Débogage kernel et applicatif avec Ftrace pour système généraliste ou embarqué[1]

Christophe BLAESS

Les outils et les méthodes de débogage pour l'espace applicatif Linux sont plutôt bien connus : de Valgrind à Strace/Ltrace en passant par GDB et toutes ses déclinaisons (DDD, Eclipse, Gdbserver), la documentation et les exemples sont très répandus. En ce qui concerne l'espace noyau, la pratique du débogage est beaucoup plus limitée. Pourtant, certains outils comme Ftrace permettent non seulement d'aider à la mise au point du code kernel, mais également d'analyser finement les comportements des tâches de l'espace utilisateur.

Ftrace

Le système Ftrace a été développé en 2008 par Steven Rostedt (Redhat) et intégré dans le noyau Linux *mainline* 2.6.26. Il fournit un moyen de suivre depuis l'espace utilisateur le comportement du noyau en enregistrant des points de passage dans le code et en les présentant dans le système de fichiers **`debugfs`**.

L'utilisation de Ftrace est plutôt simple, mais nécessite d'écrire des paramètres dans plusieurs fichiers successifs, ce qui s'avère fastidieux à la longue. Un outil nommé **`trace-cmd`** propose une commande unique qui prend en charge la gestion des fichiers de **`debugfs`** concernés. Les résultats de Ftrace se présentent sous forme de fichiers de texte brut, ce qui est parfaitement suffisant dans certains cas (mesure de durée de latence maximale, liste des fonctions invoquées, etc.). Toutefois dans certaines circonstances, une présentation graphique des résultats serait préférable (synchronisation de tâches, mesure de temps de commutation, etc.). Une interface graphique simple, nommée **`kernelshark`** permet de visualiser les résultats de **`trace-cmd`**.

Dans le cas des systèmes embarqués, les restrictions concernant le système de fichiers (accès en lecture seulement, taille réduite…) ainsi que l'absence d'environnement graphique limitent l'usage de **`trace-cmd`** et de **`kernelshark`**. Heureusement il est possible de les utiliser sur un hôte de développement distant (habituellement un PC) pour récupérer et afficher les informations statistiques.

Cet article va présenter rapidement le système Ftrace sans détailler toutes ses possibilités. Pour le lecteur désireux d'en savoir plus, je conseille [Ficheux 2011], [Rostedt 2009] ainsi que le fichier **`Documentation/trace/ftrace.txt`** se trouvant dans les sources du noyau Linux (à consulter par exemple sur [LXR]). Nous verrons plusieurs utilisations typiques de Ftrace, pour l'espace applicatif comme pour le noyau, ainsi qu'un exemple de mise en œuvre pour un système embarqué.

Configuration du noyau

La première opération consiste à s'assurer du support de Ftrace dans le noyau du système. Pour pouvoir explorer la problématique des systèmes embarqués, je vais prendre en exemple le Raspberry Pi 2, largement répandu et facile d'accès. Pour obtenir rapidement une base de travail assez complète, je pars de la dernière distribution Raspbian disponible lors de la rédaction de ces lignes, la version du 9 février 2016.

Après avoir inscrit l'image téléchargée sur une carte SD de capacité suffisante (par exemple 16 Go), je démarre le Raspberry Pi 2 et j'examine la configuration. L'image décompressée de la distribution Raspbian occupe 4 Go de mémoire flash. Toutefois, pour mener à bien l'ensemble de nos expériences (avec compilation de code noyau) nous verrons plus loin qu'il nous faudra plusieurs giga-octets disponibles, aussi il ne faut pas lésiner sur la capacité de la micro-SD employée, et penser à utiliser la commmande **`raspi-config`** pour agrandir le système de fichiers pour utiliser tout l'espace disponible.

```
$ grep  debugfs  /proc/filesystems
nodev   debugfs
$
```

Le noyau a bien été compilé avec le support pour le système de fichiers **`debugfs`**, voyons maintenant s'il est monté dans notre arborescence :

```
$ mount  |  grep  debugfs
```

1 Cet article est paru dans le numéro 18 d'Open Silicium (Avril / Mai / Juin 2016).

```
debugfs on /sys/kernel/debug type debugfs (rw,relatime)
$
```

Il est accessible à l'endroit habituel **`/sys/kernel/debug`**. Si ça n'avait pas été le cas, j'aurais pu le monter manuellement avec :

> **`sudo mount none /sys/kernel/debug/ -t debugfs.`**

Notons que la plupart des manipulations à venir ne pourrons se faire qu'avec les droits *root*, aussi je vais basculer sur ce compte de manière prolongée, chose qu'on ne fait évidemment pas sur un système sensible.

```
$ sudo -i
# cd /sys/kernel/debug/
# ls
bcm2708_fb  extfrag          hid       pinctrl       sleep_time  vc-smem
bdi         f2fs             ieee80211 pm_qos        tracing
cleancache  fault_around_bytes kprobes  pwm           usb
clk         frontswap        memblock  regmap        vchiq
dma_buf     gpio             mmc0      sched_features vc-mem
#
```

Le répertoire qui permet l'accès aux fonctionnalités du sous-système Ftrace s'appelle **`tracing`**.

```
# cd tracing/
# ls
available_events          options             stack_max_size
available_filter_functions per_cpu            stack_trace
available_tracers         printk_formats      stack_trace_filter
buffer_size_kb            README              trace
buffer_total_size_kb      saved_cmdlines      trace_clock
current_tracer            saved_cmdlines_size trace_marker
dyn_ftrace_total_info     set_event           trace_options
enabled_functions         set_ftrace_filter   trace_pipe
events                    set_ftrace_notrace  trace_stat
free_buffer               set_ftrace_pid      tracing_cpumask
function_profile_enabled  set_graph_function  tracing_max_latency
instances                 set_graph_notrace   tracing_on
max_graph_depth           snapshot            tracing_thresh
```

Commençons par examiner les traceurs disponibles dans cette version du noyau :

```
# cat available_tracers
blk function_graph wakeup_dl wakeup_rt wakeup irqsoff function nop
```

Le noyau est donc compilé avec un bon support pour Ftrace, nous pourrons obtenir facilement des informations intéressantes. Les traceurs présents sont :

- **`blk`** : permet d'obtenir un suivi de l'activité de la file de requêtes sur un périphérique bloc. Ceci peut être utile pour affiner les paramètre de fonctionnement d'un disque par exemple, mais également pour étudier et comprendre les interactions entre les tâches, les systèmes de fichiers, le cache et les périphériques blocs.

- **`function`** : un suivi de l'exécution des fonctions internes du noyau, utile pour analyser et mettre au point des drivers par exemple, mais aussi pour étudier les commutations entre les tâches du système.

- **`function_graph`** : le traceur **`function`** enregistre le passage dans les fonctions du noyau, tandis que celui-ci mémorise les points de passage en entrée et en sortie de chaque fonction. Il est ainsi possible de retracer le graphe des appels imbriqués dans le noyau.

- **`wakeup`** : ce traceur indique la plus longue durée observée entre le réveil d'une tâche de priorité supérieure

à toutes les autres et son activation effective.

- **wakeup_rt** : le traceur **wakeup** indiquait la plus longue latence de réveil d'une tâche prioritaire parmi toutes les tâches du système, celui-ci est plus orienté vers le temps réel et s'intéresse aux tâches ordonnancées en **SCHED_FIFO** ou **SCHED_RR**.

