POLYTECHNIQUE
MONTRÉAL

INF2610 - Noyau d'un système d'exploitation
Ajout de points de trace pour les événements de type trappe noyau

Francis Deslauriers - 1540541 - Section 1
Sous la supervision de Raphaël Beamonte

Session hiver 2013
Remis le 19 avril 2013

Chapitre 1

Introduction

L'utilisation de systèmes informatiques dans des contextes temps réels et critiques demande des méthodes de diagnostics efficaces et non-intrusives. Les traceurs de systèmes d'exploitation permettent d'enregistrer les faits et gestes de ces systèmes.Les traceurs modernes s'appuient sur les point de trace présents dans le code du système d'exploitation.

Le noyau Linux contient des points de trace dans les fonctions liées aux événements et opérations critiques pour la plupart des architectures de processeurs. Cependant, la famille d'événements des trappes noyaux ne contient pas de points de trace. Or, ce genre de point de trace peut s'avérer très utile pour diagnostiquer certains problèmes, particulièrement dans un contexte de système temps-réel. Il est clair qu'une faute de page au mauvais moment peut s'avérer très coûteuse en terme de nombre de cycles de processeur et en terme d'échéances de contraintes.

Dans le cadre du cours Noyau de système d'exploitation, j'eus la chance d'effectuer un projet personnel sur l'ajout des points de trace pour les trappes noyaux du système d'exploitation Linux. De plus, afin de tester mon travail, j'ai également développé une sonde pour la suite de traçage LTTng 2.X [1].

Ce rapport exposera d'abord l'analyse préliminaire que j'ai effectuée dans le but de comprendre qu'est-ce qu'un point de trace, quels sont les points de trace déjà présents dans le noyau Linux et également bien comprendre qu'est-ce qu'une trappe noyau. Par la suite, il sera question de la façon dont j'ai défini les points de trace. Je poursuivrai avec la conception de la sonde qui permettra au traceur LTTng d'enregistrer les événements de type trappe. Je terminerai avec l'instrumentation des points de trace de trappes noyaux pour l'architecture x86.

1. `http://www.lttng.org`

Chapitre 2

Développement

2.1 Analyse préliminaire

Dans cette section, j'exposerai la démarche que j'ai suivie afin de me familiariser avec la définition et l'utilisation des points de trace (PT) noyaux. J'aborderai également mon étude des PT qui sont présents dans le noyau officiel et ceux qui étaient supportés par les patchs noyaux compatibles avec LTTng 0.X. Cette étude m'a permis de bien cerner quelle information il était pertinent de sauvegarder lorsqu'une trappe noyau apparaît.

2.1.1 Point de trace

Les définitions de points de trace dans le noyau se trouvent dans le répertoire *include/trace/events/* du code source du noyau. Une définition de PT définie quel prototype de fonction il faudra appeler pour enregistrer un événement. Le tableau 2.1 présente les différentes macros à utiliser lors de la définition d'un point de trace et explique leurs fonctions.

Table 2.1: Macros de préprocesseur à utiliser lors de la définition d'un point de trace

Macro	**Fonctionnalité**
TRACE_EVENT	Fonction de déclaration de point de trace. Prend en argument les macros qui suivent
TP_PROTO	Décrit le prototype du point de trace
TP_ARGS	Décrit les noms des variables à utiliser à l'intérieur du point de trace
TP_STRUCT__entry	Décrit la structure contenant l'information à stocker pour ce point de trace
TP_fast_assign	Effectue l'assignation des données passées en paramètre au champ de la structure
TP_printk	Décrit le format d'affichage des informations pertinentes pour ce point de trace

2.1.2 Trappe noyau

Une trappe noyau est un type d'interruption de processeur. Contrairement à une interruption matérielle, qui peut survenir à tout instant, une trappe arrive à l'exécution d'une opération. On parle souvent d'interruption synchrone. Un bon exemple de trappe est un point de débogage(*Breakpoint*) : lorsque le processeur décode l'opération INT3, une interruption trappe est levée et le système d'exploitation la gère à son gré.

