Débogage du noyau Linux avec kGDB

Dans le hors-série précédent, en conclusion de son article dédié à l'introduction à la programmation noyau sous Linux, Philippe Biondi passait rapidement en revue les différentes techniques permettant de déboguer le noyau Linux.
L'une de ces techniques, kGDB, mérite que l'on s'y intéresse de plus près, car il s'agit d'une méthode largement utilisée par certains des développeurs noyau (dont par exemple Andrew Morton, le très probable futur mainteneur du prochain noyau stable 2.6).

kGDB ou KDB ou... ?

Avant même de commencer à présenter **kGDB**, parlons un peu des autres solutions permettant de déboguer le noyau, notamment **KDB : oss.sgi.com/projects/kdb/**.
Le grand avantage de kGDB par rapport à KDB est que ce premier est un débogueur source (le débogage – consultation des variables, mode pas à pas, etc. – se fait par rapport au code source C), alors que le deuxième est uniquement un débogueur assembleur (le débogage se fait par rapport au code machine désassemblé). De ce fait, KDB est beaucoup moins pratique à utiliser.
Le désavantage de kGDB vient du fait que le noyau est compilé avec des options spéciales de débogage (utilisation de l'option -g et compilation avec frame pointers), ce qui aura des effets en termes d'empreinte mémoire et de performances, pouvant même aller jusqu'à modifier la dynamique du système et empêcher la reproduction des bogues que l'on recherche. KDB, lui, ne nécessite aucune option spécifique de compilation et ne consomme pas ou peu de ressources quand il est présent.
Un autre désavantage de kGDB est son besoin d'une infrastructure assez lourde (deux machines reliées par un câble série, comme on le verra un peu plus loin), alors que KDB est embarqué directement sur la machine à déboguer (le débogueur est intégré dans le gestionnaire d'interruptions du clavier).
Pour toutes ces raisons, les développeurs noyau s'accordent pour dire que kGDB est une solution recommandée lors du développement, mais que l'on utilisera plutôt KDB pour le débogage d'une machine en production.
Mentionnons cependant qu'il existe d'autres solutions permettant de déboguer le noyau Linux. Il peut être intéressant de les explorer, afin de choisir celle la mieux adaptée à votre besoin. Parmi ces solutions, on trouve **DProbes (www-124.ibm.com/linux/projects/dprobes), Kprobes (www-124.ibm.com/linux/projects/kprobes), UML (user-mode-linux.sourceforge.net), Linux Kernel Crash Dump (lkcd.sourceforge.net), Linux Trace Toolkit (www.opersys.com/LTT/index.html), OProfile (oprofile.sourceforge.net)**, etc.

Préparation des machines et du lien série

Pour utiliser kGDB, il faut disposer de deux machines : la machine de développement et la machine cible (souvent appelée par son nom anglais, la target).
La machine de développement est, comme son nom l'indique, celle sur laquelle on effectue le développement : la modification du code source et sa compilation. Cette dernière peut éventuellement être croisée (en utilisant un ensemble d'outils de cross-compilation) lorsque l'architecture pour laquelle on développe (l'architecture de la cible) n'est pas la même que l'architecture de la machine de développement. (Cependant, dans la suite de cet article, les exemples seront basés sur une architecture x86.)

La machine de développement et la cible doivent être reliées par un câble série de type null-modem. (Il est donc nécessaire de disposer d'au moins un port série sur chacune des deux machines, chose qui devient cependant de plus en plus rare aujourd'hui, les ports série étant en train d'être abandonnés au profit de ports USB ou IEEE1394. Heureusement, des travaux sont en cours pour rendre kGDB utilisable sans devoir passer obligatoirement par un lien série, tel qu'on le verra un peu plus tard).
Les machines ayant été mises en place, la prochaine étape est de tester que le lien série fonctionne correctement (il vaut mieux prendre quelques dizaines de secondes pour le tester maintenant plutôt que de perdre plusieurs dizaines de minutes plus tard, en essayant de faire marcher kGDB sans succès, avant de découvrir que le câble est défectueux...). On peut faire le test simplement, sans utiliser de logiciel particulier. Sur la première machine, faites (en supposant que vous utilisez le premier port série, d'où /dev/ttyS0, sinon remplacez ttyS0 par ttyS1 pour le deuxième port série) :

```
$ stty -F /dev/ttyS0 9600
$ cat /dev/ttyS0
```

Et sur la deuxième :

```
$ stty -F /dev/ttyS0 9600
$ echo "foobar" > /dev/ttyS0
```

Si vous voyez le message apparaître sur la première machine, et que vous pouvez également refaire l'opération dans l'autre sens, alors la liaison est proprement configurée et on peut passer à la suite. Sinon, bidouillez les câbles jusqu'à ce que ça marche :)