- **wakeup_dl** : comme le précédent, ce traceur mesure la latence la plus longue au réveil d'une tâche de priorité élevée, en s'intéressant à celles sous ordonnancement **SCHED_DEADLINE**. Il s'agit d'un nouvel ordonnancement inspiré du mécanisme *Earliest Deadline First*. Apparu dans le noyau 3.14, il reste encore peu utilisé, mais offre une prédictibilité comparable à **SCHED_FIFO** ou **SCHED_RR**.

- **irqsoff** : indique la durée la plus longue pendant laquelle les interruptions sont masquées, ce qui permet d'avoir une approximation de la plus longue latence rencontrée pour la réponse à un événement externe.

- **nop** : n'est pas vraiment un outil de suivi, c'est l'indicateur d'arrêt du système Ftrace.

On peut arrêter le suivi à tout moment en sélectionnant ce dernier traceur, c'est-à-dire en inscrivant son nom dans le fichier **current_tracer**.

```
# echo nop > /sys/kernel/debug/tracing/current_tracer
# cat  /sys/kernel/debug/tracing/current_tracer
nop
#
```

Il y a deux types de traceurs : ceux qui mesurent une valeur (et plus précisément la valeur maximale observée pour un paramètre du système) et ceux qui effectuent un suivi en enregistrant et restituant une série d'appels. Commençons par les traceurs de mesure puis nous examinerons par la suite les suivis de fonctions.

Traceurs de mesures

Pour avoir une bonne cohérence des résultats, il est généralement préférable de prendre des mesures sur un seul cœur à la fois. On peut sélectionner grâce au fichier **tracing_cpumask** l'ensemble des cœurs sur lesquels on souhaite réaliser une mesure. Comme son nom l'indique, il s'agit d'un masque binaire entre les différents CPU (cœur 0 = 1, cœur 1 = 2, cœur 2 = 4, cœur 3 = 8). Ici je vais me limiter au cœur 0, et écrire ainsi 1 dans le fichier.

```
# echo  1  >  tracing_cpumask
```

Temps de réveil

Lorsqu'une tâche s'endort, c'est toujours en attente d'un événement. Cet événement peut être interne au système (relâchement d'un mutex, activation d'un sémaphore, transmission d'un signal d'un processus à l'autre, passage de données dans une file de messages, etc.) ou externe (arrivée d'un octet sur un port RS-232 / i²c / SPI etc, réception d'une trame depuis le réseau, déplacement de la souris, pression sur une touche du clavier, alarme d'un timer matériel, etc.). Dans tous les cas, la tâche qui se réveille va être placée dans un état d'attente (nommé *Runnable*) jusqu'à ce que l'ordonnanceur décide qu'elle doit être effectivement exécutée par le processeur (état *Running*).

Il peut être intéressant, pour évaluer la réactivité du système, de mesurer combien de temps la tâche *Runnable* la plus prioritaire va devoir patienter avant d'être effectivement exécutée. Ce paramètre va dépendre du comportement de l'ordonnanceur lui-même, mais également des interruptions matérielles qui viennent s'intercaler dans le déroulement de la tâche considérée et dont les traitements peuvent être parfois longs.

Nous activons le traceur **wakeup** qui va mesurer cette durée pour chaque tâche *Runnable* qui devient la plus prioritaire.

```
# echo   wakeup   >   current_tracer
```

Après quelques instants, nous lisons, dans le fichier **tracing_max_latency**, la valeur maximale observée.

```
# cat  tracing_max_latency
68
```

Elle est exprimée en microsecondes. Une tâche prioritaire qui vient juste de se réveiller peut donc patienter jusqu'à 68 microsecondes avant d'être effectivement exécutée.

Nous allons charger un peu le système, en lançant une commande faisant un grand nombre d'appels au noyau : **top**, en lui demandant de se rafraîchir après un délai nul (donc en boucle permanente). Je vais également fixer la tâche sur le cœur zéro grâce à la commande **taskset**.

```
# taskset -c 0 top -d 0
```

En outre, depuis un autre poste j'envoie des *pings* en mode *flood* pour charger fortement l'interface réseau du Raspberry Pi et déclencher un grand nombre d'interruptions.

```
[cpb@LabPC3 :~]$ sudo ping -f 192.168.3.148
```

De surcroît, pour perturber le système et mesurer des valeurs réalistes, j'aime bien lancer en parallèle le petit programme ci-dessous qui saute régulièrement de cœur en cœur pour y exécuter des boucles actives pendant quelques secondes.

```
// CPU-jump! - Copyright 2013 Christophe Blaess
// This program is distributed under the terms of the GNU GPL.

#define _GNU_SOURCE  // sched_getcpu() is a GNU extension
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
  int i;
  int cpu;
  cpu_set_t cpu_set;
  for (;;) {
    for (cpu = 0; cpu < sysconf(_SC_NPROCESSORS_ONLN); cpu ++) {
      CPU_ZERO(&cpu_set);
      CPU_SET(cpu, &cpu_set);
      sched_setaffinity(0, sizeof(cpu_set), &cpu_set);
      fprintf(stderr, "[%d] I'm on CPU %d\n",
                  getpid(), sched_getcpu());
      for (i = 0; i < 100000000; i++)
        ;
    }
  }
  return EXIT_SUCCESS;
}
```

J'ai donc lancé, avant **top**, la ligne suivante :

```
# ~/cpu-jump &
```

Après une vingtaine de secondes d'affichage hystérique des paramètres du système par **top -d0**, j'arrête la commande (en pressant '**q**') et je vérifie la valeur maximale :

```
# cat tracing_ max_latency
84
```

Elle a peu augmenté, son ordre de grandeur est une petite centaine de microsecondes. C'est une valeur typique pour un système basé sur un noyau Linux standard (qui est optimisé pour les performances réseau et la puissance de calcul et pas du tout pour les aspects temps réel), on peut même s'attendre à ce qu'elle augmente jusqu'à quelques centaines de microsecondes dans les pires cas.

Nous pouvons obtenir plus d'informations sur les circonstances ayant conduit à cette latence maximale, en examinant le contenu du fichier **trace**.

```
# cat trace
# tracer: wakeup
#
# wakeup latency trace v1.1.5 on 4.1.17-v7+
# --------------------------------------------------------------------
# latency: 84 us, #29/29, CPU#0 | (M:desktop VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
```

```
#    | task: migration/0-9 (uid:0 nice:0 policy:1 rt_prio:99)
#    -----------------
#
#                _------=> CPU#
#               / _-----=> irqs-off
#              | / _----=> need-resched
#              || / _---=> hardirq/softirq
#              ||| / _--=> preempt-depth
#              |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
cpu-jump-663     0dn..    3us+:    663:120:R   + [000]     9:  0:R migration/0
cpu-jump-663     0dn..   15us : 0
cpu-jump-663     0dn..   18us : _raw_spin_unlock_irqrestore <-try_to_wake_up
cpu-jump-663     0dn..   21us : _raw_spin_unlock_irqrestore <-cpu_stop_queue_work
cpu-jump-663     0.n..   23us : wait_for_completion <-stop_one_cpu
cpu-jump-663     0.n..   26us : wait_for_common <-wait_for_completion
cpu-jump-663     0.n..   28us : _cond_resched <-wait_for_common
cpu-jump-663     0.n..   30us : __schedule <-preempt_schedule_common
cpu-jump-663     0.n..   32us : rcu_note_context_switch <-__schedule
cpu-jump-663     0.n..   34us : _raw_spin_lock_irq <-__schedule
cpu-jump-663     0dn..   37us : pick_next_task_stop <-__schedule
cpu-jump-663     0dn..   39us : put_prev_task_fair <-pick_next_task_stop
cpu-jump-663     0dn..   42us : put_prev_entity <-put_prev_task_fair
cpu-jump-663     0dn..   44us : update_curr <-put_prev_entity
cpu-jump-663     0dn..   46us : update_min_vruntime <-update_curr
cpu-jump-663     0dn..   49us : cpuacct_charge <-update_curr
cpu-jump-663     0dn..   51us : __enqueue_entity <-put_prev_entity
cpu-jump-663     0dn..   54us : __compute_runnable_contrib <-put_prev_entity
cpu-jump-663     0dn..   56us : __update_entity_load_avg_contrib <-put_prev_entity
cpu-jump-663     0dn..   59us : __update_entity_utilization_avg_contrib <-put_prev_entity
cpu-jump-663     0dn..   61us : put_prev_entity <-put_prev_task_fair
cpu-jump-663     0dn..   63us : update_curr <-put_prev_entity
cpu-jump-663     0dn..   65us : update_min_vruntime <-update_curr
cpu-jump-663     0dn..   68us : __enqueue_entity <-put_prev_entity
cpu-jump-663     0dn..   70us : __compute_runnable_contrib <-put_prev_entity
cpu-jump-663     0dn..   72us : __update_entity_load_avg_contrib <-put_prev_entity
cpu-jump-663     0dn..   75us : __update_entity_utilization_avg_contrib <-put_prev_entity
cpu-jump-663     0d...   80us : __schedule
cpu-jump-663     0d...   82us :    663:120:R ==> [000]     9:  0:R migration/0
```

