

# Outils de mise au point pour Linux embarqué

Pierre FICHEUX  
pierre.ficheux@openwide.fr

Avril 2011

- Présentation d'Open Wide
- Rappels sur l'architecture d'un OS Linux embarqué
- Mise au point: GDB/KGDB
- Instrumentation: Ftrace
- QEMU
  - Présentation, Test de non régression, mise au point
- OpenOCD

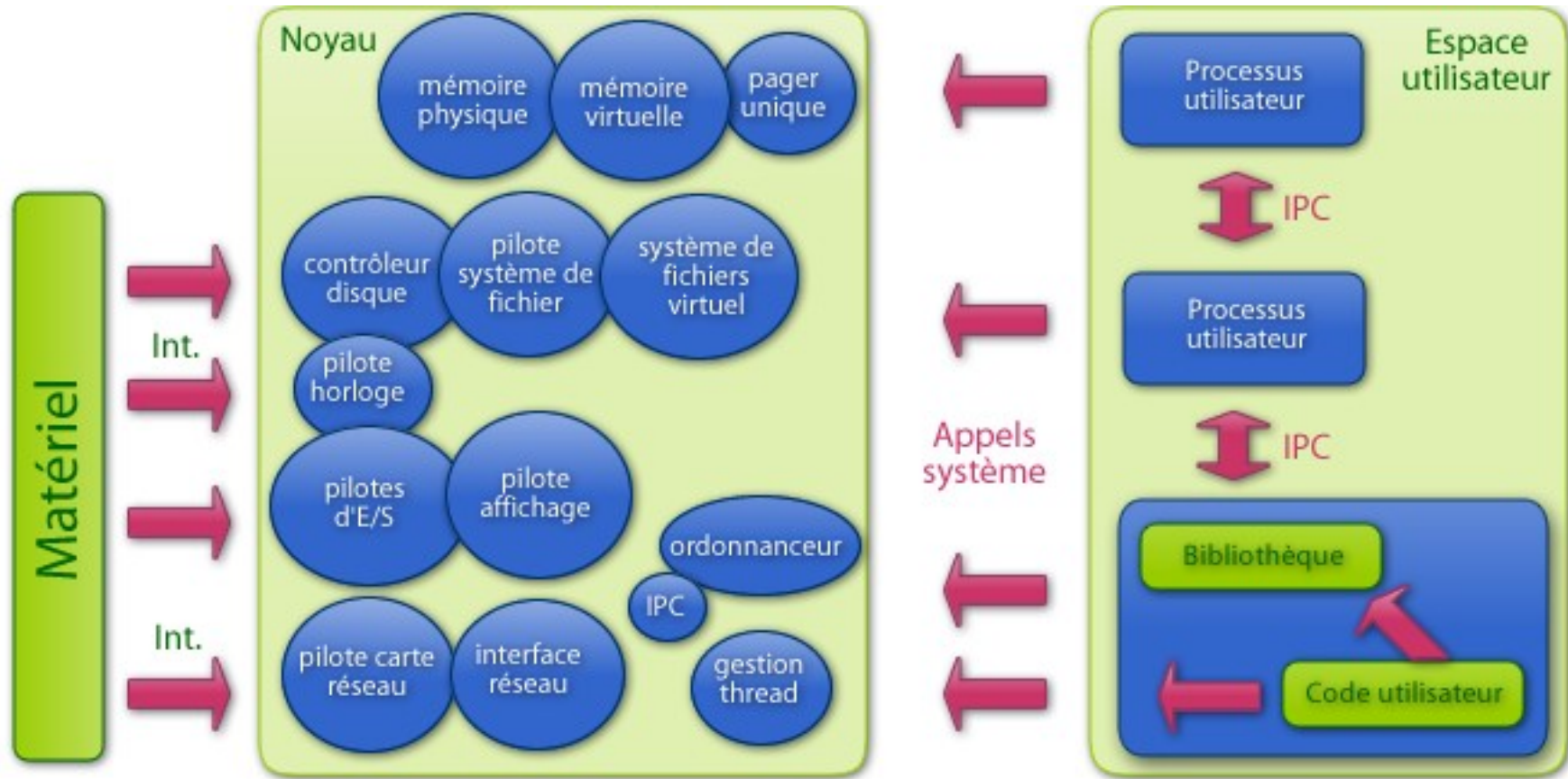
# Présentation Open Wide

- SSII/SSL 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



- 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

# Mise au point / instrumentation

# Mise au point ou instrumentation ?

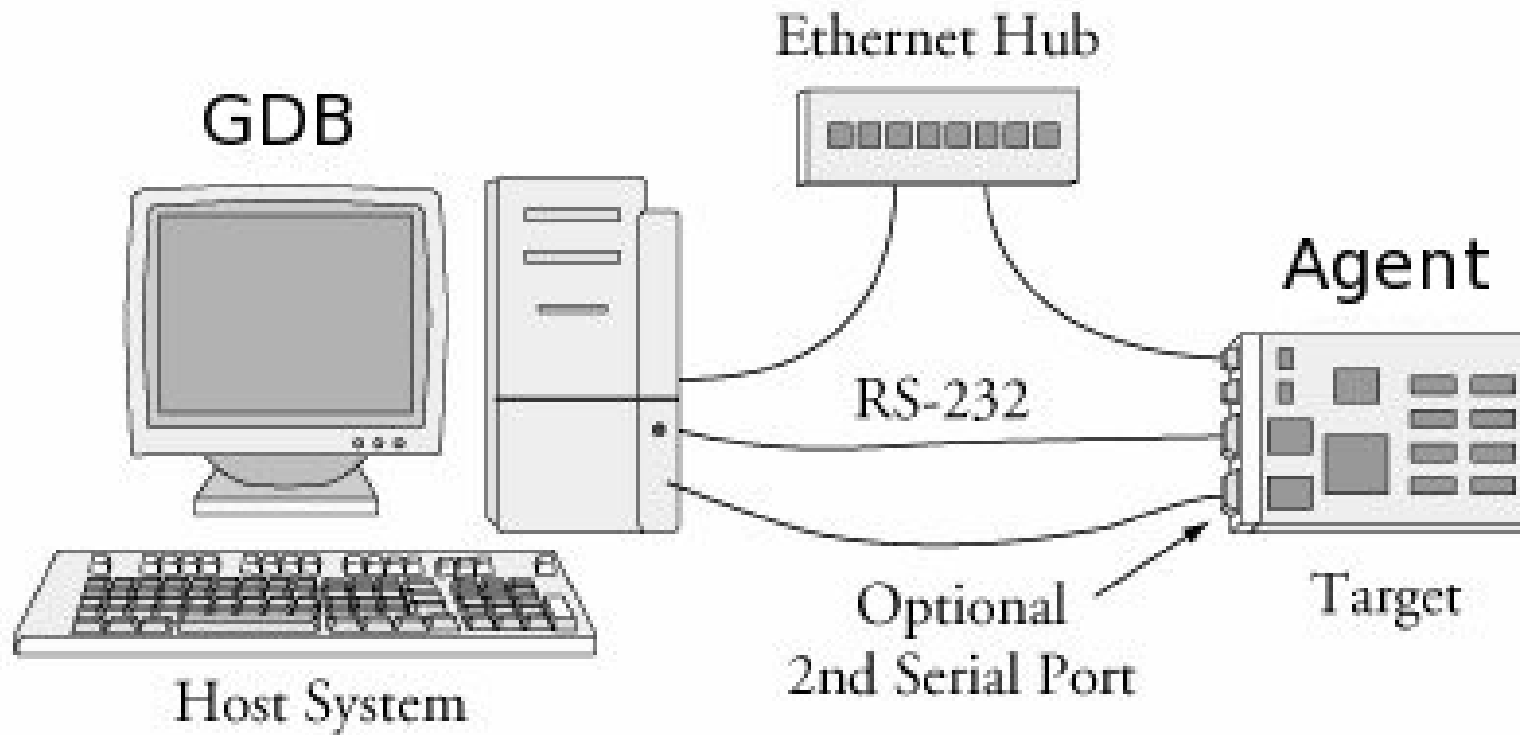
- Mise au point: résolution d'un problème de fonctionnement
  - Valgrind: pour problèmes de mémoire
  - GDB / KGDB
  - strace / ltrace
  - Analyse de crash (LKCD / Ksymoops)
- Instrumentation: analyse préventive, profiling
  - Traces en espace noyau: LTTng, Ftrace, etc.
  - Profiling OProfile (noyau/utilisateur)
  - 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* depuis 2.6.35
- 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 /user/my_module_2.6...  
(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>

- 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

# 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

- 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() {
```

- 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  $\mu$ s au lieu de quelques ms :)

- 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 | }
```

- 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		

- Utilisation simple car intégré au noyau mainline => multiplateforme
- 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 en mode texte on peut tracer des courbes !