Téléchargement du patch kGDB

Le noyau officiel standard ne contient pas par défaut le code permettant son débogage par un lien série. Pour être plus précis, certaines architectures disposent du code kGDB (parmi ces architectures, on peut noter MIPS, PPC, SH), mais pas l'architecture x86, qui fait principalement l'objet de cet article.
La raison de ce manque est le fait que Linus Torvalds n'a jamais aimé les débogueurs, considérant que lorsque l'on utilise un débogueur on aura plutôt tendance à corriger les effets des bogues qu'à essayer de comprendre le code d'abord et corriger les causes des bogues ensuite. De ce fait, il a toujours refusé d'inclure le patch kGDB dans le noyau officiel, en tout cas dans sa version x86 (architecture qui est directement maintenue par lui). D'autres mainteneurs d'architectures ont un avis différent, et c'est pour cela que certaines architectures supportent kGDB sans modifications, alors que pour l'x86 il faut appliquer un patch supplémentaire.
Le patch kGDB pour les architectures x86 a été écrit initialement par Dave Grothe. Plusieurs développeurs ont contribué à son écriture par la suite, et il est actuellement maintenu par Amit S. Kale qui le propose en téléchargement sur le site **kgdb.sourceforge.net**.
Hélas, Amit S. Kale ne propose pas de patch pour tous les noyaux récents, mais uniquement pour certains d'entre eux (le patch pour le noyau RedHat 2.4.20-18.7 est le dernier en date). Bien qu'il soit possible, avec relativement peu de difficultés, d'adapter ce patch pour n'importe quelle version du noyau, il peut être plus facile de télécharger un patch déjà mis à jour sur mon site personnel, situé à **popies.net/kgdb**.
Si vous cherchez une version adaptée aux noyaux 2.5/2.6, il faut savoir que le patch kGDB est inclus dans les noyaux -mm d'Andrew Morton, téléchargeables (on peut ne télécharger que la partie kGDB

du patch) sur : www.kernel.org/pub/linux**/kernel/people/akpm/patches/2.6/**.
Par exemple :

```
$ wget http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.22.tar.bz2
$ wget http://popies.net/kgdb/kgdb-1.6-for-2.4.22.patch.bz2
```

Application du patch kGDB

Une fois les sources du noyau et le patch kGDB téléchargés, on peut appliquer ce dernier facilement (pour des raisons de clarté, une fois les sources du noyau extraites, on renommera le répertoire les contenant pour prendre en compte le fait que l'on compile un noyau modifié par kGDB) :

```
$ bzcat linux-2.4.22.tar.bz2 | tar xf -
$ mv linux-2.4.22 linux-2.4.22-kgdb
$ cd linux-2.4.22-kgdb
$ bzcat ../kgdb-1.6-for-2.4.22.patch.bz2 | patch -p1
patching file arch/i386/config.in
patching file arch/i386/kernel/entry.S
patching file arch/i386/kernel/gdbstart.c
patching file arch/i386/kernel/gdbstub.c
patching file arch/i386/kernel/Makefile
patching file arch/i386/kernel/nmi.c
patching file arch/i386/kernel/signal.c
patching file arch/i386/kernel/traps.c
patching file arch/i386/Makefile
patching file arch/i386/mm/fault.c
patching file Documentation/Configure.help
patching file Documentation/i386/gdb-serial.txt
patching file Documentation/sysrq.txt
patching file drivers/char/gdbserial.c
patching file drivers/char/Makefile
patching file drivers/char/serial.c
patching file drivers/char/sysrq.c
patching file drivers/char/tty_io.c
patching file include/asm-i386/ioctls.h
patching file include/asm-i386/page.h
patching file include/asm-i386/processor.h
patching file include/linux/dcache.h
patching file include/linux/gdb.h
patching file include/linux/sched.h
patching file init/main.c
patching file kernel/ksyms.c
patching file kernel/sched.c
patching file Makefile
```

On peut d'ores et déjà remarquer que le patch kGDB ne modifie qu'un petit nombre de fichiers, et que, parmi ces modifications, un bon nombre sont triviales.
Fonctionnellement, le patch est composé de plusieurs parties :

- des modifications dans le code du noyau à plusieurs endroits clés (handlers des exceptions du processeur, ordonnanceur, traitement des séquences sysrq) afin de pouvoir activer kGDB et effectuer les traitements demandés lors du débogage, notamment l'exécution pas à pas ;
- un driver simplifié pour piloter un port série (dans drivers/char) ;
- l'implémentation du protocole GDB pour le débogage série (dans

arch/i386/kernel/gdbstub.c) ;
- un utilitaire, gdbstart permettant de paramétrer et d'activer le débogage si celui-ci est lancé sur la machine cible (dans arch/i386/kernel/gdbstart) ;
- des modifications dans le système de configuration et de compilation du noyau (ajout des nouveaux fichiers, ajout de l'option -g et enlèvement de l'option -fomit-frame-pointers) permettant de prendre en charge les nouvelles fonctionnalités ;
- de la documentation (dans Documentation/i386/gdb-serial.txt).

Le patch kGDB modifie aussi la version du noyau, en ajoutant -kgdb à la fin de la version (en obtenant donc une version 2.4.22-kgdb dans notre exemple), nous permettant d'installer facilement le noyau modifié par kGDB à côté d'un noyau non modifié.

Configuration du noyau patché

Le noyau étant patché, on peut maintenant le configurer :

```
$ make config
...
*
* Kernel hacking
*
KGDB: Remote (serial) kernel debugging with gdb (CONFIG_X86_REMOTE_DEBUG)
[N/y/?] (NEW) y
KGDB: Thread analysis (CONFIG_KGDB_THREAD) [N/y/?] (NEW) y
KGDB: Console messages through gdb (CONFIG_GDB_CONSOLE) [N/y/?] (NEW) y
...
```

Lors de la configuration, comme on le voit plus haut, trois nouvelles questions sont posées par l'outil :

- **CONFIG_X86_REMOTE_DEBUG** : cette option détermine si le code kGDB va être compilé dans le noyau. En répondant non à cette question, on obtiendrait le même noyau que si l'on n'avait même pas appliqué le patch. Bien évidemment, on va donc répondre oui ;
- **CONFIG_KGDB_THREAD** : cette option permet, lors du débogage, d'utiliser les commandes GDB prévues pour les threads (comme par exemple **info threads**, **thread N**) pour consulter des informations sur les processus tournant sur la cible. L'activation de cette option va cependant induire un petit surcoût dans l'ordonnanceur ;
- **CONFIG_GDB_CONSOLE** : cette option va nous permettre d'utiliser une fonctionnalité intéressante de GDB : en plus des commandes de débogage, GDB peut faire passer des messages supplémentaires par la ligne série et les afficher dans la fenêtre. L'activation de cette option permettra de faire passer les messages de la console du noyau par GDB et donc de les afficher dans la même fenêtre que celle où l'on fait le débogage. C’est particulièrement utile lorsque, comme c'est souvent le cas, la cible est dépourvue d'écran...