On voit que la latence est due à la migration de notre tâche **`cpu-jump`** (de PID 663) par le *thread* kernel **`migration/0`** (PID 9). Sur la droite du listing on voit la succession horodatée des fonctions du noyau ayant été impliquées dans cette latence. La progression de l'horodatage est assez régulière, aucune fonction ne

prend de temps excessif (plusieurs dizaines ou centaines de microsecondes) comme on pourrait l'observer dans des cas pathologiques.

L'étude précise des fonctions noyau indiquées dépasse le cadre de cet article. Pour un exemple d'analyse, je conseillerai de ce reporter à l'article [Kobayashi 2012].

J'ai réalisé l'expérience ci-dessus successivement sur les quatre cœurs du processeur du Raspberry Pi 2, pendant des durées longues (plusieurs heures) et une charge importante, obtenant des résultats assez homogènes :

Cœur	0	1	2	3
Latence max. traceur **`wakeup`**	189	125	111	134

Les quatre cœurs du Raspberry Pi 2 n'ont pas un comportement homogène vis-à-vis du traitement des interruptions, comme le montre l'examen du fichier **`/proc/interrupts`**.

```
# cat /proc/interrupts
       CPU0       CPU1       CPU2       CPU3
 16:          0          0          0          0   ARMCTRL  16 Edge      bcm2708_fb dma
 20:       1507          0          0          0   ARMCTRL  20 Edge      DMA IRQ
 32:     111881          0          0          0   ARMCTRL  32 Edge      dwc_otg, dwc_otg_pcd, dwc_otg_hcd:usb1
 49:          0          0          0          0   ARMCTRL  49 Edge      3f200000.gpio:bank0
 50:          0          0          0          0   ARMCTRL  50 Edge      3f200000.gpio:bank1
 65:         50          0          0          0   ARMCTRL  65 Edge      3f00b880.mailbox
 66:          2          0          0          0   ARMCTRL  66 Edge      VCHIQ doorbell
 75:          1          0          0          0   ARMCTRL  75 Edge
 77:        135          0          0          0   ARMCTRL  77 Edge      DMA IRQ
 83:      17928          0          0          0   ARMCTRL  83 Edge      uart-pl011
 84:       3944          0          0          0   ARMCTRL  84 Edge      mmc0
 96:          0          0          0          0   ARMCTRL  96 Edge      arch_timer
 97:       4919       3434       2117       1309   ARMCTRL  97 Edge      arch_timer
FIQ:              usb_fiq
IPI0:          0          0          0          0  CPU wakeup interrupts
IPI1:          0          0          0          0  Timer broadcast interrupts
IPI2:       3045       2976       3525       3935  Rescheduling interrupts
IPI3:          7          8          6          7  Function call interrupts
IPI4:          2          2          1          1  Single function call interrupts
IPI5:          0          0          0          0  CPU stop interrupts
IPI6:          0          0          0          0  IRQ work interrupts
IPI7:          0          0          0          0  completion interrupts
Err:           0
```

Nous voyons que la majorité des interruptions est gérée sur le cœur 0, aussi est-il normal que la latence y soit un peu plus longue que sur les autres cœurs.

On pourrait obtenir de la même manière les latences mesurées par les traceurs **`wakeup_rt`** et **`wakeup_dl`** qui filtrent les tâches concernées.

Masquage d'interruptions

Il est très utile, dans une optique temps réel, de connaître le temps maximal pendant lequel les interruptions sont masquées. Ceci donne une idée du temps de réponse que l'on peut attendre à un événement externe.

En nous concentrant toujours sur le cœur 0, nous activons ensuite le traceur **`irqsoff`** en écrivant son nom

dans **current_tracer** :

```
# echo  irqsoff  >  current_tracer
```

Au bout de quelques instants, nous observons la plus longue latence :

```
# cat  tracing_max_latency
7718
```

La latence étant mesurée en micro-secondes ceci correspond donc à 7,7 millisecondes. Cette valeur peut sembler très importante, mais il faut prendre en compte deux choses :

- il s'agit d'un noyau Linux *Vanilla*, non optimisé pour le temps de réponse aux sollicitations externes. On aurait des résultats probablement bien meilleurs (de l'ordre de la centaine de microsecondes) avec un noyau sur lequel est appliqué le patch **PREEMPT_RT**,

- le suivi des interruptions dans le noyau modifié pour intégrer les drivers spécifiques Raspberry Pi me semble un peu douteux, pour preuve la trace des fonctions ayant conduit à cette latence maximale :

```
# cat trace
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 4.1.17-v7+
# --------------------------------------------------------------------
# latency: 7718 us, #6/6, CPU#0 | (M:desktop VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: swapper/0-0 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: rcu_idle_enter
#  => ended at:   arch_cpu_idle
#
#
#                  _------=> CPU#
#                 / _-----=> irqs-off
#                | / _----=> need-resched
#                || / _---=> hardirq/softirq
#                ||| / _--=> preempt-depth
#                |||| /     delay
#  cmd     pid   ||||| time  |   caller
#     \   /      |||||  \    |   /
  <idle>-0       0d...    0us : rcu_idle_enter
  <idle>-0       0d...    2us : rcu_eqs_enter_common <-rcu_idle_enter
  <idle>-0       0d...    4us#: arch_cpu_idle <-cpu_startup_entry
  <idle>-0       0d... 7718us : arch_cpu_idle
  <idle>-0       0d... 7720us+: trace_hardirqs_on <-arch_cpu_idle
  <idle>-0       0d... 7771us : <stack trace>
 => cpu_startup_entry
 => rest_init
 => start_kernel
#
```

La fonction ayant pris un temps de 7,7 millisecondes est **arch_cpu_idle** qui assure la mise au repos du processeur lorsqu'il n'y a rien à faire. La prise en compte de cette durée ne me paraît pas légitime. Sur

d'autres processeurs (*Allwiner A20* par exemple), les latences que j'ai observées avec un noyau Linux *Vanilla* sont plutôt de 500 à 700 microsecondes.

