsysSoft, repris par Wind River
- Peu de mise à jour sur la version « libre »
- Patch finalement accepté dans Linux « mainline » pour 2.6.26 => Kernel hacking/KGDB:
- Utilisation de `vmlinux` (non compressé) sur le poste de développement
- Options coté cible :
 - `kgdboc=ttyAMA0,115200` ← Port série (cible)
 - `kgdboe=@192.168.0.1/,@192.168.0.2`
 - `kgdbwait`

- ATTENTION: kgdboe n'est plus *mainline* en l'état
- Coté GDB hôte :

```
$ gdb vmlinux
```

```
...
```

```
(gdb) b sys_sync => arrêt sur sync
```

```
(gdb) b panic
```

```
(gdb) target remote /dev/ttyS0
```



Port série (hôte)

- Mise au point de modules .ko => cas le plus fréquent

- Insérer le module par modprobe ou insmod

- Obtenir les valeurs des sections :

```
$ cat /sys/module/helloworld/sections/.text  
0xbf000000
```

- Coté GDB, ajout des adresses des sections

```
(gdb) add-symbol-file helloworld.ko 0xbf000000 -  
s .data 0xbf00052e -s .bss 0xbf000660 -s .rodata  
0xbf0000ac
```

- Debug module_init() avec version spéciale de GDB => *pending breakpoint*

```
(gdb) set solib-search-path
```

```
(gdb) b my_module_init
```

```
(gdb) Breakpoint 1 (my_module_init) pending.
```



Analyse de « OOPS » noyau 1/2

- Voir le fichier `Documentation/oops-tracing.txt`
- Utiliser si possible une console série pour enregistrer le oops
- Principe :
 - Récupérer la valeur du « program counter » (EIP) => « unable to handle...at virtual adresse XXX »
 - Grâce au fichier `/proc/kallsyms`, récupérer le nom et l'adresse de la fonction incriminée (souvent indiqué dans le oops)
 - Calculer l'adresse de l'instruction par
$$\text{EIP} - \text{adresse_fonction}$$
 - Désassembler avec `objdump -D` (et/ou `-S`) pour retrouver la ligne