Compilation du noyau

Il ne nous reste plus qu'à lancer la compilation du noyau :

```
$ make dep bzImage modules
```

...et à aller boire un petit café (selon la puissance de la machine de chacun et des options choisies lors de la configuration du noyau, il peut s'agir d'un tout petit café ou bien d'un café double assorti d'un croissant :)).
Au bout de quelques minutes, la compilation sera finie et l'on pourra passer à l'étape d'installation.

Installation du noyau

Le choix de la meilleure façon d'installer le noyau sur la machine cible dépend beaucoup du type de la cible : dans le cas d'une carte embarquée, on transférera le noyau par le même lien série que celui qui servira au débogage, en utilisant un chargeur particulier. Lorsque la cible dispose d'une carte réseau, on préférera son utilisation, car on peut ainsi atteindre des vitesses de transfert beaucoup plus élevées.
L'une des façons qui permettent d'installer le noyau sur la cible lorsque l'on dispose d'une connectivité réseau est d'avoir un serveur NFS sur la machine de développement. Le serveur exporte le répertoire contenant les sources (compilées) du noyau, on monte ce répertoire sur la cible, et on effectue l'installation directement à partir de la cible:

```
$ ssh root@cible
cible> mount dev:/home/stelian/linux-2.4.22-kgdb /mnt
cible> cd /mnt
cible> cp arch/i386/boot/bzImage /boot/vmlinuz-2.4.22-kgdb
cible> make modules_install
```

On peut en profiter pour installer aussi gdbstart, l'utilitaire de paramétrage/activation qui a été compilé en même temps que le noyau (cette installation n'est à faire qu'une seule fois, car le binaire gdbstart est indépendant de la version de noyau compilée et on peut utiliser le même pour les prochains noyaux) :

```
cible> cp arch/i386/kernel/gdbstart /sbin/
```

Si l'on ne peut pas se connecter à la cible par un répertoire partagé par NFS, il faut d'abord effectuer l'installation du noyau et des modules sur la machine de développement dans un répertoire temporaire :

```
$ mkdir /tmp/kgdb
$ mkdir /tmp/kgdb/boot
$ cp arch/i386/boot/bzImage /tmp/kgdb/boot/vmlinuz-2.4.22-kgdb
$ make modules_install INSTALL_MOD_PATH=/tmp/kgdb
```

puis transférer, d'une façon ou d'une autre, l'intégralité du contenu du répertoire /tmp/kgdb sur la machine cible.
A partir de ce moment, le noyau kGDB et ses modules ont été correctement installés sur la machine cible. Il ne reste plus qu'à modifier le gestionnaire de démarrage de la machine (lilo ou grub) afin d'ajouter le noyau fraîchement installé dans le menu de démarrage de la machine.
Lors du redémarrage de la cible sur le nouveau noyau, on ne verra rien d'exceptionnel, car kGDB n'est pas actif initialement.

Paramétrage et activation de kGDB

Sur la cible, les paramètres par défaut de kGDB sont :

- le port utilisé est le deuxième port série (**/dev/ttyS1**) ;
- la vitesse de ce port est initialisée à 38400 bps.

Si l'on veut changer ces paramètres, on peut le faire :

- soit au démarrage du noyau, en rajoutant les paramètres suivants sur la ligne de commande du noyau : **gdbttyS=0 gdbbaud=9600**
- soit une fois que le noyau est démarré, en utilisant gdbstart : **cible> gdbstart -t /dev/ttyS0 -s 9600**

(à la différence que cette deuxième méthode activera aussi kGDB tout de suite, ce qui peut ne pas être l'action désirée).
Lorsque les paramètres de kGDB correspondent à notre installation, on peut commencer à l'utiliser. L'activation de kGDB (c'est-à-dire l'interruption du noyau de la cible en donnant le contrôle au GDB tournant sur la machine de développement) peut se faire de plusieurs façons :

- si une erreur quelconque se produit en mode noyau, kGDB est automatiquement activé ;
- l'utilitaire gdbstart permet de faire une activation manuelle ;
- une autre activation manuelle peut être effectuée en faisant la combinaison de touches SysRq + g sur la console (sur un PC, cela se traduit par la séquence de touches AltGr + Print Screen + g) ;
- l'envoi du caractère Control + C sur le port série active aussi le débogueur : **$ echo -e "\003" > /dev/ttyS0**
- enfin, si l'on souhaite démarrer la machine avec kGDB déjà activé (avec l'avantage que l'on peut déboguer du code s'exécutant très tôt lors de la procédure de démarrage), il suffit de rajouter le paramètre gdb sur la ligne de commande du noyau.

kGDB pourra donc être activé sur la machine cible à tout moment, à condition que les interruptions soient actives. Si la machine est bloquée et que les interruptions sont désactivées, il n'y a aucun moyen logiciel permettant de la réveiller. La seule façon possible de le faire est d'attaquer directement le matériel et de générer une interruption NMI. Comme son nom l'indique, une Non Maskable Interrupt sera prise en compte par le processeur même si celui-ci se trouve dans une section de code où les interruptions ont été désactivées, et cela réveillera kGDB. La génération d'une NMI peut être réalisée avec plus ou moins de matériel électronique ; on peut par exemple le faire avec un simple trombone sur les cartes mères ayant au moins un slot ISA en court-circuitant les pins A1 et B1 de ce slot !