Les traceurs de suivi

Il existe plusieurs traceurs qui assurent un suivi des fonctions invoquées dans le noyau. Le premier que nous verrons s'intéresse spécifiquement aux périphériques blocs (disques, flashs Nand, cartes SD, clés USB, etc.), les autres permettent le suivi de n'importe quel routine du kernel.

Entrées-sorties sur périphériques blocs

Le fonctionnement du traceur de périphériques blocs est très simple, cependant son utilité est limitée à des cas plus rares que les autres. On commence par activer ce type de suivi :

```
# echo  blk  >  /sys/kernel/debug/tracing/current_tracer
```

Puis on active le suivi sur un périphérique bloc. Par exemple la carte SD du Raspberry Pi : **/dev/mmcblk0p2**

```
# echo  1  >  /sys/block/mmcblk0/mmcblk0p2/trace/enable
```

Et l'on observe les résultats dynamiquement avec :

```
# cat  /sys/kernel/debug/tracing/trace_pipe
       bash-664  [000] ....  487.628334: 179,2   A   W 7581720 + 8 <- (179,2) 7450648
       bash-664  [000] ....  487.628390: 179,2   Q   W 7581720 + 8 [bash]
       bash-664  [000] ....  487.628410: 179,2   G   W 7581720 + 8 [bash]
       bash-664  [000] ....  487.628420: 179,2   P   N [bash]
       bash-664  [000] ....  487.628448: 179,2   I   W 7581720 + 8 [bash]
       bash-664  [000] ....  487.628457: 179,2   U   N [bash] 1
     mmcqd/0-54  [000] ....  487.628847: 179,2   D   W 7581720 + 8 [mmcqd/0]
     mmcqd/0-54  [000] ....  487.632242: 179,2   C   W 7581720 + 8 [0]
```

On trouve dans les traces affichées, successivement :

- le nom du processus ou du *thread kernel* à l'origine de l'action, suivi de son PID,

- le numéro du cœur (entre crochets) sur lequel l'action s'exécute,

- l'horodatage de l'opération en secondes et microsecondes,

- les numéros majeur et mineur du périphérique bloc concerné (**179** et **0**), que l'on peut interpréter en observant le résultat de « **ls -l /dev** » et en voyant qu'ils correspondent à **mmcblk0**, la carte SD.

- Un indicateur du type d'action sur une lettre. Il y a une dizaine de types d'action, certains sont complexes et dépassent le cadre de cet article. Voici les plus courants, que l'on peut observer dans l'exemple ci-dessus.

A	*Remap*	L'opération est déférée à un autre périphérique bloc.
C	*Complete*	Fin de l'opération d'entrée-sortie.
D	*Driver*	Envoi de la requête au driver bas-niveau.
G	*Get request*	Allocation d'une structure pour contenir la requête.
I	*Insert*	Insertion d'une requête dans une file.
P	*Plug*	Ajout d'une file de requête dans l'ordonnanceur d'I/O.
Q	*Queue*	Préparation des données de la requête.
U	*Unplug*	Extraction des requêtes de la file.

- Le cas échéant, l'offset (compté en blocs) du début de l'accès ainsi que le nombre de blocs concernés

- le nom du processus à l'origine de l'événement.

L'utilisation de ce traceur est surtout réservée aux développeurs de drivers de périphériques blocs et aux administrateurs désireux de maîtriser complètement le fonctionnement de leur système pour l'ajuster au mieux. On trouve peu de documentation sur ce traceur, je recommanderai tout de même [Axboe 2007] qui présente un outil nommé **blktrace** s'appuyant sur des mécanismes similaires. Le parallèle permet de

comprendre les résultats du traceur **blk** de Ftrace.

Les traceurs de fonctions

Les deux traceurs de suivis de fonctions (**function** et **function_graph**) agissent de la même manière : ils enregistrent des points de passage dans les routines du noyau. Leur différence est minime : **function** n'enregistre que les points d'entrée dans ces routines alors que **function_graph** mémorise également les points de sortie, permettant de reconstituer l'imbrication des appels de fonctions intermédiaires.

Suivi d'appel système

Bien entendu ces traceurs nous permettent de limiter la liste des fonctions qui nous intéressent pour ne pas être surchargés par une énorme quantité d'informations inexploitables. Dans un premier temps nous pouvons suivre par exemple toutes les invocations de l'appel système **open()**, probablement l'un des plus fréquemment demandé.

Pour cela, nous examinons les filtres disponibles dans le fichier **available_filter_functions** :

```
# cat  available_filter_functions
asm_do_IRQ
handle_fiq_as_nmi
do_IPI
run_init_process
[…]
nl80211_parse_random_mac [cfg80211]
nl80211_put_signal.part.3 [cfg80211]
#
```

Problème : il y a plus de vingt-deux mille fonctions du noyau proposées dans ce fichier. Nous pouvons rechercher avec **grep** toutes celles qui contiennent **sys_open()**, le nom interne de l'appel-système.

```
# grep  sys_open  available_filter_functions
do_sys_open
proc_sys_open
```

Après un coup d'œil aux sources du noyau à l'aide du site [LXR] nous voyons que c'est bien la première fonction qui nous intéresse.

```
# echo  do_sys_open  >  set_ftrace_filter
# echo  function  >  current_tracer
```

Le suivi est lancé. Après quelques commandes simples (**ls** par exemple), nous examinons les résultats dans le fichier **trace**.

```
# cat  trace
# tracer: function
#
# entries-in-buffer/entries-written: 45/45   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
```

```
      systemd-1    [002] ....   582.607484: do_sys_open <-SyS_open
      systemd-1    [002] ....   582.607938: do_sys_open <-SyS_open
         ls-648   [002] ....   591.031477: do_sys_open <-SyS_open
         ls-648   [002] ....   591.031584: do_sys_open <-SyS_open
         ls-648   [002] ....   591.031864: do_sys_open <-SyS_open
  [...]
        bash-601   [000] ....   593.074110: do_sys_open <-SyS_open
        bash-601   [000] ....   594.313399: do_sys_open <-SyS_openat
        bash-601   [000] ....   594.315076: do_sys_open <-SyS_openat
 […]
#
```

L'examen du fichier trace nous donne un historique instantané du suivi. Nous pouvons également avoir un suivi dynamique, qui s'étend au gré des nouvelles actions en consultant le contenu de **trace_pipe**.

```
# cat  trace_pipe
     [...]
        ntpd-466   [001] ....   675.418646: do_sys_open <-SyS_open
      systemd-1    [002] ....   682.627250: do_sys_open <-SyS_open
      systemd-1    [002] ....   692.627248: do_sys_open <-SyS_open
      systemd-1    [002] ....   722.627353: do_sys_open <-SyS_open
      systemd-1    [002] ....   732.627352: do_sys_open <-SyS_open
  (Contrôle-C)
```

On peut, inversement, choisir de suivre toutes les fonctions du noyau en se limitant à celles invoquées pour le compte d'un processus donné. Pour cela on désactive le filtre de fonctions :

```
# echo  nop  >  current_tracer
# cat  set_ftrace_filter
do_sys_open
# echo  >  set_ftrace_filter
# cat  set_ftrace_filter
#### all functions enabled #####
#
```

Ensuite on inscrit le PID du shell courant (toujours contenu dans la variable spéciale **$$**) dans le fichier des processus à suivre :

```
# echo  $$  >  set_ftrace_pid
```

Enfin, on lance le suivi. Pour éviter d'accumuler une liste de fonctions à chaque touche pressée, je préfère utiliser une seule ligne de commande qui active le suivi, lance une commande (ici la simple commande date) et désactive le suivi.

```
# echo  0  >  tracing_on
# echo  function  >  current_tracer
# echo  1  >  tracing_on ;  date ;  echo  0  >  tracing_on
```

L'ensemble des fonctions noyau exécutées est listé dans le fichier **trace**. Il est néanmoins gigantesque (plus de 25 000 lignes) même pour une commande aussi simple. C'est pour cela qu'il est généralement nécessaire de filtrer les fonctions intéressantes.