# QEMU

- Emulateur de matériel initialement développé par Fabrice Bellard, diffusé sous GPLv2
- Exécuté dans l'espace utilisateur de Linux
- Permet d'émuler diverses architectures: x86, PPC, ARM, etc.
- Emulation de carte « complète », y compris (parfois) la flash NOR
- Réseau en mode « user » ou « bridge » (TUN/TAP)
- Désormais, large communauté avec dépôt Git sur <http://git.savannah.gnu.org/cgiit/qemu.git>

# 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 (carte complète)

## 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

(\*) détaillé plus loin

# QEMU, tests automatisés avec stimuli extérieurs

- Bus de communication (série, Ethernet, USB)
- Le *tout logiciel* ouvre des possibilités :
  - Commandes d'allumage/extinction
  - QMP (Qemu Monitor Protocol) = 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
- Pré-requis: 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 : mise au point avec sonde JTAG
- Environnement simulé : « serveur » gdb dans QEMU
- GDB ouvre un seul binaire
  - Bootloader
  - Noyau Linux
  - init
  - application « bare board »
- Une application espace utilisateur nécessite un lien avec l'OS => gdbserver est nécessaire
- N'importe quel *front-end* GDB est utilisable

- Utilise des options particulières de QEMU
  - S : « gel » du CPU au démarrage
  - s : serveur GDB
- Autre solutions disponibles : KGDB ou sonde
- QEMU est un émulateur de matériel DONC on peut mettre au point un autre programme que le noyau Linux
- Commande QEMU à utiliser :

```
$ qemu-system-arm -M versatilepb -m 64 -kernel  
zImage -initrd rootfs.cpio -append  
"console=ttyAMA0" -s -S
```

- Coté hôte: on utilise la version nom compressée du noyau (fichier `vmlinux`) :
  - \$ `arm-linux-gdb vmlinux`
  - ...
  - (gdb) `b start_kernel`
  - (gdb) `target remote localhost:1234`
- QEMU utilise par défaut le port TCP 1234
- Fonctions remarquables déjà évoquées :
  - `start_kernel()`, `sys_sync()`, `panic()`, ...

# QEMU, mise au point U-Boot

- Procédure similaire à la mise au point du noyau
- L'image binaire U-Boot est utilisée avec l'option `-kernel`

```
$ qemu-system-arm -M versatilepb -m 64 -kernel  
u-boot.bin -nographic -s -S
```

- Coté hôte, utiliser la version ELF (fichier `u-boot`), seule compatible avec le débogueur croisé.

```
$ arm-linux-gdb u-boot
```

...

```
(gdb) b start_armboot
```

```
(gdb) target remote localhost:1234
```

# QEMU, utilisation de KGDB

- Plus d'options -s -S !
- Utilisation d'un noyau compatible KGDB

```
$ qemu-system-arm -M versatilepb -m 64  
-kernel zImage -initrd rootfs.cpio -append  
"kgdboc=ttyAMA0 kgdbwait" -serial pty  
char device redirected to /dev/pts/5
```

- Coté hôte :

```
$ arm-linux-gdb vmlinux
```

```
...
```

```
(gdb) target remote /dev/pts/5
```

Redirection console



# 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 %)	

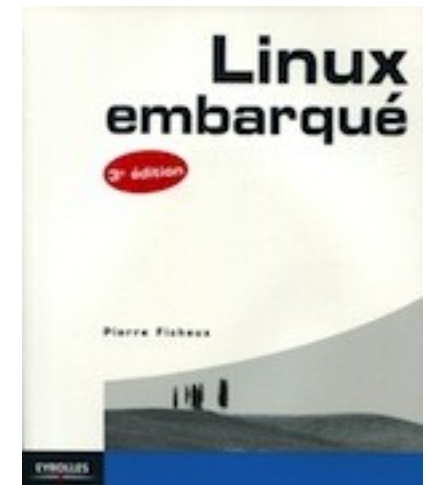
covered not covered visual summary

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

- 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](http://ingenierie.openwide.fr/content/download/2472/18834/file/Qemu_article_technique.pdf)
- QMP sur: <http://wiki.qemu.org/QMP>
- Couverture : <http://www.projet-couverture.com>
- Open DO: <http://www.open-do.org>
- Fully automated bisecting :  
<https://lwn.net/Articles/317154>
- <http://www.miximum.fr/methodes-et-outils/79-debusquer-une-regression-avec-git-bisect>

Questions ?