Lancement de GDB

Lorsque kGDB a été activé, la machine cible se retrouve complètement bloquée, en attendant d'être pilotée par un GDB distant au moyen du lien série.
Sur la machine de développement, on va donc lancer gdb en lui passant en paramètre le binaire vmlinux se trouvant à la racine des sources du noyau. vmlinux est le même noyau que celui se trouvant dans le fichier **arch/i386/boot/bzImage** et que l'on a installé sur la cible, le premier étant le binaire ELF alors que le deuxième contient une version compressée de ce binaire ainsi que du code permettant le chargement initial en mémoire de ce noyau.
Une fois gdb lancé, on va utiliser les commandes set remotebaud et target remote pour dire à gdb

que le binaire ne s'exécute pas localement mais à distance :

```
$ gdb vmlinux
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) set remotebaud 9600
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
breakpoint () at gdbstub.c:1344
1344    }
warning: shared library handler failed to enable breakpoint
warning: shared library handler failed to enable breakpoint
```

Bingo ! **gdb** a bien réussi à se connecter à la cible, qui est interrompue à la ligne 1344 de **gdbstub.c**, en attendant cette connexion.
Le message de warning de gdb concernant les bibliothèques partagées n'est pas important (après tout, le noyau que l'on débogue n'utilise pas de bibliothèques partagées !) et peut être ignoré.
Afin de simplifier les futurs lancements, on pourra créer, à la racine des sources du noyau, le fichier .gdbinit qui sera lu et évalué par gdb à chaque lancement, et qui contiendra les deux lignes de paramétrage du lien série. On peut en profiter pour rajouter dans ce même fichier d'autres macro-commandes utiles, comme la macro ps permettant de connaître la liste des processus, lsmod qui affiche la liste des modules chargés, ainsi que des macros qui seront utilisées pour poser des points d'arrêt matériels (et que l'on utilisera un peu plus loin dans cet article). Toutes ces macros sont téléchargeables sur le site de **kGDB : kgdb.sourceforge.net/gdbmacros.html.**
Il peut être utile de mentionner que l'utilisation d'un front-end graphique à gdb, comme par exemple ddd, est tout à fait possible et compatible avec toutes les opérations que l'on va faire.
Maintenant que la cible est bien sous le contrôle de gdb, on peut la débloquer en utilisant la commande cont :

```
(gdb) cont
Continuing.
```

Lorsque l'on veut reprendre la main dans le débogueur, il suffit de taper Control + C :

```
^C
Program received signal SIGTRAP, Trace/breakpoint trap.
breakpoint () at gdbstub.c:1344
1344    }
(gdb)
```

Si on veut arrêter le débogage et donc quitter gdb tout en laissant le noyau de la cible en train de tourner, on se retrouve devant un petit dilemme : si on utilise Control + C pour reprendre la main dans **gdb** afin de quitter correctement, le noyau de la cible sera laissé bloqué. La solution est de quitter gdb en tapant Control + Z, puis, une fois revenu au shell, tuer le processus gdb :

```
(gdb) cont
Continuing.
^Z
```

```
[1]+  Stopped                 gdb vmlinux
$ kill %1
[1]+  Stopped                 gdb vmlinux
[1]+  Terminated              gdb vmlinux
```

On peut bien entendu par la suite relancer gdb à tout moment et reprendre la main sur le noyau de la cible en utilisant la même procédure que pour l'activation initiale.
De temps en temps, il arrive aussi que gdb devienne un peu confus quant au fil de l'exécution, et dans ce cas il est préférable de le quitter et de le relancer...

Utilisation de kGDB

L'utilisation de kGDB sur le noyau ressemble beaucoup à l'utilisation de GDB sur un programme standard. On peut consulter la valeur des variables, poser des points d'arrêt, tracer l'exécution en mode pas à pas, connaître la pile d'appel à un instant donné, etc.
Par rapport à ce type d'utilisation standard, kGDB apporte quelques fonctionnalités avancées, comme l'utilisation des points d'arrêts hardware qui permettent de surveiller une adresse mémoire par le processeur, afin de détecter des accès en lecture ou écriture à cette adresse, ou bien la possibilité de consulter la pile d'appel de chaque processus ordonnancé par le noyau.
Dans la suite de cet article, nous allons présenter rapidement quelques-unes de ces fonctionnalités au travers d'exemples réels d'exécution. Pour une documentation plus complète sur les commandes gdb, veuillez vous rapporter au manuel **info**, très complet, de **gdb**.