Débogage de driver

La destination première du système Ftrace est de pouvoir déboguer le code développé pour un driver en

assurant un suivi. Le fichier **`trace`** donne la liste de toutes les fonctions invoquées jusqu'alors, et le fichier **`trace_pipe`** permet d'avoir une liste dynamique, les fonctions étant présentées au fur et à mesure de leurs appels. Ceci rappelle vaguement le fonctionnement d'un débogueur pas-à-pas.

Nous allons écrire un petit driver minimal de la classe de périphériques **`misc.`** Ses méthodes **`open()`**, **`close()`**, **`read()`** et **`write()`** seront implémentées de manière très simple. Il permettra d'écrire un message dans un petit *buffer* de taille fixe et de le récupérer ensuite. Le *buffer* est initialisé avec un message de bienvenue au chargement du module.

Le code du driver est assez classique :

```
// mini-driver - Copyright 2016 Christophe Blaess
// This program is distributed under the terms of the GNU GPL.

#include <linux/fs.h>
#include <linux/miscdevice.h>
#include <linux/module.h>
#include <asm/uaccess.h>

#define MINIDRV_BUFFER_SIZE  32

static char minidrv_buffer[MINIDRV_BUFFER_SIZE + 1];

static int minidrv_open(struct inode * ind, struct file * filp)
{
 return 0;
}


static int minidrv_release(struct inode * ind, struct file * filp)
{
 return 0;
}

static ssize_t minidrv_read(struct file * filp, char * buffer,
                            size_t max_lg, loff_t * offset)
{
 int lg;

 lg = strlen(minidrv_buffer) - (*offset);
 if (lg <= 0)
   return 0;
 if (max_lg < lg)
   lg = max_lg;
 if (copy_to_user(buffer, & minidrv_buffer[* offset], lg) != 0)
   return -EFAULT;
 *offset += lg;
 return lg;
}

static ssize_t minidrv_write(struct file * filp, const char * buffer,
                             size_t lg, loff_t * offset)
{
 if (lg > MINIDRV_BUFFER_SIZE)
   lg = MINIDRV_BUFFER_SIZE;
 if (copy_from_user(minidrv_buffer, buffer, lg) != 0)
   return -EFAULT;
 minidrv_buffer[lg] = '\0';
 return lg;
}

static struct file_operations minidrv_fops = {
 .owner   =  THIS_MODULE,
 .open    =  minidrv_open,
 .release =  minidrv_release,
 .read    =  minidrv_read,
 .write   =  minidrv_write,
};

static struct miscdevice minidrv_driver = {
```

```
    .minor        = MISC_DYNAMIC_MINOR,
    .name         = THIS_MODULE->name,
    .fops         = &minidrv_fops,
};

static int __init minidrv_init (void)
{
 strcpy(minidrv_buffer, "Hello World!\n");
 return misc_register(&minidrv_driver);
}

static void __exit minidrv_exit (void)
{
 misc_deregister(& minidrv_driver);
}

module_init(minidrv_init);
module_exit(minidrv_exit);

MODULE_AUTHOR("Christophe Blaess <Christophe.Blaess@Logilin.fr>");
MODULE_LICENSE("GPL");
```

Pour pouvoir compiler ce module, il est nécessaire de disposer des fichiers d'en-tête du noyau et de configuration. Ceci est généralement fourni par les distributions dans un package **`kernel-devel`** ou un nom approchant. Malheureusement ce n'est pas le cas pour la distribution Raspbian. La solution la plus simple est de recompiler le noyau. Ce n'est pas difficile, mais nécessite un peu de temps et de place sur la carte SD (voir encadré « Compilation du noyau du Raspberry Pi 2 »).

Nous allons suivre ses fonctions. Pour cela il faut commencer par charger le module.

```
# echo  nop  >  /sys/kernel/debug/tracing/current_tracer
# insmod  mini-driver.ko
```

Vérifions si les fonctions de notre module sont bien accessibles au suivi :

```
# grep  minidrv  /sys/kernel/debug/tracing/available_filter_functions
minidrv_open [mini_driver]
minidrv_release [mini_driver]
minidrv_write [mini_driver]
minidrv_read [mini_driver]
```

Nous allons donc les inscrire dans le filtre, puis activer le suivi.

```
# echo  'minidrv_*'  >  /sys/kernel/debug/tracing/set_ftrace_filter
# echo  function  >  /sys/kernel/debug/tracing/current_tracer
```

À présent, utilisons un peu les méthodes de notre module :

```
# cat  /dev/mini_driver
Hello World!
# echo  "Bonjour à tous"  > /dev/mini_driver
Bonjour à tous
# cat  /dev/mini_driver
Bonjour à tous
```

Et vérifions les traces laissées par nos fonctions ;

```
# cat  /sys/kernel/debug/tracing/trace
# tracer: function
#
# entries-in-buffer/entries-written: 11/11   #P:4
#
#                              _-----=> irqs-off
```

```
#                              / _----=> need-resched
#                             | / _---=> hardirq/softirq
#                             || / _--=> preempt-depth
#                             ||| /     delay
#          TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#             | |       |   ||||       |         |
           cat-2827  [002] ....  4065.304117: minidrv_open <-misc_open
           cat-2827  [002] ....  4065.304194: minidrv_read <-__vfs_read
           cat-2827  [002] ....  4065.304248: minidrv_read <-__vfs_read
           cat-2827  [002] ....  4065.304299: minidrv_release <-__fput
          bash-1063  [000] ....  4087.307571: minidrv_open <-misc_open
          bash-1063  [000] ....  4087.307636: minidrv_write <-__vfs_write
          bash-1063  [000] ....  4087.307656: minidrv_release <-__fput
           cat-2828  [002] ....  4091.084123: minidrv_open <-misc_open
           cat-2828  [002] ....  4091.084202: minidrv_read <-__vfs_read
           cat-2828  [002] ....  4091.084255: minidrv_read <-__vfs_read
           cat-2828  [002] ....  4091.084311: minidrv_release <-__fput
#
```

Nous retrouvons bien la succession de nos méthodes **`open()`**, **`read()`**, **`write ()`** et **`release()`** – cette dernière étant invoquée lors de l'appel système **`close()`**.

Ordonnancement et commutations

Une autre utilisation fréquente de Ftrace est l'observation des commutations entre tâches réalisées par l'ordonnanceur (la fonction **`schedule()`** du kernel). Ceci fournit un aperçu de l'exécution du code, des tranches de temps accordées respectivement à chaque tâche, des temps de réveil sur les objets de synchronisation (sémaphores, mutex, variables conditions), etc.

Nous allons en voir un exemple simple où deux processus (ordonnancés en temps partagé) exécutent des boucles actives consommant du temps CPU. Ces deux processus (père et fils) sont placés sur le même cœur de processeur. Le processus parent démarre ses boucles deux secondes après son fils, mais ce dernier a une priorité temps-partagé plus faible (valeur de **`nice`** plus élevée).