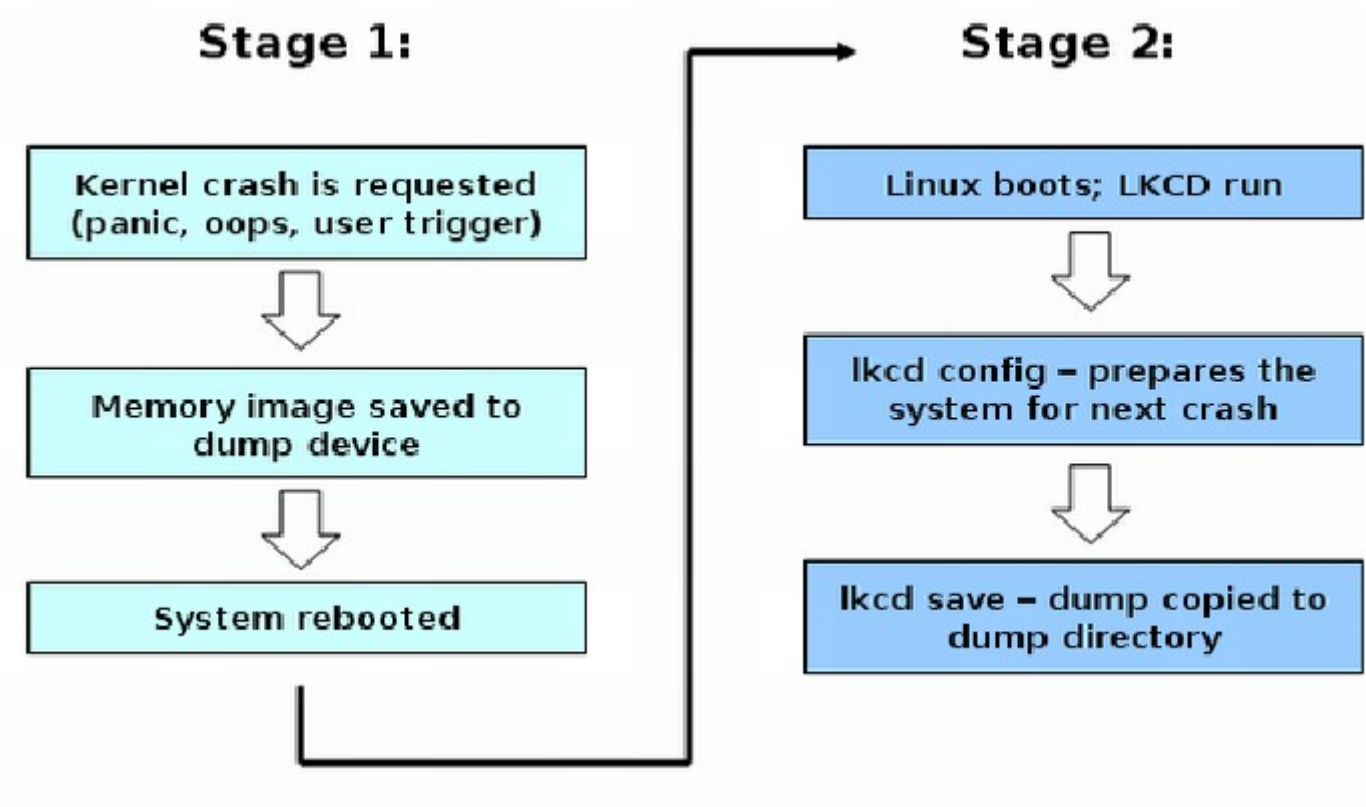


Analyse de « OOPS » noyau 2/2

- On utilise GDB sur le noyau courant
- Utiliser /proc/kcore comme fichier core

```
# gdb vmlinux /proc/kcore
(gdb) add-symbol-file helloworld.ko ...
(gdb) list nom_de_fonction
```
- Voir la suite sur :
<http://www.scribd.com/doc/19294114/Debugging-Kernel-Oops>

- LKCD = Linux Kernel Crash Dump
- Enregistrement des données lors d'un crash pour analyse « post mortem »
- Au reboot, copie des données sur une partition dédiée ou bien sur un serveur
- Installation détaillée et test sur :
<http://www.dedoimedo.com/computers/lkcd.html#mozTocId759823>





Trace en espace noyau

- Comprendre le comportement du système => monitoring
- Analyse de problèmes de performances (pas de crash) => systèmes temps réel, ou parallèles
- Enregistrement d'une grande quantité de données => optimisation nécessaire
- Types d'événements enregistrés :
 - Appels systèmes
 - Fonctions de traitement d'interruption
 - Ordonnancement
 - Réseau



LTT: Linux Trace Toolkit

- Utilitaire de trace (statique) le plus célèbre pour le noyau Linux => patch pour le noyau 2.4
- Travaux publiés en 1998/2000 par Karim Yaghmour: <http://www.linuxjournal.com/article/3829>
- Deux parties :
 - Enregistrement des données :

```
$ trace 60 result.dat
```
 - Visualisation texte ou graphique (GTK) :

```
$ trace result.dat  
$ traceview trace.dat
```



LTTng, présentation 1/2

- LTTng = nouveau LTT (Mathieu Desnoyers, EP Montréal, EficiOS), depuis 2005
- Projet très actif, nombreuses publications et conférences
- Deux parties: enregistrement + visualisation (LTTv)
- Instrumentation statique basée sur *Markers* puis *Tracepoints* dans le noyau Linux
- Instrumentation dynamique (Kprobes) depuis 2009
- LTTng n'est pas dans le noyau « mainline » => patch nécessaire



LTTng, présentation 2/2

- Architecture portable => fonctionne sur ARM/OMAP3
- « Per-CPU buffer » => pas de problème de verrouillage mémoire en SMP
- Trace clock: Lecture directe des « timestamps » CPU => précision (monotone) mais adaptation nécessaire sur certains CPU
- Selon l'auteur, il y a peu de modifications sur le coeur du noyau, donc devrait être « mainline »
- Voir présentations CELF2009 + ELC2010 + présentation vidéo



The screenshot displays the Linux Trace Toolkit Viewer interface. The top section shows 'Traceset statistics' for a traceset named 'traceset statistics'. The statistics are as follows:

- statistics summed : 1
- cpu time : 65.495830804
- events count : 1994693
- elapsed time (includes per process waiting time) : 1003.526752178
- cumulative cpu time (includes nested routines and modes) : 16.318190440

The middle section shows a 'Resource' view with a vertical axis and a horizontal axis, displaying CPU usage for various resources like CPU0, IRQ 37, and SOFTIRQs.