Consultation du source et des données

La consultation du code source peut se faire à tout moment grâce à la commande list de gdb. Sans argument, cette commande affiche le code source correspondant à l'endroit où le pointeur d'exécution se trouve, mais on peut sélectionner un autre fichier source, une autre fonction ou un autre numéro de ligne en les passant en paramètre à la commande **list** :

```
^C
Program received signal SIGTRAP, Trace/breakpoint trap.
breakpoint () at gdbstub.c:1344
1344    }
(gdb) list
1339
1340    void breakpoint(void)
1341    {
1342            if (initialized)
1343                    BREAKPOINT();
1344    }
1345
1346    #ifdef CONFIG_GDB_CONSOLE
1347    char    gdbconbuf[BUFMAX];
1348
(gdb) list sched.c:0
1       /*
2        *  linux/kernel/sched.c
3        *
4        *  Kernel scheduler and related syscalls
5        *
6        *  Copyright (C) 1991, 1992  Linus Torvalds
```

```
7        *
8        *  1996-12-23  Modified by Dave Grothe to fix bugs in semaphores and
9        *              make semaphores SMP safe
10       *  1998-11-19  Implemented schedule_timeout() and related stuff
(gdb) list sys_sethostname
1041            up_read(&uts_sem);
1042            return errno;
1043    }
1044
1045    asmlinkage long sys_sethostname(char *name, int len)
1046    {
1047            int errno;
1048            char tmp[__NEW_UTS_LEN];
1049
1050            if (!capable(CAP_SYS_ADMIN))
(gdb)
```

La consultation des variables peut se faire en utilisant la commande print :

```
(gdb) print system_utsname
$1 = {sysname = "Linux", '\0' <repeats 59 times>,
  nodename = "cible", '\0' <repeats 59 times>,
  release = "2.4.22-kgdb", '\0' <repeats 53 times>,
  version = "#2 Wed Oct 15 13:22:54 CEST 2003", '\0' <repeats 32 times>,
  machine = "i686", '\0' <repeats 60 times>,
  domainname = "(none)", '\0' <repeats 58 times>}
(gdb) print system_utsname.nodename
$2 = "cible", '\0' <repeats 59 times>
```

La commande **x** (abréviation de examine) permet de consulter une zone mémoire, en précisant la taille de cette zone, le format des données à lire et le format à utiliser à l'affichage. Par exemple, pour afficher en format hexadécimal les 8 premiers octets se trouvant à l'adresse de la variable **system_utsname.nodename** on fera :

```
(gdb) x/8xb system_utsname.nodename
0xc0269a21 <system_utsname+65>: 0x63    0x69    0x62    0x6c    0x65    0x00
0x00    0x00
```

La commande **print** peut aussi servir à modifier dynamiquement une variable. Dans l'exemple suivant, nous allons modifier le nom d'hôte de la cible:

```
(gdb) print system_utsname.nodename="target"
$3 = "target", '\0' <repeats 58 times>
```

Par la suite, toute application effectuant l'appel système gethostname aura en retour le nouveau nom d'hôte.

Points d'arrêt

Les points d'arrêt peuvent être posés à n'importe quel endroit du code en utilisant la commande break, à l'exception de :

- la phase initiale de démarrage du noyau (correspondant à l'initialisation de la plate-forme, la mise en place des mappings mémoire, l'initialisation de tous les sous-systèmes et pilotes du noyau, etc.). Si l'on veut cependant activer kGDB le plus tôt possible lors du démarrage du

noyau, on peut utiliser l'option gdb sur la ligne de commande du noyau comme précisé plus haut.
- les parties du noyau utilisées par kGDB pour sa gestion interne (le driver série simplifié, l'implémentation du protocole de kGDB, le code permettant d'intercepter les exceptions et interruptions à destination de kGDB).

Par exemple, on peut mettre un point d'arrêt sur l'implémentation de l'appel système **stat64**, puis l'effacer une fois qu'il sera atteint par l'un des processus ordonnancés sur la cible:

```
(gdb) break sys_stat64
Breakpoint 1 at 0xc0143c6c: file stat.c, line 339.
gdb) cont
Continuing.
```

```
Breakpoint 1, sys_stat64 (filename=0x804dc61 "/dev/initctl",
    statbuf=0x804dc61, flags=2) at stat.c:339
339             error = user_path_walk(filename, &nd);
(gdb) delete 1
```

Mode pas à pas

Le mode pas à pas est effectué grâce aux commandes step et next de gdb. La différence entre les deux est que, si la prochaine ligne du code source est un appel de fonction, la commande step exécutera en mode pas à pas la fonction appelée, alors que la commande next exécutera l'appel en un seul pas et s'arrêtera sur la ligne de code suivant l'appel de la fonction.
L'exécution en mode pas à pas peut être relativement difficile à suivre parfois, car le noyau est compilé en mode optimisé (et donc le compilateur a quelques fois réarrangé l'ordre les lignes du code source lors de la génération du code), et l'on utilise très souvent des fonctions inline ou des macro-commandes...
En reprenant l'exemple où l'on avait posé un point d'arrêt dans l'appel système stat64, on peut tracer cette fonction en mode pas à pas (et remarquer le déroulement de la trace, influencé à la fois par l'optimisation du code et par l'exécution de la fonction inline kdev_t_to_nr) :

```
339             error = user_path_walk(filename, &nd);
(gdb) next
340             if (!error) {
...
(gdb) next
343                             error = cp_new_stat64(nd.dentry->d_inode,
statbuf);
(gdb) print *nd.dentry->d_inode
$11 = {i_hash = {next = 0xcbf25a20, prev = 0xcbf25a20}, i_list = {
...
   i_ino = 411, i_count = {counter = 1}, i_dev = 6,
   i_mode = 4480, i_nlink = 1, i_uid = 0, i_gid = 0, i_rdev = 0, i_size = 0,
   i_atime = 1066323791, i_mtime = 1066323791, i_ctime = 1066323791,
   i_blkbits = 10, i_blksize = 1024, i_blocks = 0, i_version = 0, i_bytes = 0,
...
(gdb) step
cp_new_stat64 (inode=0x0, statbuf=0xcbf25a20) at stat.c:280
280             memset(&tmp, 0, sizeof(tmp));
(gdb) next
276     {
```

```
(gdb) next
280             memset(&tmp, 0, sizeof(tmp));
(gdb) next
93              return (MAJOR(dev)<<8) | MINOR(dev);
(gdb) next
92      static inline unsigned int kdev_t_to_nr(kdev_t dev) {
(gdb) next
282             tmp.st_ino = inode->i_ino;
(gdb) next
284             tmp.__st_ino = inode->i_ino;
(gdb) next
286             tmp.st_mode = inode->i_mode;
(gdb) next
287             tmp.st_nlink = inode->i_nlink;
(gdb) next
288             tmp.st_uid = inode->i_uid;
(gdb) next
289             tmp.st_gid = inode->i_gid;
(gdb) next
93              return (MAJOR(dev)<<8) | MINOR(dev);
(gdb) next
92      static inline unsigned int kdev_t_to_nr(kdev_t dev) {
(gdb) next
291             tmp.st_atime = inode->i_atime;
(gdb) next
292             tmp.st_mtime = inode->i_mtime;
(gdb) next
293             tmp.st_ctime = inode->i_ctime;
(gdb) next
294             tmp.st_size = inode->i_size;
(gdb) next
313             if (!inode->i_blksize) {
(gdb) next
328                     tmp.st_blocks = inode->i_blocks;
(gdb) next
329                     tmp.st_blksize = inode->i_blksize;
(gdb) next
328                     tmp.st_blocks = inode->i_blocks;
(gdb) next
331             return copy_to_user(statbuf,&tmp,sizeof(tmp)) ? -EFAULT : 0;
(gdb) print tmp
$12 = {st_dev = 6, __pad0 = "\000\000\000\000\000\000\000\000\000\000",
  __st_ino = 411, st_mode = 4480, st_nlink = 1, st_uid = 0, st_gid = 0,
  st_rdev = 0, __pad3 = "\000\000\000\000\000\000\000\000\000\000", st_size = 0,
  st_blksize = 1024, st_blocks = 0, __pad4 = 0, st_atime = 1066323791,
  __pad5 = 0, st_mtime = 1066323791, __pad6 = 0, st_ctime = 1066323791,
  __pad7 = 0, st_ino = 411}
(gdb)
```