2.1.3 Travaux antérieurs

Afin de produire une *patch* de qualité, j'ai premièrement analysé ce qui a déjà été fait sur le sujet par le passé. En effet, j'ai recherché sur la *Linux Kernel Mailing list* (LKML) des fils de discussion concernant l'ajout de PT pour les trappes noyaux et pour les fautes de page. Deux discussions pertinentes sont présentes sur la LKML. L'une d'elles concerne uniquement les fautes de page [1] tandis que l'autre concerne toutes les trappes noyaux [2]. Une lecture attentive de ces discussions m'a permis de bien cerner quels sont les critères pour que ce genre de *patch* soit accepté dans le noyau. J'ai donc pris en considération ces critères durant la suite du projet. Voici les points importants mentionnés lors de ces discussions :

- Deux points de trace, un pour l'entrée et un pour la sortie permettent d'informer sur la durée de gestion de l'exception.
- Le pointeur d'instruction permet de connaître l'instruction qui a causé la trappe.
- Pour l'événement de faute de page, il est pertinent de stocker l'adresse concernée par la faute.

2.2 Définition du point de trace

Bien que les fautes de page soient considérées comme des trappes noyau, j'ai décidé de développer deux type de PT différents. En effet, j'ai conçu un PT pour les fautes de page et un autre pour les trappes noyaux en général. Cette décision a été prise parce que l'information pertinente à être sauvegardée est plus abondante pour des fautes de page que pour les autres trappes noyaux. Par exemple, il est pertinent de sauvegarder l'adresse visée par cet accès mémoire. Dans cette section, j'exposerai mon implémentation de ces deux PT.

2.2.1 Faute de page

Le listing 2.1 représente l'implémentation du PT pour l'événement faute de page. Une faute de page survient lorsqu'un processus accède à une adresse mémoire comprise dans son espace d'adressage virtuelle, mais qui n'est pas chargée en mémoire principale. Pour cette raison, il est pertinent de sauvegarder l'adresse concernée par cette faute. De plus, l'argument *write_access* permet d'identifier s'il s'agit d'un accès en lecture ou en écriture. Finalement, grâce à l'argument *regs*, il est possible d'afficher la valeur du *ip* qui représente l'adresse pointée par le pointeur d'instruction.

1. `https://lkml.org/lkml/2010/11/10/89`
2. `https://lkml.org/lkml/2011/4/22/326`

Listing 2.1: include/trace/events/fault.h

```
TRACE_EVENT(page_fault_entry,
  TP_PROTO(struct pt_regs *regs, unsigned long address, int write_access
      ),
  TP_ARGS(regs, address, write_access),
  TP_STRUCT__entry(
    __field(  unsigned long,    ip    )
    __field(  unsigned long,    addr  )
    __field(  int,         write )
  ),
  TP_fast_assign(
    __entry->ip    = regs ? instruction_pointer(regs) : 0UL;
    __entry->addr = address;
    __entry->write  = write_access;
  ),
  TP_printk("ip=%lu addr=%lu write_access=%d", __entry->ip,
          __entry->addr, __entry->write)
);
TRACE_EVENT(page_fault_exit,
  TP_PROTO(int result),
  TP_ARGS(result),
  TP_STRUCT__entry(
    __field(  int,  res )
  ),
  TP_fast_assign(
    __entry->res  = result;
  ),
  TP_printk("result=%d", __entry->res)
);
```

2.2.2 Trappe

Dans le but de connaître le temps de gestion de la trappe par le système d'exploitation, il a été décidé de créer deux événements, un pour l'entrée et un pour la sortie de la trappe. On peut voir sur le listing 2.2 que le PT de l'événement trappe sauvegarde le numéro de trappe [3] ainsi que l'adresse du pointeur de l'instruction ayant provoqué la trappe.