The bottom section shows a 'Process' table with columns: Process, Brand, PID, TGID, PPID, CPU, Birth sec, Birth nsec, TRACE. The processes listed are:

Process	Brand	PID	TGID	PPID	CPU	Birth sec	Birth nsec	TRACE
modem-manager		506	506	1	0	0	331177016	0
console-kit-dae		522	522	1	0	0	331185492	0
/usr/sbin/gdm-binary		16610	16610	1	0	18	720745954	0
/usr/lib/gdm/gdm-simple-slave		16614	16614	16610	0	18	910856978	0
/usr/bin/Xorg		16616	16616	16614	0	19	103704608	0
/usr/sbin/gdm-binary		16617	16617	16610	0	19	217090070	0

Below the process table is an event log table with columns: Trace, Tracefile, CPUID, Event, Time (s), Time (ns), PID, Event Description. The events shown are:

Trace	Tracefile	CPUID	Event	Time (s)	Time (ns)	PID	Event Description
/home/x0129185/trace1	rcu	0	tree_callback	19	205643352	506	rcu.tree_callback: 19.205643352 (/home/x0129185/trace1/rcu_0), 506, 506, modem-manager, , 1, 0
/home/x0129185/trace1	rcu	0	tree_callback	19	205645978	506	rcu.tree_callback: 19.205645978 (/home/x0129185/trace1/rcu_0), 506, 506, modem-manager, , 1, 0
/home/x0129185/trace1	rcu	0	tree_callback	19	205649096	506	rcu.tree_callback: 19.205649096 (/home/x0129185/trace1/rcu_0), 506, 506, modem-manager, , 1, 0
/home/x0129185/trace1	rcu	0	tree_callback	19	205653752	506	rcu.tree_callback: 19.205653752 (/home/x0129185/trace1/rcu_0), 506, 506, modem-manager, , 1, 0
/home/x0129185/trace1	rcu	0	tree_callback	19	205656524	506	rcu.tree_callback: 19.205656524 (/home/x0129185/trace1/rcu_0), 506, 506, modem-manager, , 1, 0

The bottom of the window shows the 'Time Frame' settings: start: 19 s 187915621 ns, end: 19 s 220337893 ns, Time Interval: 0 s 32422272 ns, Current Time: 19 s 205638509 ns.



LTTng, conclusions

- Outil puissant, portable
- Communauté dynamique, nombreuses publications
- Installation / configuration / utilisation complexe (ajout de points de traces dans le code + modification fichier XML)
- Toujours pas « mainline »
- Rapprochement avec Ftrace => ex: LTTv en cours d'adaptation pour utiliser les résultats de Ftrace (?)



Ftrace, présentation 1/2

- Initialement « Function Tracer » (2.6.27)
- « Unification » des outils de trace du noyau
- Capable de tracer entre autres : latences, IRQ, Context switch, graphes de fonctions, etc.
- Possibilité d'ajouter des « plugins »
- Standard dans le noyau:
 - valider `Kernel hacking/Tracers`
 - choisir les traceurs dans le menu Tracers
- <http://lwn.net/Articles/322666>
- Voir `Documentation/trace/ftrace.txt`



Ftrace, présentation 2/2

- Interface basée sur DebugFS
 - # mount -t debugfs nodev /sys/kernel/debug
- Pilotage/configuration par fichier virtuel
- /proc/sys/kernel/ftrace_enabled: active ou non les traces (1/0)
- Principe :
 - Sélectionner le traceur
 - Définir un filtre
 - Activer/désactiver la trace
 - Exploiter le résultat

Ftrace, principales entrées

- Chemin d'accès relatif à `/sys/kernel/debug/tracing`
 - `available_tracers`: traceurs disponibles (nop, function, function_graph, ...)
 - `current_tracer`: traceur courant, contient une des valeurs précédentes
 - `tracing_enabled`: 1/0
 - `trace`: résultat lisibles (texte)
 - `buffer_size_kb`: taille du tampon circulaire
 - `available_filter_functions`: fonction traçables => environ 25000 entrées !
 - `available_events`: événements disponibles



Ftrace, exemple 1/2

- Exemple de pilote `mydriver1.ko`
- Fonctions `open()`, `read()`, `write()`, `release()`
=> `printk`
`# echo 'mydriver1_*' > set_ftrace_filter`
`# cat set_ftrace_filter`
`mydriver1_release`
`mydriver1_open`
`mydriver1_write`
`mydriver1_read`
`# echo 1 > tracing_on`



Ftrace, exemple 2/2

```
# < /dev/mydriver1 ← accès pilote
```

```
# cat trace
```

```
# tracer: function
```

```
#
```

```
#          TASK-PID      CPU#      TIMESTAMP  
FUNCTION
```

```
#          | |          |          |          |
```

```
          bash-6842  [000]  2245367.171290:  
mydriver1_open <-chrdev_open
```

```
          bash-6842  [001]  2245399.996460:  
mydriver1_open <-chrdev_open
```

```
          bash-6842  [001]  2245399.996478:  
mydriver1_release <-__fput
```