Accès à la pile d'appel des processus

La pile d'appel du noyau peut être consultée à n'importe quel moment par la commande backtrace ou son équivalent **where** :

```
(gdb) backtrace
#0  cp_new_stat64 (inode=0xcb6eba20, statbuf=0x400) at stat.c:331
#1  0xc013fe2f in sys_stat64 (filename=0x400 <Address 0x400 out of bounds>,
```

```
    statbuf=0x400, flags=-1073743088) at stat.c:343
```

De plus, si lors de la configuration du noyau on a choisi d'activer l'option **CONFIG_KGDB_THREAD**, alors on peut utiliser les commandes gdb prévues pour gérer les threads, et on pourra connaître la pile d'appel de chaque processus se trouvant en mode noyau :

```
(gdb) info threads
  26 Thread 1108  schedule_timeout (timeout=2147483647) at sched.c:424
  25 Thread 1107  schedule_timeout (timeout=2147483647) at sched.c:424
  24 Thread 1106  schedule_timeout (timeout=2147483647) at sched.c:424
...
  13 Thread 934  do_syslog (type=6184, buf=0x804d6a0 "Initdefault level '%c' is
invalid", len=4095) at printk.c:190
...
  5 Thread 4  kswapd (unused=0x0) at current.h:7
  4 Thread 3  ksoftirqd (__bind_cpu=0x0) at current.h:7
  3 Thread 2  context_thread (startup=0xc02a7f24) at context.c:101
  2 Thread 1  cp_new_stat64 (inode=0xcb6eba20, statbuf=0x400) at stat.c:331
* 1 Thread 0  cpu_idle () at process.c:138
(gdb) thread 2
[Switching to thread 2 (Thread 1)]#0  cp_new_stat64 (inode=0xcb6eba20,
    statbuf=0x400) at stat.c:331
331             return copy_to_user(statbuf,&tmp,sizeof(tmp)) ? -EFAULT : 0;
(gdb) bt
#0  cp_new_stat64 (inode=0xcb6eba20, statbuf=0x400) at stat.c:331
#1  0xc013fe2f in sys_stat64 (filename=0x400 &mt;Address 0x400 out of bounds>,
    statbuf=0x400, flags=-1073743088) at stat.c:343
(gdb) thread 5
[Switching to thread 5 (Thread 4)]#0  kswapd (unused=0x0) at current.h:7
7       {
(gdb) bt
#0  kswapd (unused=0x0) at current.h:7
#1  0xc010728a in arch_kernel_thread (fn=0xc0259cc0 <contig_page_data>,
    arg=0x1, flags=9) at process.c:492
(gdb) thread 13
[Switching to thread 13 (Thread 934)]#0  do_syslog (type=6184,
    buf=0x804d6a0 "Initdefault level '%c' is invalid", len=4095)
    at printk.c:190
190                     error = wait_event_interruptible(log_wait, (log_start -
log_end));
(gdb) bt
#0  do_syslog (type=6184, buf=0x804d6a0 "Initdefault level '%c' is invalid",
    len=4095) at printk.c:190
#1  0xc0158d7d in kmsg_read (file=0xcb4c42c0,
    buf=0x1828 <Address 0x1828 out of bounds>, count=6184, ppos=0xcb4c42e0)
    at kmsg.c:35
#2  0xc0138d17 in sys_read (fd=6184,
    buf=0x1828 <Address 0x1828 out of bounds>, count=4095) at read_write.c:177
```

