Formation au Noyau Linux

Jérôme Pouiller <j.pouiller@sysmic.org>

Cinquième partie V

Debuguer

printk

- `printk` affiche les information sur la console et les copie dans un buffer. Il est possible d'afficher ce buffer avec `dmesg`
- Il est possible d'utiliser directemenr `printk` en specifiant le degré d'importance de l'information :

```
printk(KERN_ALERT "Aie\n");
```

- Notez l'absence de virgule entre `KERN_ALERT` et la chaîne
- Néanmoins, il est maintenant conseillé d'utiliser les macros `pr_cont`, `pr_develpr_debug`, `pr_info`, `pr_notice`, `pr_warning`, `pr_err`, `pr_crit`, `pr_alert`, `pr_emerg`
- Existe aussi suffixé de `_once` pour n'être affiché que la première fois.
- `print_hex_dump_bytes` et `print_hex_dump` affichent un dump en hexadécimal d'un buffer. La seconde est la forme générale qui permet de faire à peu près la même chose que la commande `hexdump`

printk

- `printk` est compatible avec la norme C99 et ajoute des extensions pour pretty-printer certains pointeurs (`%p`). En particulier :
  - `%pF` pour résoudre le nom d'une fonction du segment de text
  - `%pUB` et `%pUL` pour afficher les UUID ou les GUID
  - `%pM` pour afficher des adresses MAC
  - `%pI4` pour affichier des adresses IPv4
  - `%pI6` et `%pI6c` pour afficher des adresses IPv6
  - Les formats d'IP peuvent être suivit de h, n, l ou b pour demander des conversions d'endian avant d'afficher
  - `%m` pour `errno`
- Référence `linux/printk.h`, `Documentation/printk-formats.txt`
- Il est aussi possible de provoquer des dump de la pile avec les macro `BUG()`, `BUG_ON(condition)` et `BUG_ON_NULL(ptr)`
- Les `pr_debug` ne sont compilé que si l'une des option `CONFIG_DEBUG` ou `CONFIG_DYNAMIC_DEBUG` est active
- Dans le cas de `DYNAMIC_DEBUG`, il est possible d'activer les messages atomiquement en passant par debugfs

debugfs

- Un certain nom d'outils de debug sont commandé par debugfs
- Il s'agit d'un type de partition virtuel
- Activé avec

```
mount -t debugfs none /sys/kernel/debug
```

- Il est possible d'obtenir le statut de dprintk avec

```
cat /sys/kernel/debug/dynamic_debug/control
```

- ... et d'activer de nouvelles traces avec

```
echo 'file hello.c +plf' > /sys/kernel/debug/
    dynamic_debug/control
```

Debuguer le noyau

Activation d `kgdb` permettant d'embarquer `gdbserver` dans le noyau :

- Compilation

```
host$ cp -a build build-dbg
host$ make O=build-dbg ARCH=arm CROSS_COMPILE=
    arm-linux- menuconfig
... Compile the kernel with debug info ...
... KGDB: kernel debugging with remote gdb ...
host$ make -j3 O=build-dbg ARCH=arm
    CROSS_COMPILE=arm-linux- uImage
```

- L'image ELF est beaucoup plus grosse

```
host$ ls -l build-dbg/vmlinux build/vmlinux
host$ cp build-dbg/uImage /srv/tftp/uImage
    -2.6.33.7-dbg
```

Debuguer le noyau

- Passage des arguments `kgdboc` indiquant quel interface `kgdb` doit utilisé et `kgdbwait` indiquant que kgdb doit attendre notre connexion avant de continuer le boot

```
uboot> set bootargs ${bootargs} kgdboc=ttyS0
    kgdbwait
uboot> tftp
<Ctrl-a q>
host$ arm-linux-gdb vmlinux
gdb> target remote /dev/ttyUSB1
```

- Référence : `htmldocs/kgdb.html`

Commandes gdb utiles

Gestion de l'exécution

- Point d'arrêt à l'adresse `0x9876`

```
gdb> b *0x9876
```

- `b hello.c:30` Point d'arrêt à la ligne 30 du fichier `hello.c`

```
gdb> b hello.c:30
```

- Point d'arrêt sur la fonction `main`

```
gdb> b main
```

- Continuer un programme arrêté

```
gdb> c
```

- Démarrer le programme avec `arg` comme argument

```
gdb> r arg
```

Commandes gdb utiles

Obtenir des informations

- Voir la pile d'appel

```
gdb> bt
```

- Afficher les variable locales à la fonction

```
gdb> i locales
```

- Afficher les arguments de la fonction

```
gdb> i args
```

- Afficher la valeur de `a`

```
gdb> p a
```

- Afficher la valeur de `abs(2 * (a - 4))`

```
gdb> p abs(2 * (a - 4))
```

Commandes utiles

Debuguer à chaud

- Attacher `gdb` à un processus existant

```
gdb> attach 1234
```

- Arrêter le debug sans arrêter le processus

```
gdb> detach
```

Commandes utiles

Et plus...