Le programme contient directement les commandes nécessaires pour configurer Ftrace et s'exécuter sur un seul cœur. Il a été écrit pour un Raspberry Pi 2, sur un autre processeur il faudra ajuster la boucle active pour qu'elle dure environ deux secondes ainsi éventuellement que le numéro de cœur.

```
// test-commutation-01 - Copyright 2016 Christophe Blaess
// This program is distributed under the terms of the GNU GPL.

#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

#define CPU  2

void start_ftrace(void)
{
  FILE * fp;

  system("echo nop > /sys/kernel/debug/tracing/current_tracer");
  system("echo __schedule > /sys/kernel/debug/tracing/set_ftrace_filter");
  if ((fp=fopen("/sys/kernel/debug/tracing/set_ftrace_pid", "w")) != NULL) {
    fprintf(fp, "%d", getpid());
    fflush(fp);
```

```
    fprintf(fp, "%d", getppid());
    fclose(fp);
  }
  if ((fp=fopen("/sys/kernel/debug/tracing/tracing_cpumask", "w")) != NULL) {
    fprintf(fp, "%d\n", 1<<CPU);
    fclose(fp);
  }
  system("echo function >/sys/kernel/debug/tracing/current_tracer");
  system("echo 1 >/sys/kernel/debug/tracing/tracing_on");
}

void stop_ftrace(void)
{
  system("echo 0 >/sys/kernel/debug/tracing/tracing_on");
}

void boucle_active(void)
{
  int i;
  for (i = 0; i < 150000000; i ++)
    ;
}

int main(void)
{
  cpu_set_t cpu_set;
  CPU_ZERO(&cpu_set);
  CPU_SET(CPU, &cpu_set);
  sched_setaffinity(0, sizeof(cpu_set), &cpu_set);

  if (fork() == 0) {
    start_ftrace();
    sleep(2);
    nice(+5);
    boucle_active();
    exit(0);
  } else {
    sleep(4);
    boucle_active();
    wait(NULL);
    stop_ftrace();
  }
  return EXIT_SUCCESS;
}
```

Lançons ce programme et voyons les résultats :

```
# ./test-commutations-01
```

Le programme s'exécute pendant huit secondes environ.

```
# cat  /sys/kernel/debug/tracing/trace
[...]
          sh-3948  [002] .... 65830.575917: __schedule <-schedule
 test-commutatio-3944  [002] .... 65830.576198: __schedule <-schedule
 test-commutatio-3944  [002] dn.. 65832.576440: __schedule <-schedule
 test-commutatio-3944  [002] dn.. 65832.584368: __schedule <-schedule
 test-commutatio-3944  [002] dn.. 65832.614363: __schedule <-schedule
 test-commutatio-3944  [002] dn.. 65833.194347: __schedule <-schedule
```

Le programme débute, le processus enfant configure Ftrace avec la commande **`system()`** – d'où la présence d'un shell de PID 3948 – puis s'endort deux secondes pour se réveiller à la seconde 65832. Le parent a commencé son sommeil de quatre secondes un peu plus tôt et se réveille donc à la seconde 65833.

Ensuite, nous voyons les commutations entre eux :

```
 test-commutatio-3944  [002] dn.. 65833.954321: __schedule <-schedule
```

```
test-commutatio-3943  [002] dn.. 65833.974316: __schedule <-schedule
test-commutatio-3943  [002] dn.. 65833.984320: __schedule <-schedule
test-commutatio-3944  [002] dn.. 65833.994318: __schedule <-schedule
test-commutatio-3943  [002] dn.. 65834.014316: __schedule <-schedule
test-commutatio-3943  [002] dn.. 65834.024320: __schedule <-schedule
test-commutatio-3944  [002] dn.. 65834.034322: __schedule <-schedule
test-commutatio-3943  [002] dn.. 65834.054316: __schedule <-schedule
test-commutatio-3943  [002] dn.. 65834.064317: __schedule <-schedule
   […]
```

Le processus parent a une priorité temps-partagé normale alors que le fils a une priorité plus faible (**`nice`** plus élevé). Il est donc normale que le parent s'exécute plus fréquemment que son enfant.

On imagine assez bien les informations que ces traces nous offrent ; elles pourraient être plus précises encore si les tâches étaient synchronisées entre elles (par exemple avec un sémaphore).

Utilisation de trace-cmd

La manipulation des fichiers du répertoire **`/sys/kernel/debug/tracing`** est un peu fastidieuse. Dans l'exemple précédent, pour éviter toute erreur de saisie, les opérations ont été directement intégrées dans le code exécutable, mais cela n'est pas possible lorsqu'il s'agit de vérifier le comportement d'un programme déjà écrit et compilé.

Afin d'éviter de manipuler directement les fichiers **`/sys/kernel/debug/tracing/`** un utilitaire frontal, nommé **`trace-cmd`**, a été créé pour proposer une unique commande de dialogue avec le système Ftrace. En outre **`trace-cmd`** utilise un format de fichier spécifique (par défaut **`trace.dat`**) pour encapsuler les paramètres utilisés et les traces de suivi.

Il faut tout d'abord l'installer :

```
# apt-get  install  trace-cmd
```

On lance la commande en lui passant un premier argument indiquant l'action à réaliser. La page de manuel de **`trace-cmd`** liste une quinzaine d'actions ; en voici quelques unes parmi les plus courantes :

Action	Signification
`record`	Démarrer un suivi et exécuter une commande en enregistrant dans le fichier **`trace.dat`** les informations fournies par Ftrace. On ajoute des arguments supplémentaires sur la ligne de commande pour paramétrer plus finement Ftrace (traceur, filtre, masque CPU, PID, etc.)
`start` **`stop`**	Démarrer et arrêter le suivi, en laissant les informations dans le buffer de Ftrace. Comme pour **`record`**, des paramètres supplémentaires affinent le comportement de Ftrace.
`extract`	Extraire les données précédemment enregistrées par Ftrace et les stocker dans un fichier **`trace.dat`**. Cette action est généralement exécutée après **`stop`**.
`report`	Afficher de manière lisible les données contenues dans le fichier **`trace.dat`**.

Nous allons utiliser **`trace-cmd record`** pour lancer un exécutable en enregistrant ses traces. Pour cela nous reprenons le même programme que précédemment en supprimant tout le code d'instrumentation et en le résumant à l'essentiel :

```
// test-commutation-02 - Copyright 2016 Christophe Blaess
// This program is distributed under the terms of the GNU GPL.

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

void boucle_active(void)
{
  int i;
  for (i = 0; i < 150000000; i ++)
    ;
}
```

```
int main(void)
{
 if (fork() == 0) {
   sleep(2);
   nice(+5);
   boucle_active();
   exit(0);
 } else {
   sleep(4);
   boucle_active();
   wait(NULL);
 }
 return EXIT_SUCCESS;
}
```

La commande lancée est la suivante :

```
# taskset -c 2 trace-cmd record -p function -l __schedule -M 4 -F -c ./test-commutations-02
```

Détaillons-là :

- « **taskset -c 2** » : exécuter du reste de la ligne sur le cœur de processeur numéro 2.
- « **trace-cmd record** » : démarrer et enregistrer les données dans **trace.dat**.
- « **-p function** » : utiliser le traceur **function**.
- « **-l __schedule** » : filtrer les traces pour ne s'intéresser qu'à la fonction **__schedule()** du noyau.
- « **-F** » : ne suivre que les événements concernant la commande venant à la suite.
- « **-c** » : suivre également les éventuels processus enfants de cette commande.
- « **-M 4** » : ne s'intéresser qu'aux CPU dans ce masque binaire, ici le cœur numéro 2.