Points d'arrêt matériels

Le patch kGDB permet d'utiliser les facilités matérielles de débogage offertes par les processeurs x86. En effet, ces processeurs ont des registres spéciaux de débogage et on peut les utiliser pour installer des points de surveillance sur des adresses mémoire. Lorsqu'un accès en lecture, écriture ou exécution est détecté à cette adresse, le processeur lèvera une exception permettant à kGDB de

reprendre le contrôle sur le noyau.
Les points de surveillance sont au nombre de 4 maximum, numérotés de 0 à 3, et ils peuvent s'appliquer à une zone mémoire de 1, 2 ou 4 octets. La mise en place et la suppression des points de surveillance ne se fait pas en utilisant les commandes standard de gdb (les commandes watch), mais se fait grâce à des macro-commandes gdb (ces macros peuvent être téléchargées sur le site de kGDB et mises dans le fichier .gdbinit pour les charger en mémoire automatiquement). On dispose alors de :

- **hwebrk** : installe un point de surveillance d'exécution ;
- **hwwbrk** : installe un point de surveillance d'écriture ;
- **hwabrk** : installe un point de surveillance d'accès ;
- **hwrmbrk** : supprime un point de surveillance ;
- **exinfo** : affiche des informations sur la dernière interruption du noyau précisant si elle a été causée par un point d'arrêt ou par un point de surveillance.

Dans l'exemple suivant, on va installer un point de surveillance sur la variable contenant le nom d'hôte de la machine, et on va déclencher l'exception en modifiant ce nom par le lancement, sur la cible, de la commande **hostname** :

```
(gdb) p system_utsname.nodename
$1 = "cible", '\0' <repeats 59 times>
(gdb) p &system_utsname.nodename
$2 = (char (*)[65]) 0xc0269a21
(gdb) hwabrk 0 4 c0269a21
sending: "Y0,1,4,c0269a21"
received: "OK"
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0xc01275f9 in sys_sethostname (name=0x0, len=3) at string.h:202
202     __asm__ __volatile__(
(gdb) exinfo
sending: "qE"
received: "Hardware breakpoint 0"
(gdb) bt
#0  0xc0125570 in sys_sethostname (name=0x0, len=3) at string.h:202
(gdb) hwrmbrk 0
sending: "y0"
received: "OK"
```

Débogage des modules

Le débogage des modules noyau est assez difficile, car, lors du lancement de gdb, on lui fournit uniquement le code de **vmlinux**. Quand on chargera des modules plus tard dans le noyau de la cible, gdb ne pourra pas le détecter et il ne pourra pas résoudre les adresses appartenant au module fraîchement chargé.
Il est donc conseillé d'utiliser les modules le moins possible lorsque l'on fait du débogage en utilisant kGDB, ou au moins de ne pas en charger dynamiquement lors du démarrage de la machine, mais de le faire manuellement par la suite en utilisant une procédure spéciale.
Le but de cette procédure est de récupérer, pour chaque module, lors de son insertion dans le noyau, les adresses où le module a été chargé. Puis, en utilisant ces adresses et la commande **add-symbol-**

file de **gdb**, on fait connaître à ce dernier l'existence du module dans l'espace d'adressage du noyau. Attention, pour insérer le module sur la cible il faut que le noyau tourne (il faut donc avoir sélectionné cont dans **gdb**). Mais après, pour insérer les symboles dans **gdb** il faut rendre la main au débogueur (en interrompant l'exécution du noyau par Control + C).
Un script appelé loadmodule.sh est fourni sur le site de kGDB et permet d'automatiser au maximum ces opérations, en effectuant :

- une copie du module sur la cible en utilisant **rsh** ;
- l'insertion du module dans le noyau de la cible ;
- la récupération des informations sur les adresses d'insertion du module ;
- la sauvegarde de ces informations, en utilisant une syntaxe directement compréhensible par gdb dans un fichier temporaire.

Une fois ce script fini, il suffira de charger le fichier temporaire dans **gdb** en utilisant sa commande source afin de faire prendre en compte le module par la suite.
Le module étant chargé, toutes les opérations sur son code sont permises, comme la consultation de son code, la mise de points d'arrêt, etc. Voyons un exemple avec le module **loop** (on utilisera un point d'arrêt que l'on atteindra en exécutant **mount -o loop image.iso /mnt** sur la cible) :

```
$ loadmodule.sh cible drivers/block/loop.o
Copying drivers/block/loop.o to cible
Loading module drivers/block/loop.o
Generating script /tmp//loadcibleloop.o
$ cat /tmp//loadcibleloop.o
add-symbol-file drivers/block/loop.o 0xcc878060 -s .fixup 0xcc879fef -s
.rodata.str1.32 0xcc87a0c0 -s .rodata.str1.1 0xcc87a2ca -s __ex_table
0xcc87a328 -s __ksymtab 0xcc87a474 -s .rodata 0xcc87a4a4 -s __archdata
0xcc87a4c0 -s __kallsyms 0xcc87a4c0 -s .data 0xcc87ac80 -s .bss 0xcc87ad28
$ gdb vmlinux
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
...
(gdb)
breakpoint () at gdbstub.c:1344
1344    }
warning: shared library handler failed to enable breakpoint
(gdb) source /tmp/loadcibleloop
add symbol table from file "drivers/block/loop.o" at
        .text_addr = 0xcc86c060
        .fixup_addr = 0xcc86deb5
        .rodata.str1.32_addr = 0xcc86df80
        .rodata.str1.1_addr = 0xcc86e183
        __ex_table_addr = 0xcc86e1e0
        __ksymtab_addr = 0xcc86e32c
        .rodata_addr = 0xcc86e35c
        __archdata_addr = 0xcc86e370
        __kallsyms_addr = 0xcc86e370
        .data_addr = 0xcc86eb40
        .bss_addr = 0xcc86ebe8
warning: section __archdata not found in /home/stelian/linux-2.4.22-
kgdb/drivers/block/loop.o
warning: section __kallsyms not found in /home/stelian/linux-2.4.22-
kgdb/drivers/block/loop.o
(gdb) break lo_open
Breakpoint 2 at 0xcc86d6bf: file loop.c, line 911.
(gdb) c
Continuing.

Breakpoint 2, lo_open (inode=0xc9998280, file=0xc9cb8220) at loop.c:911
911             if (!inode)
```

```
(gdb) n
913             if (MAJOR(inode->i_rdev) != MAJOR_NR) {
(gdb) n
917             dev = MINOR(inode->i_rdev);
(gdb) n
918             if (dev >= max_loop)
(gdb) n
921             lo = &loop_dev[dev];
(gdb) where
#0  lo_open (inode=0xc9998280, file=0xc9cb8220) at loop.c:921
#1  0xc013f194 in do_open (bdev=0xc1394a60, inode=0x0, file=0x0)
    at block_dev.c:571
#2  0xc013f2e5 in blkdev_open (inode=0xc9998280, filp=0xc9cb8220)
    at block_dev.c:623
#3  0xc017aab9 in devfs_open (inode=0xc9998280, file=0xc9cb8220) at base.c:2790
#4  0xc013803d in dentry_open (dentry=0xc9b352a0, mnt=0xc121f320,
    flags=-912686464) at open.c:698
#5  0xc0137f3b in filp_open (filename=0x0, flags=32768, mode=-912686464)
    at open.c:656
#6  0xc0138270 in sys_open (filename=0x0, flags=0, mode=0) at open.c:798
```