Ftrace, affichage des latences

```
# echo 1 > options/latency-format
# cat trace
# ...
# latency: 0 us, #2/2, CPU#0 | (M:desktop VP:0, KP:0, SP:0 HP:0 #P:2)
# -----
# | task: -0 (uid:0 nice:0 policy:0 rt_prio:0)
# -----
#           _-----=> CPU#
#           / _-----=> irqs-off
#           | / _-----=> need-resched
#           || / _----=> hardirq/softirq
#           ||| / _--=> preempt-depth
#           |||| / _--=> lock-depth
#           |||||/      delay
# cmd      pid  ||||| time  | caller
#  \      /    |||||  \  |  /
      bash-7361 0.... 4100574us+: mydriver1_open <-chrdev_open
      bash-7361 0..... 4100600us : mydriver1_release <-__fput
```



Ftrace, fonctions avancées

- `set_ftrace_pid`: trace un PID donné
 - # `echo $$ > set_ftrace_pid`
 - # `cat trace`
 - # `echo -1 > set_ftrace_pid`
- `function_graph`: graphe d'appel des fonctions
 - Trace l'entrée et la sortie d'une fonction
 - Calcul de temps d'exécution (DURATION)
 - Affichage de la hiérarchie des appels par { et }
 - # `echo function_graph > current_tracer`
 - # `cat trace`



Ftrace, function_graph

#	CPU	DURATION	FUNCTION CALLS
1)		1.015 us	_spin_lock_irqsave();
1)		0.476 us	internal_add_timer();
1)		0.423 us	wake_up_idle_cpu();
1)		0.461 us	_spin_unlock_irqrestore();
1)		4.770 us	}
1)		5.725 us	}
1)		0.450 us	mutex_unlock();
1)	+	24.243 us	} ← délai > 10 µs
1)		0.483 us	_spin_lock_irq();
1)		0.517 us	_spin_unlock_irq();
1)			prepare_to_wait() {
1)		0.468 us	_spin_lock_irqsave();
1)		0.502 us	_spin_unlock_irqrestore();
1)		2.411 us	}
1)		0.449 us	kthread_should_stop();
1)			schedule() {



Ftrace, trace_printk() 1/2

- La fonction `printk()` est très utilisée pour la mise au point mais très intrusive
 - Appel scheduler
 - Désactivation IRQ
 - Spinlocks
 - ...
- La fonction `trace_printk()` est équivalente mais utilisable dans tous les contextes: scheduler, NMI, IRQ => affichage dans trace
- Utilise quelques dizaines de μ s au lieu de quelques ms :)



Ftrace, trace_printk() 2/2

- Ajout à une fonction:

```
trace_printk("read foo %d out of bar
%p\n", bar->foo, bar);
```

- Affichage trace :

```
#          TASK-PID    CPU#    TIMESTAMP  FUNCTION
#          | |        |         |         |
<...>-10690 [003] 17279.332920: : read foo 10 out of bar
ffff880013a5bef8
```

- Affichage function_graph :

```
3)          | do_one_initcall() {
3)          |     /* read foo 10 out of bar ffff88001191bef8 */
3) 4.221 us | }
```



Ftrace, trace_marker 1/2

- Accès depuis l'espace utilisateur

```
# echo hello world > trace_marker
```

- Visible dans trace

```
bash-6842 [000] 2245230.244514: 0: hello world
```

- Utilisation des traces sélectives en espace utilisateur

```
req.tv_sec = 0;
```

```
req.tv_nsec = 1000;
```

```
write(marker_fd, "before nano\n", 12);
```

```
nanosleep(&req, NULL);
```

```
write(marker_fd, "after nano\n", 11);
```

```
write(trace_fd, "0", 1);
```



- Résultat avec function_graph

```
# CPU DURATION          FUNCTION CALLS
# |      | |          | | | |
0) |      | |          | /* before nano */
0) |      | |          | kfree() {
0) 0.475 us |          |   __phys_addr();
0) 2.062 us |          | }
0) 0.608 us |          | inotify_inode_queue_event();
...
0) |      | |          | /* after nano */
0) |      | |          | kfree() {
0) 0.486 us |          |   __phys_addr();
```



Ftrace, activation par programme

- Dans un module on peut insérer :

```
if (test_for_error())  
    tracing_off();
```
- Equivalent à :

```
# echo 0 > tracing_on
```