- Afficher la valeur de `i` à chaque fois que l'on passe sur le point d'arrêt 1

```
gdb> command 1
> silent
> p i
> c
> end
```

- Arrêter le programme lorsque la variable `i` est modifiée ou lue

```
gdb> watch i
gdb> awatch i
```

- Passer d'une thread à une autre

```
gdb> thread 2
gdb> thread 1
```

Kgdb et les modules

- Les modules sont chargé dynamiquement. Ils ne sont pas contenu dans le fichier `vmlinux`
- Il est nécessaire d'indiquer à gdb de charger le module que l'on souhaite débuguer
- Néanmoins, gdb ne peut pas savoir à quel addresse à été chargé le module
- Ces informations peuvent être récupérées dans `/sys/modules/*/sections`
- On peut alors les donner à gdb avec `add-symbol-file` :

```
target% cat /sys/modules/my_modules/sections
host$ gdb vmlinux
> target remote 192.168.1.55:2345
> add-symbol-file my_modules.ko 0xd0832000 \
    -s .bss 0xd0837100 -s .data 0xd0836be0
```

- Il existe des scripts effectuant la procédure automatiquement. A adapter suivant vos besoins.

kmemleak

- Active un algorithme proche d'un garbage collector. Mais au lieu de libérer la mémoire, enregistre simplement les blocs perdus.
- Le résultat se trouve dans `/sys/kernel/debug/kmemleak`
- La procédure classique :

```
target% echo clear > /sys/kernel/debug/kmemleak
target% modprobe my_module
...
target% modprobe -r my_module
target% echo scan > /sys/kernel/debug/kmemleak
target% cat /sys/kernel/debug/kmemleak
```

- Référence : `Documentation/kmemleak.txt`

kmemcheck

Modifie la MMU de manière à ce qu'une exception soit déclenchée à chaque accès à la mémoire. Lors qu'un accès en lecture est effectué, kmemcheck vérifie si l'octet à déjà été écrit auparavent. Si ca n'est pas le cas, une erreur du même type qu'un Oops est générée

(Pas disponible sur ARM)

Référence : `Documentation/kmemcheck.txt`

Kprobe

D'un point de vue générique, Les Kprobes permettent de d'effectuer une action lors du passage de l'exécution d'une case mémoire particulière (Référence : `Documentation/kprobes.txt`)

Dans la partie userspace, il existe une api d'avoir des informations statistique sur le passage sur une ligne. Il suffit d'indiquer une adresse (ou un symbole) à prober :

```
echo 'p do_fork' > /sys/kernel/debug/tracing/
    kprobe_events
```

... et de l'activer

```
echo 1 > /sys/kernel/debug/tracing/events/kprobes/
    p_do_fork_0/enable
```

On peut maintenant lire les résultats dans :

```
cat /sys/kernel/debug/tracing/trace
```

Référence : `Documentation/trace/kprobetrace.txt`

Tracepoint

Les tracepoints sont des endroit spécifié dans le code du noyau qui peuvent sembler intéressant à mesurer :

- Pas de possibilité de les déclarer n'importe ou dans le code
- Permet d'obtenir plus d'informations que les kprobes
- Les développeur du noyau ont estimé que l'endroit était pour y mettre un tracepoint et ont configuré le format adéquate (cf. `/sys/kernel/debug/tracing/events/*/format`)
- L'overhead pour un tracepoint désactivé est un poil plus important que pour les kprobe vu qu'il n'y a pas de patch à la volée du code (mais du coup l'usage est permit en XIP)

Pour en obtenir la liste des tracepoints :

```
find /sys/kernel/debug/tracing/events/ -type d
```

Il est évidement possible de de définir ses propres tracepoints (cf. `Documentaiton/trace/tracepoints.txt`)

Référence : `Documentation/trace/events.txt`

ftrace

Permet d'instrumenter certaines latence particulière. Il utilise l'option `-pg` de `gcc`. Cela ajoute un appel à `_mcount` avant chaque appel de fonction. Contrairement à kprobe, ftrace ne peut se placer qu'au début d'une fonction.

Néanmoins, ftrace est très utilisé dans le temps réel, la mesure des temps de latence et l'étude du scheduling.

Reference : `Documentation/trace/ftrace.txt`
`Documentation/trace/ftrace-design.txt`

oprofile

Les CPU modernes possèdent des registres de debugs et breakpoints hardware capable de profiler le code. Oprofile permet de paramétrer et de visualiser le contenu de ces registres. Oprofile n'est disponible que sur certaines architectures.

perf

perf vous offre une interface pour accéder :

- aux tracepoint
- aux kprobe
- aux registres de debug hardware et au breakpoints hardware

Ainsi, perf offre une interface plus complète que oprofile (et disponibles sur plus d'architecture)

(perf ne compile pas sur notre carte de test car notre toolchain est un peu trop vieille pour le noyau 3.3 (et j'ai la flemme de la recompiler))

SystemTap

Permet d'accéder à quasiment toutes les technologies précédentes. Compiler en natif, ce qui permet d'être très rapide... Nécessite une formation à lui-seul.

Le principe :

- On écrit du pseudo-code C
- Il est converti en un modules C
- Il est compilé, puis chargé sur le noyau de la cible
- Le résultats est récupéré par un service appelé *relayfs*

Référence : systemtap-doc

Pour une vue globale est différents outils d'instrumentation : `Documentation/trace/tracepoint-analysis.txt`