Après quelques instants, **trace-cmd** nous affiche quelques statistiques dont :

```
[…]
CPU: 2
entries: 0
overrun: 0
commit overrun: 0
bytes: 2116
oldest event ts: 69897.269489
now ts: 69903.832032
dropped events: 0
read events: 103
[…]
```

On a bien enregistré (**read events**) des événements s'étant produits sur le cœur numéro 2.

Pour consulter les résultats, on peut utiliser l'action **report** de **trace-cmd**.

```
# trace-cmd report
version = 6
CPU 0 is empty
CPU 1 is empty
CPU 3 is empty
cpus=4
 test-commutatio-4138  [002] 69897.269489: function:             __schedule
 test-commutatio-4138  [002] 69897.275190: function:             __schedule
 test-commutatio-4139  [002] 69897.276006: function:             __schedule
```

```
test-commutatio-4139  [002] 69897.276736: function:          __schedule
  […]
test-commutatio-4138  [002] 69901.326643: function:          __schedule
test-commutatio-4139  [002] 69901.336632: function:          __schedule
test-commutatio-4138  [002] 69901.356646: function:          __schedule
test-commutatio-4138  [002] 69901.366672: function:          __schedule
```

Les résultats sont comparables à la consultation de directe de Ftrace comme nous l'avons vu précédemment. Toutefois, à moins de chercher un événement s'étant produit à un instant bien précis, la consultation de ces données est fastidieuse et peu explicite.

Utilisation de kernelshark

Pour rendre les résultats de **`trace-cmd`** plus lisibles et attrayants un utilitaire graphique nommé **`kernelshark`** a été développé. Son nom évoque celui du célèbre **`wireshark`**, mais il faut quand même reconnaître que son domaine d'utilisation est beaucoup plus limité.

J'utilise mon Raspberry Pi 2 comme un système embarqué classique, c'est-à-dire au travers d'une liaison série et/ou une connexion SSH. Je n'ai donc pas d'interface graphique sur ce système et je vais devoir utiliser **`kernelshark`** sur un PC distant. Néanmoins cet outil peut parfaitement s'installer sur une distribution Raspbian, avec « **`apt-get install kernelshark`** » et peut donc être utilisé localement sur le Raspberry Pi 2.

Je vais transférer le fichier **`trace.dat`** obtenu sur le poste distant :

```
# scp  trace.dat  cpb@192.168.3.103:~/
```

et sur ce PC je vais lancer **`kernelshark`** dans un environnement graphique

```
[cpb@LabPC3 :~]$ kernelshark
```

J'obtiens alors une fenêtre me présentant graphiquement mes résultats.

La partie graphique supérieure représente l'activité sur les différents CPU (nous l'avons filtré pour ne nous intéresser qu'au cœur 2). Nous pouvons zoomer sur cette fenêtre en cliquant sur l'instant de début et en glissant la souris vers la droite du graphique jusqu'à l'instant d'arrivée. Le zoom arrière s'obtient en cliquant et glissant la souris vers la *gauche* du graphique.

En passant la souris au dessus d'un événement celui-ci est décrit dans une petite fenêtre pop-up.

La liste dans la partie inférieure de la fenêtre regroupe tous les événements du fichier **`trace.dat`** ; en cliquant sur l'un d'eux un curseur s'affiche dans la zone graphique.

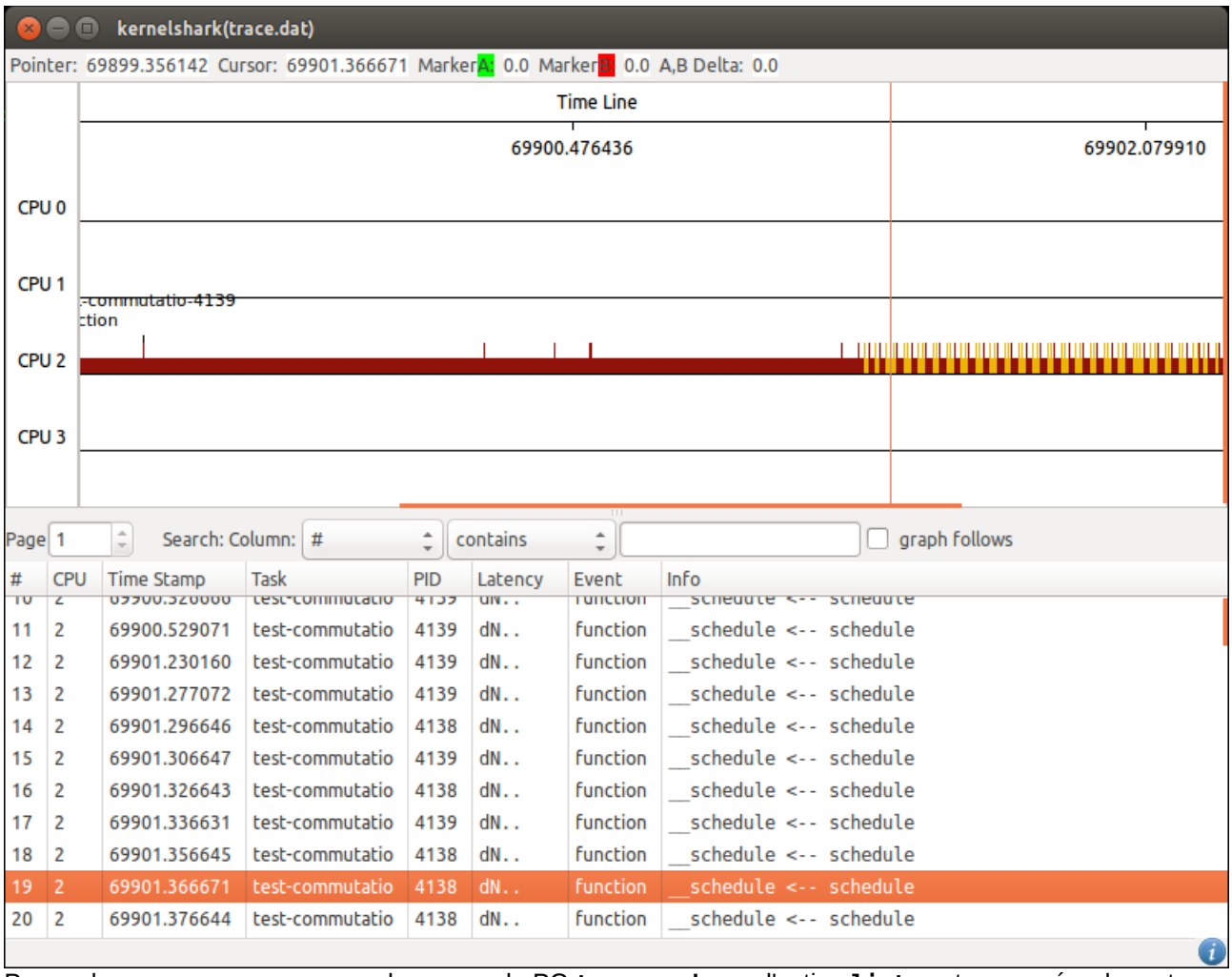
Kernelshark propose des possibilités de filtrage, de sélection, etc. dans ses divers menus. Il faut néanmoins signaler que la stabilité de cette application n'est pas exemplaire, et que des crashs surviennent parfois.

On peut néanmoins apprécier son aspect dynamique pour l'analyse des enregistrements de **`trace-cmd`**.