Ftrace, autres fonctions

- Profiling: `function_profile_enabled`
- Pile: `stack_tracer_enabled`
- Dump sur OOPS : `ftrace_dump_on_oops`
- ...
- Voir :
 - <http://lwn.net/Articles/365835/>
 - <http://lwn.net/Articles/366796/>
 - <http://lwn.net/Articles/370423/>
 - `Documentation/trace/ftrace.txt`



Résultat de mesure sched_wakeup

	5051	5957	5002	5002	5004	5014
	5052	10215	5001	5005	5003	5013
	5051	9940	5002	5017	5003	5013
	5052	9810	5003	5015	5003	5015
	5051	10021	5001	5007	5004	5015
	5052	18174	5002	5017	5003	5015
	5052	9424	11584	7409	5004	5016
	5051	18528	5002	5018	5004	5018
	5052	9740	5002	5018	5003	5019
	5051	9869	5002	5007	5003	5012
	5052	9991	5001	5007	5003	5013
	5051	9717	5002	5017	5004	5014
	5052	9800	5002	5482	5002	5015
	5052	9694	5001	5014	5004	5014
	5051	9840	5002	5015	5003	5016
	5052	9782	5001	5007	5003	5020
	5053	9729	5003	5013	5005	5019
	5052	19497	5002	17132	5004	5014
	5052	12273	5002	5036	5004	5015
moyenne	5075.28	10602.89	5107.34	5698.6	5003.92	5014.35
medianne	5052	9733	5002	5015	5003	5014
max	7109	19497	11584	17132	5029	5021
min	5030	5074	5001	5001	5002	5004
écart-type	207.65	3251.39	763.11	2119.53	3.29	2.63
légende	5000µs -> 5100µs	5100µs -> 6000µs	6000µs -> 10000µs	> 10000µs		



Ftrace, trace-cmd 1/3

- Interface pour l'utilisation de Ftrace
- Evite de manipuler les entrées dans /sys
- A compiler à partir du dépôt Git
- Voir: <http://lwn.net/Articles/341902>
- Exemple :

```
# ./trace-cmd record -e all ls /bin
```

- Résultat :

```
# ./trace-cmd report
version = 0.5
cpus=4
    trace-cmd-10995 [003] 235302.428904: kmem_cache_alloc:
call_site=ffffffff810df1e4 ptr=0xffff880027585a68 bytes_req=168
bytes_alloc=168 gfp_flags=GFP_KERNEL
```



- Autre exemple d'enregistrement :

```
# trace-cmd record -e irq ping www.google.com
```

- Résultats :

```
# trace-cmd report
```

```
ping-26778 [001] 113159.964853: irq_handler_entry:  
irq=44 name=i915
```

```
ping-26778 [001] 113159.964861: irq_handler_exit:  
irq=44 ret=handled
```

```
ping-26778 [001] 113159.966428: softirq_entry:          vec=3  
[action=NET_RX]
```

```
ping-26778 [001] 113159.966461: softirq_exit:          vec=3  
[action=NET_RX]
```

```
dnsmasq-31639 [001] 113159.966545: softirq_entry:  
vec=3 [action=NET_RX]
```

```
dnsmasq-31639 [001] 113159.966549: softirq_exit:  
vec=3 [action=NET_RX]
```



Ftrace, trace-cmd 3/3

```
# trace-cmd
```

```
trace-cmd version 1.1.0-rc1
```

```
usage:
```

```
trace-cmd [COMMAND] ...
```

```
commands:
```

```
record - record a trace into a trace.dat file
```

```
start - start tracing without recording into a file
```

```
extract - extract a trace from the kernel
```

```
stop - stop the kernel from recording trace data
```

```
reset - disable all kernel tracing and clear the trace buffers
```

```
report - read out the trace stored in a trace.dat file
```

```
split - parse a trace.dat file into smaller file(s)
```

```
listen - listen on a network socket for trace clients
```

```
list - list the available events, plugins or options
```

```
restore - restore a crashed record
```

```
stack - output, enable or disable kernel stack tracing
```



Ftrace, conclusions

- Utilisation simple car intégré au noyau mainline => multiplateforme
- A la fois dynamique et statique
- Intégré à des extensions comme PREEMPT-RT
- Peu consommateur de ressources => peut tourner en permanence
 - Réglage de `buffer_size_kb`
 - En cas de problème visible on peut sauver le tampon courant et analyser
 - A partir des données on peut tracer des courbes