Si l'on veut décharger le module et le recharger par la suite, il faut quitter et relancer gdb, puis refaire la procédure de chargement du module, car il n'y a pas de moyen de libérer une partie de symboles dans la mémoire de **gdb**.

Désassemblage du code source

Lors du débogage, il faut parfois descendre au niveau du code assembleur généré lors de la compilation afin de détecter certains problèmes d'optimisation. Pour ce faire, **gdb** dispose des commandes info line et **disassamble**. La première permet de consulter le mapping entre le code source et les adresses en mémoire. La seconde effectue de désassemblage d'une fonction entière ou d'un intervalle d'adresses :

```
(gdb) info line cp_new_stat64
Line 280 of "stat.c" starts at pc 0xc014550 and ends at 0xc0145619
```

```
(gdb) disassamble cp_new_stat64
Dump of assembler code for function cp_new_stat64:
0xc014f550 <cp_new_stat64+0>:   push   %esi
0xc014f551 <cp_new_stat64+1>:   push   %ebx
0xc014f552 <cp_new_stat64+2>:   sub    $0x6c,%esp
0xc014f555 <cp_new_stat64+5>:   mov    0x78(%esp,1),%ebx
0xc014f559 <cp_new_stat64+9>:   movl   $0x60,0x8(%esp,1)
0xc014f561 <cp_new_stat64+17>:  lea    0xc(%esp,1),%esi
0xc014f565 <cp_new_stat64+21>:  mov    %esi,(%esp,1)
0xc014f568 <cp_new_stat64+24>:  movl   $0x0,0x4(%esp,1)
0xc014f570 <cp_new_stat64+32>:  call   0xc014f6e0
<__constant_c_and_count_memset>
...
```

Futur de kGDB

Il y a eu des discussions à plusieurs reprises sur la liste de diffusion du noyau (www.tux.org/lkml) sur la possibilité de faire évoluer kGDB afin de s'affranchir de l'obligation de posséder un port série sur la machine.
Ces discussions ont récemment donné lieu à de réels travaux sur ce sujet, et on trouve aujourd'hui une implémentation de kGDB fonctionnant avec certaines cartes réseau (en attaquant la carte à un niveau très bas, en mode polling, sans utilisation des interruptions ni des couches réseau). Ces développements récents sont intégrés dans la version de kGDB distribuée avec les noyaux 2.6-mm d'Andrew Morton et se basent sur la même couche d'abstraction que NetConsole (qui permet de connecter une console sur la machine en utilisant une interface réseau) et NetDump (qui permet, en cas de crash, de sauvegarder l'image mémoire du noyau sur une autre machine du réseau pour être analysée ultérieurement).
Toujours dans la version maintenue par Andrew Morton, on retrouve un certain nombre d'améliorations du patch kGDB apportées par George Anzinger (la possibilité de déboguer du code se trouvant encore plus tôt dans la séquence de démarrage du noyau, à la première ligne de code C !, l'amélioration de la sortie de info threads, la possibilité de faire des traces événementielles dans le noyau et de les récupérer par kGDB, etc.). Ces améliorations n'ont cependant pas (encore) été réintégrées dans le patch kGDB de Amit S. Kale.
Soulignons aussi le fait que, selon les rumeurs qui courent parmi les développeurs noyau, une fois le noyau stable 2.6 sorti, Linus Torvalds se consacrera au développement de la nouvelle branche de développement – la 2.7 –, et le maintien du noyau stable incombera à ... Andrew Morton. Si ces rumeurs s'avèrent fondées, il est fort possible que l'intégration de kGDB dans le version officielle du noyau revienne à l'ordre du jour...

Conclusion

Nous voici à la fin de cette présentation de la mise en place de kGDB et des fondements de son utilisation pour le débogage du noyau Linux.
J'espère que cet article a rempli son rôle de démystification de kGDB et qu'il vous aura donné l'envie de l'utiliser lors de vos prochains développements de code noyau.