Listing 2.2: include/trace/events/trap.h

```
TRACE_EVENT(trap_entry,
    TP_PROTO(struct pt_regs *regs, long trap),
    TP_ARGS(regs, trap),
    TP_STRUCT__entry(
        __field(    long,           trap   )
        __field(    unsigned long,  ip     )
    ),
    TP_fast_assign(
        __entry->trap = trap;
        __entry->ip    = regs ? instruction_pointer(regs) : 0UL;
    ),
    TP_printk("number=%ld ip=%lu", __entry->trap, __entry->ip)
);

TRACE_EVENT(trap_exit,
    TP_PROTO(long trap),
    TP_ARGS(trap),
    TP_STRUCT__entry(
        __field(  long,   trap     )
    ),
    TP_fast_assign(
        __entry->trap    = trap;
    ),
    TP_printk("number=%ld", __entry->trap)
);
```

3. L'énumération des différents types de trappe utilisée pour l'architecture x86 et amd64 est exposée en annexe A.1

2.3 Développement du sonde LTTng

L'ajout du support pour un nouveau PT a été étonnamment simple. En effet, le développement d'une sonde LTTng a été grandement simplifié par la présence de nombreux exemples dans *lttng-modules*. Le développement de la sonde LTTng m'a, par contre, demandé de me familiariser avec le concept de module noyau. Un module noyau est un programme qui est exécuté en mode noyau et qui peut être chargé à la volée selon les besoins [4]. La suite LTTng possède un module noyau pour chacun des PT présents dans le noyau.

Les *commits* exposant les modifications que j'ai dû faire pour ajouter les sondes *lttng-probe-trap* et *lttng-probe-fault* sont disponibles sur mon entrepôt GIT [5].
On peut voir dans ces *commits* que pour créer une nouvelle sonde *lttng-modules* il est nécessaire de créer 3 fichiers et de modifier le *Makefile*.

Une fois ces modifications et la compilation effectuées, je n'eus qu'à charger ces modules dans l'arborescence du noyau contenant les PT ajoutés. Par exemple, voici la commande à lancer pour installer la sonde du PT de fautes de page dans le noyau :

```
# sudo insmod lttng-probe-fault.ko
```

2.4 Localisation des points de trace

La localisation des PT dans le noyau a été la tâche pour laquelle j'ai eu le plus de difficultés. Cela est dû au fait que je n'ai qu'une connaissance très limitée du code source du noyau Linux. J'ai dû passer beaucoup de temps comprendre le code et les nombreuses macros utilisées dans ces sections. Mes lectures préliminaires des *patchs* déjà proposées sur la LKML et le code de LTTng 0.x m'ont été très utiles à cette étape.

2.4.1 Faute de page

Après discussion avec les membres de l'équipe de LTTng, j'ai décidé de tenter de reproduire l'instrumentation des PT de LTTng 0.X [6]. pour les PT de faute de page. Deux fichiers sont concernés par cette instrumentation. Dans un premier lieu, le fichier arch/x86/mm/fault.c

Listing 2.3: Localisation des points de trace dans le fichier arch/x86/mm/fault.c

```
//ligne 754
  if (likely(show_unhandled_signals))
    show_signal_msg(regs, error_code, address, tsk);
+ trace_page_fault_entry(regs, address, error_code & PF_WRITE);

  tsk->thread.cr2         = address;
  tsk->thread.error_code  = error_code;
  tsk->thread.trap_nr     = X86_TRAP_PF;
```

4. The Linux Kernel Module Programming Guide - http://linux.die.net/lkmpg/x40.html

5. Trappe :http://goo.gl/lHien, faute de page :http://goo.gl/LyJM0

6. Cette instrumentation se trouvait à ces endroit : http://goo.gl/M2Igv, http://goo.gl/IGyEh et http://goo.gl/wCpdj

```
  force_sig_info_fault(SIGSEGV, si_code, address, tsk, 0);
+   trace_page_fault_exit(-1);

//ligne 1185
+       trace_page_fault_entry(regs, address, write);
        fault = handle_mm_fault(mm, vma, address, flags);
+       trace_page_fault_exit(fault);
```