- Approche différente => statistique
- Proche de prof (UNIX) et gprof (GNU/Linux)
- Comptage des fonctions les plus utilisées
- Fonctionne en espace utilisateur et noyau
- Prise en compte du multi-thread pour une application
- Nécessite des adaptations (simples) si l'on veut « profiler » une portion précise de code
- <http://oprofile.sourceforge.net>

- Démarrage du démon et enregistrement

```
# opcontrol --start-daemon
```

```
# opcontrol --start
```

```
./myprog
```

- Arrêt de l'enregistrement

```
# opcontrol --stop
```

- Vidage du tampon

```
# opcontrol --dump
```

- Arrêt du démon

```
# opcontrol --stop-daemon
```

- Affichage des résultats

```
$ oprofile -l -d
```

- Sortie en XML

```
$ oprofile -l -d -X > myprog.xml
```

- Utilisation de XSLT pour formater les données

```
$ xsltproc oprofile_test.xsl myprog.xml  
> myprog.html
```



QEMU



QEMU, domaines d'application

- Émulation de périphériques + hyperviseur KVM = virtualisation
 - exemple : partitionnement de serveur
- Émulation processeur avec TCG (Tiny Code Generator) = user mode
 - exemple : développement d'application embarquée
- Émulation complète avec TCG = **simulation**
 - exemple : développement de système embarqué

Simuler du matériel en langage C

- Environnement entièrement logiciel ou partiellement = cosimulation
- Facilités en environnement logiciel
 - accès au fonctionnement interne du processeur
 - interaction avec des outils logiciels

- Matériel non disponible : obsolète, trop cher, non transportable, en développement, SDK...
- Debug système non intrusif (*)
- Tests automatisés avec stimuli extérieurs (*)
- Couverture de test non intrusive (*)
- Arrêt/reprise de session de mise au point

Dans tous les cas, les contraintes, incidents et doutes matériels n'existent plus.

(*) détaillé plus loin



QEMU, mise en place (1)

- Existant ?
 - processeurs : x86, ARM, MIPS, SH4, PPC, Microblaze, etc...
 - cartes : environ 30 préconfigurations
 - périphériques : variés
- Suffisamment complet / proche de la réalité ?
 - approche feignante : on ne simule que ce dont le système a besoin

- Besoin ?
- Classification de l'effort restant :
 - 0) le composant est déjà modélisé
 - 1) il existe un modèle proche que l'on adapte
 - 2) le composant n'est pas très important -> "fake" (exemple : détection obligatoire)
 - 3) le composant est nouveau ou n'a jamais été modélisé



QEMU dans le « workflow »

Une fois mis en place,

3 exemples d'utilisations



QEMU, tests automatisés avec stimuli extérieurs

- Bus de communication (série, ethernet, USB)

le "**tout logiciel**" ouvre des possibilités :

- Commandes d'allumage/extinction
- QMP = système de communication bidirectionnelle au format JSON
 - influencer sur l'exécution (hotplug, boutons, défauts matériels...)
 - statut et événements (erreurs I/O, watchdog...)



QEMU, débuser une régression (cas idéal)

- Écriture de scripts de tests au cours du développement
- Utilisables en intégration continue car facilement automatisables
 - tests à chaque commit



QEMU, débuser une régression (cas réel)

- Régression avérée
 - puis écriture du test correspondant
- Heureusement, le programme est versionné avec Git ou un gestionnaire compatible (svn, hg)
- « **git bisect run** <script> » trouve le commit fautif automatiquement
 - le script compile, exécute QEMU et détecte l'erreur
 - codes de retour du script :
 - 0 = OK
 - 125 = non testable
 - 1-124, 126-127 = régression
 - 128-255 = abandon



QEMU, debug système non intrusif (1)

- Environnement réel : debug avec sonde JTAG
- Environnement simulé : stub gdb dans Qemu
- GDB ouvre un seul binaire
 - bootloader
 - noyau
 - init
 - application bare board
- Application userland nécessite « OS awareness »
 - dans ce cas, gdbserver est nécessaire
- N'importe quel front-end GDB est utilisable



QEMU, debug système non intrusif (2)

- Serveur pour GDB accessible par l'option « -gdb »
 - Raccourci « -Ss » = attente de connexion gdb sur le port 1234
- Connexion GDB = commande « target remote »

```
 ${PREFIX}gdb $BIN \  
  --eval-command="target remote localhost:1234" \  
  --eval-command="hbreak *0x$ADDR"
```
- Si le programme n'est pas le premier à s'exécuter au démarrage du système, point d'arrêt matériel : « hbreak » au lieu de « break ».

```
 ADDR=$( ${PREFIX}nm $BIN | \  
  sed -n "s,^\([^ ]*\).*$FUNC$, \1,p")
```