> On notera qu'il est possible également de visualiser les résultats de Ftrace en employant l'utilitaire **`gnuplot`**. L'avantage est de mieux maîtriser le format de sortie (dimensions, résolutions, etc.) et de pouvoir automatiser le traitement dans un script lorsqu'on désire avoir une représentation automatique en continu ou en étant invoqué depuis une IHM distante (serveur web, etc.).
>
> L'inconvénient de cette approche est la complexité pour arriver à zoomer sur les portions de traces intéressantes, et le manque d'interactions dynamiques dans son utilisation.

trace-cmd et les systèmes embarqués

Le fichier **`trace.dat`** peut rapidement représenter plusieurs dizaines de mégaoctets. Le stockage local n'est donc pas une solution envisageable pour les systèmes embarqués restreints. Heureusement **`trace-cmd`** nous permet de transférer directement les données vers un système hôte distant.



Pour cela, nous commençons par lancer sur le PC **`trace-cmd`** avec l'action **`listen`** et un numéro de port TCP/IP arbitrairement choisi (ici **`1234`**)

```
[cpb@LabPC3:~]$ trace-cmd  listen  -p 1234
```

La commande reste en attente. Pendant ce temps nous lançons de nouveau **`trace-cmd record`** sur le Raspberry Pi 2, en lui ajoutant l'option **`-N`** suivi de l'adresse IP de l'hôte de développement et le numéro de port TCP choisi :

```
# taskset  -c 2  trace-cmd record  -p function  -l __schedule  -M 4  -F -c  -N
192.168.3.103:1234  ./test-commutations-02
```

Sur le PC, nous voyons que **`trace-cmd`** a bien reçu la demande de connexion, des données apparaissent :

```
connected!
Connected with 192.168.3.148:57521
cpus=4
pagesize=4096
[...]
connected!
```

La commande est terminée sur le Raspberry Pi, j'arrête celle du PC en pressant *Contrôle-C* :

```
^C
[cpb@LabPC3:~]$ ls
trace.192.168.3.148:57521.dat
```

Le fichier de trace est identifié avec l'adresse du Raspberry Pi et le numéro de port qui a été utilisé pour ce transfert (choisi par le noyau automatiquement). Nous pouvons alors appeler **`kernelshark`** en lui précisant d'afficher le contenu de ce fichier de traces :

```
[cpb@LabPC3:~]$ kernelshark  -i trace.192.168.3.148\:57521.dat
```

La représentation graphique est similaire à la précédente.

Conclusion

Le système Ftrace est très puissant. Il propose de nombreuses autres possibilités que nous n'avons pas traitées (par exemple le suivi de la taille de pile, la sélection directe d'événements, l'émission de traces depuis un programme utilisateur, etc.). Son interface par l'intermédiaire de fichier de **`debugfs`** est un peu aride, mais des efforts tendent à le rendre plus accessible. Les outils **`trace-cmd`** et **`kernelshark`** proposent une première approche mais on peut espérer voir des projets plus complets par l'avenir utilisant par exemple une interface Web ou un *plug-in* Eclipse.

Compilation du noyau du Raspberry Pi

Pour recompiler le noyau d'un système embarqué, la méthode normale consiste à préparer un environnement de compilation sur un poste de développement, généralement un PC, incluant une chaîne de compilation croisée (*cross toolchain*) afin de générer l'image du noyau pour la cible (le Raspberry Pi 2). J'ai décrit cette méthode par exemple dans l'article [Blaess 2015].

Pour recompiler le noyau du Raspberry Pi 2, on peut simplifier cette tâche en tirant parti du fait que la distribution Raspbian inclut une chaîne de compilation GCC complète, et réaliser le travail directement sur le Raspberry Pi. Ceci est au prix d'un temps de compilation plus long bien entendu.

Si vous avez du temps devant vous pour le téléchargement, vous pouvez rapatrier tout l'historique du dépôt Github du noyau pour Raspberry Pi et sélectionner la version exacte qui convient :

```
$ git clone http://github.com/raspberrypi/linux rpi-linux
$ cd rpi-linux
$ git checkout rpi-4.1.y
```

Sinon, vous pouvez télécharger une archive compressée contenant uniquement la version 4.1 de ce noyau :

```
$ wget https://github.com/raspberrypi/linux/archive/rpi-4.1.y.zip
$ unzip -q rpi-4.1.y.zip
$ cd linux-rpi-4.1.y
```

Cette dernière solution présente l'avantage d'être beaucoup plus économe en espace de mémoire flash utilisée (comptez quand même 1 à 2 Go d'espace nécessaire pour réaliser la compilation).

Quelques packages nous seront indispensables :

```
$ sudo apt-get install -y libncurses5-dev bc
```

Préparons une configuration par défaut pour Raspberry Pi 2 :

```
$ make bcm2709_defconfig
```

Que nous affinons ensuite :

```
$ make menuconfig
```

```
  General Setup --->
    (-custom) Local version - append to kernel release
  Kernel hacking --->
    [*] Tracers --->
      -*- Kernel Function Tracer
      [*]   Kernel Function Graph Tracer
      [*] Interrupts-off Latency Tracer
      [*] Scheduling Latency Tracer
      [*] Trace syscalls
      -*- Create a snapshot trace buffer
```

-*- Allow snapshot to swap per CPU

Branch Profiling (No branch profiling) --->

[*] Trace max stack

[*] Support for tracing block IO actions

[ ] Enable kprobes-based dynamic events

[ ] Enable uprobes-based dynamic events

[*] enable/disable function tracing dynamically

[*] Kernel function profiler

Avant de lancer la compilation du noyau, il est important de désactiver le *swap* mémoire, afin d'éviter que des lectures / écritures incessantes usent prématurément votre carte SD. Pour cela il suffit de faire :

```
$ sudo swapoff --all
```

Personnellement (mais c'est une question de choix individuel), je préfère même désactiver définitivement le *swap* de mes Raspberry Pi avec :

```
$ sudo apt-get remove dphys-swapfile
```

La compilation du système prend une à deux heures en fonction des autres options sélectionnées.

```
$ make -j 4
```

Nous installons ensuite les modules et le noyau :

```
$ sudo make modules_install
```

```
$ sudo cp arch/arm/boot/zImage /boot/kernel-custom.img
```

Pour choisir de redémarrer sur ce nouveau noyau, il suffit de modifier (ou d'ajouter) la ligne **`kernel`** du fichier **`/boot/config.txt`** :

```
$ sudo nano /boot/config.txt
```

```
kernel=kernel-custom.img
```

```
$ sudo reboot
```

Pour aller plus loin…

[Axboe 2007] « *Blktrace User Guide* » Jens Axboe
http://www.cse.unsw.edu.au/~aaronc/iosched/doc/blktrace.html

[Blaess 2015] « *Création d'un système complet avec Buildroot 20115.11* »
http://www.blaess.fr/christophe/2015/12/08/

[Ficheux 2011] « *Introduction à Ftrace* » Pierre Ficheux
http://www.linuxembedded.fr/2011/03/introduction-a-ftrace

[Kobayashi 2012] « *Ineffective and effective way to find out latency bottlenecks by Ftrace* » Yoshitake Kobayashi : **https://events.linuxfoundation.org/images/stories/pdf/lf_elc12_kobayashi.pdf**

[Rostedt 2009] « *Debugging the kernel with Ftrace* » **http://lwn.net/Articles/365835/** et **http://lwn.net/Articles/366796/**

[LXR] « *The Linux Cross Reference* » **https://lxr.missinglinkelectronics.com/#linux/** (ou **http://lxr.free-electrons.com**)

Christophe BLAESS
christophe.blaess@logilin.fr
www.blaess.fr/christophe/