Listing 2.4: Localisation des points de trace dans le fichier mm/memory.c

```
//ligne 1828
+    trace_page_fault_entry(0, start,
+                    foll_flags & FOLL_WRITE);
     ret = handle_mm_fault(mm, vma, start,
                           fault_flags);
+    trace_page_fault_exit(ret);
```

2.4.2 Trappe

Dans l'architecture x86, la gestion des trappes noyaux est faite dans le fichier *arch/x86/kernel/traps.c*. La fonction principale pour la gestion des événements trappes est nommée *do_trap*. Cette fonction est appelée la plupart du temps lorsqu'une trappe survient.

Certaines trappes provoquent l'appel de fonctions spécifiques à un type de trappe. Ces différences entre la gestion des trappes noyaux en fonction de leurs types a eu pour effet qu'il était difficile de trouver un endroit commun dans le code source pour ajouter le PT. J'ai donc dû ajouter des PT à plusieurs endroits dans le noyau.

Plusieurs trappes sont gérées par deux macros très similaires connues sous les nom *DO_ERROR*[7] et *DO_ERROR_INFO*[8]. Ces deux macros appellent ensuite la fonction *do_trap*, il a donc été simple d'ajouter un PT dans ces macros. Le listing 2.5 présente ces deux ajouts.

Listing 2.5: Localisation des points de trace dans les macros DO_ERROR et DO_ERROR_INFO

```
//DO_ERROR ligne 185
  conditional_sti(regs);                \
+ trace_trap_entry(regs, trapnr);       \
  do_trap(trapnr, signr, str, regs, error_code, NULL);    \
+ trace_trap_exit(trapnr);              \
  exception_exit(regs);                \
 }
//DO_ERROR_INFO ligne 204
  conditional_sti(regs);                \
+ trace_trap_entry(regs, trapnr);       \
  do_trap(trapnr, signr, str, regs, error_code, &info);   \
+ trace_trap_exit(trapnr);              \
  exception_exit(regs);                \
 }
```

Certaines trappes noyaux, comme X86_TRAP_NM, ne sont pas gérées par la fonction *do_trap* mais par une fonction spécifique pour ce type de trappe. Dans cette situation, la fonction *do_device_not_available*[9] effectue la gestion. J'ai ajouté en annexe A.2 l'instrumentation de cet événement. L'instrumentation de

7. Déclarée ici : `http://goo.gl/OXgHn`
8. Déclarée ici : `http://goo.gl/lWQYo`
9. La fonction en question : `http://goo.gl/wAULX`

la plupart des autres événements n'utilisant pas une macro comme *DO_ERROR* est faite de façon très similaire à l'événement X86_TRAP_NM.

Il y a cependant certains événements qui ne surviennent que dans des situations exceptionnelles que j'ai décidé de ne pas instrumenter. Par exemple, la trappe *machine_check*, X86_TRAP_MC, signifie un problème matériel dans le processeur et nécessite un redémarrage du système [10]. Comme ce genre d'exception sont très difficiles à reproduire et donc à tester. De plus, l'entête de la fonction *do_machine_check* contient la phrase :

> `This is executed in NMI context not subject to normal locking rules. This implies that most kernel services cannot be safely used. Don't even think about putting a printk in there!` [11]

ce qui démontre l'aspect critique d'une situation comme celle là. Pour ces deux raisons, j'ai décidé de ne pas inclure de PT dans cette fonction pour le moment.

10. `http://en.wikipedia.org/wiki/Machine_Check_Exception`
11. `http://lxr.linux.no/linux+v3.8.8/arch/x86/kernel/cpu/mcheck/mce.c#L997`

Chapitre 3

Conclusion

Ce projet a été très enrichissant pour moi. J'ai acquis beaucoup de connaissances tout en consolidant certaines déjà acquises. Notamment, j'ai appris en détail comment configurer, compiler et installer un noyau sur ma machine. J'en ai également appris sur le fonctionnement interne des processus et des interruptions.