QEMU, couverture de test non intrusive

- Particulièrement utile en certification DO-178
 - Méthode classique d'analyse de la couverture de test
 - système instrumenté testé sur cible réelle
 - Nouvelle approche
 - système réel testé sur cible instrumentée
- = projet Couverture
- Qemu fournit des traces
 - xcov analyse les traces
 - synthèse au niveau source ou objet



Couverture, rapport général

XCOV coverage report

Coverage level: branch

Trace Filename	Program	Date	Tag
explore.trace	obj/powerpc-elf/explore	2011-03-23 18:19:14	

covered not covered visual summary

	total nb of lines				
Total	337 lines	294 lines (87 %)	19 lines (6 %)	24 lines (7 %)	

covered not covered visual summary

Filename	total nb of lines				
explore.adb	19 lines	16 lines (84 %)	2 lines (11 %)	1 lines (5 %)	
b_explore.adb	25 lines	24 lines (96 %)	0 lines (0 %)	1 lines (4 %)	
actors.adb	6 lines	3 lines (50 %)	0 lines (0 %)	3 lines (50 %)	
actors.ads	3 lines	3 lines (100 %)	0 lines (0 %)	0 lines (0 %)	
queues.adb	27 lines	19 lines (70 %)	6 lines (22 %)	2 lines (7 %)	
links.adb	39 lines	33 lines (85 %)	3 lines (8 %)	3 lines (8 %)	
geomaps.adb	15 lines	15 lines (100 %)	0 lines (0 %)	0 lines (0 %)	
robots_devices.ads	5 lines	5 lines (100 %)	0 lines (0 %)	0 lines (0 %)	
robots.adb	43 lines	36 lines (84 %)	4 lines (9 %)	3 lines (7 %)	
robots.ads	2 lines	2 lines (100 %)	0 lines (0 %)	0 lines (0 %)	



Couverture, rapport détaillé

```
44 +   procedure Pop (Item : out Data_Type; Q : in out Queue) is
45 .   begin
46 !     if Empty (Q) then
47 -       raise Program_Error;
48 .     end if;
49 .
50 +     Item := Q.Items (Q.Front);
51 !     if Q.Front = Q.Items'Last then
52 +       Q.Front := Q.Items'First;
53 .     else
54 !       Q.Front := Q.Front + 1;
55 .     end if;
56 +     Q.Size := Q.Size - 1;
57 +     end Pop;
--
```



Consolider un environnement de développement industriel ?



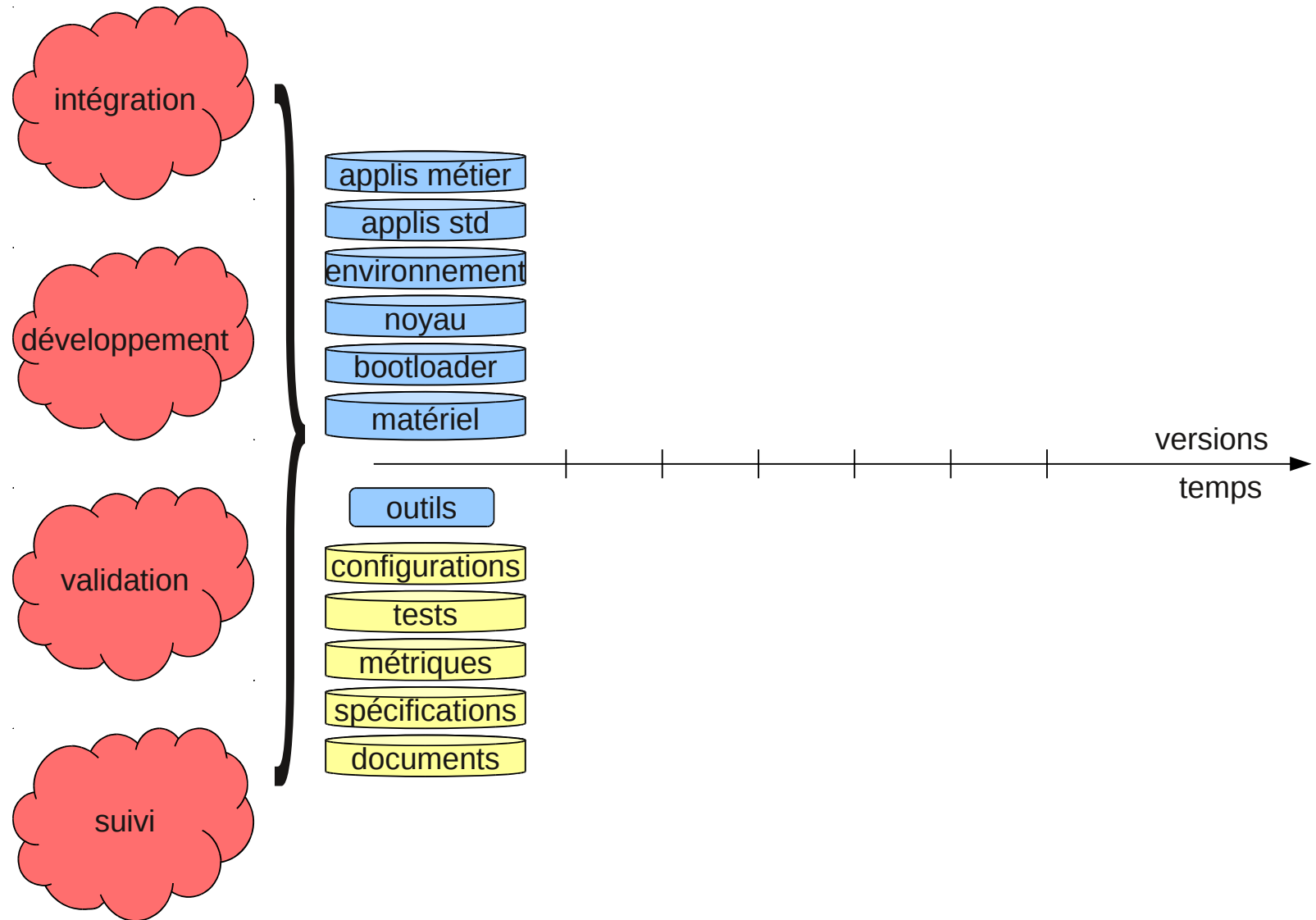
Environnement industriel, besoins

- Chaque projet définit des règles d'organisation pour améliorer l'efficacité du développement, de la coordination et du suivi
- Mais cela prend du temps et rarement suffisant
- Certains outils sont indisponibles ou mal connus
- Peu de processus sont automatisés
- L'organisation, l'automatisation et la fourniture d'outils ne sont pas dans le cœur de métier

Il faut **capitaliser** sur la construction d'un environnement projet efficace



Environnement industriel, concept





Environnement industriel, apports

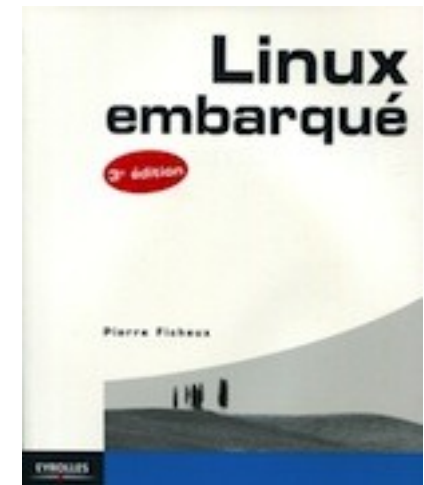
- Création rapide de projets types
- Collecte des licences
- Visualisation macroscopique de l'état du projet
- Visualisation de métriques et évolutions
- Releases et reproductibilité
- Profils de configurations alternatives (ex : debug)
- Assistance à la documentation (ex : changelog)
- Aggrégation et classification de documents
- Classification des outils compatibles
- Intégration d'outils
- Campagnes de validation

- Linux embarqué version 3 :

<http://www.eyrolles.com/Informatique/Livre/linux-embarque-9782212124521>

- Busybox: <http://www.busybox.net>
- Buildroot: <http://buildroot.uclibc.org>
- OE: <http://www.openembedded.org>
- KGDB: <https://kgdb.wiki.kernel.org>
- Ftrace :

- <http://lwn.net/Articles/365835/>
- <http://lwn.net/Articles/366796/>
- <http://lwn.net/Articles/370423/>
- `Documentation/trace/ftrace.txt`



- Qemu : <http://wiki.qemu.org>
- Contribuer à Qemu :
http://ingenierie.openwide.fr/content/download/2472/18834/file/Qemu_article_technique.pdf
- Couverture : <http://www.projet-couverture.com>
- Fully automated bisecting :
<https://lwn.net/Articles/317154>



Questions ?