J'ai rencontré plusieurs problèmes au cours de ce projet. D'abord, j'ai eu beaucoup de difficulté à tester la localisation des PT dans le noyau. Principalement pour le PT pour les trappes noyaux qui peuvent être très difficiles à produire artificiellement. En effet, des trappes comme *Alignment Check* et *Spurious Interrupt* se produisent dans des situations très extrêmes. D'un autre côté, la trappe Debug peut être déclenchée avec une simple commande assembleur x86 [1]. Évidemment, les fautes de page n'ont pas à être générées parce qu'elles ont lieu normalement plusieurs fois par seconde.

Au début du projet, j'avais comme objectif de soumettre une *patch* concernant tous les PT de trappe. Après discussion avec des membres de l'équipe LTTng, j'ai compris qu'il est plus facile d'intégrer une *patch* dans le noyau lorsque celle-ci concerne qu'un nombre limité de modifications. Pour cette raison, j'ai décidé de soumettre une *patch* pour les PT de fautes de page uniquement pour ensuite continuer avec une *patch* pour chacune des autres trappes noyaux.

À l'heure actuelle, ce projet n'est pas terminé. J'ai reçu au cours des derniers jours le soutien de l'équipe LTTng pour ma définition et l'instrumentation du PT des fautes de page. Je vais donc pouvoir soumettre cette *patch* pour révision sur la LKML très bientôt. Je poursuivrai avec la définition du PT du type d'événement trappe ainsi que l'instrumentation des trappes que j'ai pu tester.

1. La ligne suivante ajout un point d'arrêt dans le code : *asm("INT3")*;

Annexe A

Extraits de code

Listing A.1: Énumération des trappes de l'architecture x86/amd64

```
/* arch/x86/include/asm/traps.h */
/* Interrupts/Exceptions */
 enum {
        X86_TRAP_DE = 0,        /*  0, Divide-by-zero */
        X86_TRAP_DB,            /*  1, Debug */
        X86_TRAP_NMI,           /*  2, Non-maskable Interrupt */
        X86_TRAP_BP,            /*  3, Breakpoint */
        X86_TRAP_OF,            /*  4, Overflow */
        X86_TRAP_BR,            /*  5, Bound Range Exceeded */
        X86_TRAP_UD,            /*  6, Invalid Opcode */
        X86_TRAP_NM,            /*  7, Device Not Available */
        X86_TRAP_DF,            /*  8, Double Fault */
        X86_TRAP_OLD_MF,        /*  9, Coprocessor Segment Overrun */
        X86_TRAP_TS,            /* 10, Invalid TSS */
        X86_TRAP_NP,            /* 11, Segment Not Present */
        X86_TRAP_SS,            /* 12, Stack Segment Fault */
        X86_TRAP_GP,            /* 13, General Protection Fault */
        X86_TRAP_PF,            /* 14, Page Fault */
        X86_TRAP_SPURIOUS,      /* 15, Spurious Interrupt */
        X86_TRAP_MF,            /* 16, x87 Floating-Point Exception */
        X86_TRAP_AC,            /* 17, Alignment Check */
        X86_TRAP_MC,            /* 18, Machine Check */
        X86_TRAP_XF,            /* 19, SIMD Floating-Point Exception */
        X86_TRAP_IRET = 32,     /* 32, IRET Exception */
 }
```

Listing A.2: Instrumentation *device not available*

```
@@ -646,6 +664,7 @@ dotraplinkage void __kprobes
  do_device_not_available(struct pt_regs *regs, long error_code)
  {
   exception_enter(regs);
+ trace_trap_entry(regs, X86_TRAP_NM);
   BUG_ON(use_eager_fpu());

  #ifdef CONFIG_MATH_EMULATION
@@ -664,6 +683,7 @@ do_device_not_available(struct pt_regs *regs, long
      error_code)
  #ifdef CONFIG_X86_32
   conditional_sti(regs);
  #endif
+ trace_trap_exit(X86_TRAP_NM);
   exception_exit(regs);
  }
```