Débogage du noyau Linux

Stelian Pop <stelian.pop@fr.alcove.com>

Retour

Fermer

Problématique

Le noyau Linux est un logiciel très stable.

Cependant, on trouve/crée des bugs lors de :
- ✓ développement noyau ;
- ✓ développement drivers ;
- ✓ utilisation fonctionnalitées *alpha* ;
- ✓ etc...

Mais :
- ✓ débogage classique impossible (débogueur, analyseur mémoire etc) ;
- ✓ arrêt noyau (plantage de la machine) ;
- ✓ matériel difficilement simulable (IT, accès mémoire, etc) ;
- ✓ corruption mémoire physique.

Solutions

Beaucoup de solutions permettent de déboguer le noyau Linux.

La difficulté consiste à choisir la bonne.

Plusieurs catégories :
- ✓ débogage proprement dit ;
- ✓ analyse post-mortem ;
- ✓ outils de trace ;
- ✓ outils de profiling ;

➴ outils spécifiques ix86 (mais pas seulement...)




Printk

La plus simple méthode de débogage : `printk`.

Affiche un message (trace) sur la console.

- très facile à mettre en oeuvre.
- peut influencer les timings.
- modification des sources.

Proc, ioctl, etc.

Méthodes de débogage d'un sous-système déjà en place.

Activation par `ioctl` ou fichier dans `/proc`.

Logs dans `/proc` ou `syslog`.

- adaptées au besoin.
- faut qu'elles soient disponibles.

Ksymoops

Erreur dans le noyau → `oops`.

`oops` = pile d'appel, état des registres.

✓ noyaux ≤ 2.4, oops numérique qui doit être "décodé" grâce à `ksymoops` + `System.map` ;

✓ noyaux 2.5, information du `System.map` incluse dans le noyau (kallsyms : option lors de la compilation).

Indispensable pour un bug report !

- pas de préparation spéciale.
- (kallsyms) facilité d'utilisation.
- difficulté de copier l'oops (console série...).
- (kallsyms) taille du noyau.
- attention aux corruptions !

Ksymoops : exemple

Code :

```
static int oops_init(void) {
        *( (char *) 0) = 0;
        return 0;
}

module_init(oops_init);
```

Oops non-décodé :

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
 printing eip:
cc847060
*pde = 00000000
Oops: 0002
CPU:    0
EIP:    0010:[<cc847060>]    Not tainted
EFLAGS: 00010286
eax: cc847060   ebx: fffffffea   ecx: 00004a08   edx: c11c829c
esi: 00000000   edi: 00000000   ebp: cc847000   esp: ca69bf28
ds: 0018   es: 0018   ss: 0018
Process insmod (pid: 604, stackpage=ca69b000)
Stack: c01164c8 00000000 0806dfad 0000006d 00000060 00000060 00000004 cab50d40
       ca5e0000 ca411000 00000060 cc85a000 cc847060 0000024c 00000000 00000000
       00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
Call Trace:    [<c01164c8>] [<cc847060>] [<c010893b>]

Code: c7 05 00 00 00 00 00 00 00 00 31 c0 c3 6f 6f 70 73 00 5f 5f
```

Oops décodé :

```
...
>>EIP; cc847060 <[oops]init_module+0/0>   <=====

>>eax; cc847060 <[oops]init_module+0/0>
>>ebx; ffffffea <END_OF_CODE+337a1197/????>
>>ecx; 00004a08 Before first symbol
>>edx; c11c829c <_end+f39224/c570fe8>
>>ebp; cc847000 <[mousedev].bss.end+23c2d/23c8d>
>>esp; ca69bf28 <_end+a40ceb0/c570fe8>

Trace; c01164c8 <sys_init_module+47c/5b0>
Trace; cc847060 <[oops]init_module+0/0>
Trace; c010893b <system_call+33/38>

Code;  cc847060 <[oops]init_module+0/0>
00000000 <_EIP>:
Code;  cc847060 <[oops]init_module+0/0>   <=====
   0:   c7 05 00 00 00 00 00      movl   $0x0,0x0   <=====
Code;  cc847067 <[oops]oops_init+7/c>
   7:   00 00 00
Code;  cc84706a <[oops]oops_init+a/c>
   a:   31 c0                     xor    %eax,%eax
Code;  cc84706c <[oops].text.end+0/1ff4>
   c:   c3                        ret
Code;  cc84706d <[oops].text.end+1/1ff4>
   d:   6f                        outsl  %ds:(%esi),(%dx)
Code;  cc84706e <[oops].text.end+2/1ff4>
   e:   6f                        outsl  %ds:(%esi),(%dx)
Code;  cc84706f <[oops].text.end+3/1ff4>
   f:   70 73                     jo     84 <_EIP+0x84>
Code;  cc847071 <[oops].text.end+5/1ff4>
  11:   00 5f 5f                  add    %bl,0x5f(%edi)
```




Retour
Fermer

Magic SysRq

Séquences de touches ”système” exécutables à la console (option lors de la compilation).

Fonctionnent même en cas de plantage (”léger”).

Permettent de (voir fichier `Documentation/sysrq.txt`) :

- ✓ synchroniser/monter R/O les disques ;
- ✓ consulter la liste des tâches ;
- ✓ obtenir des informations sur la mémoire ;
- ✓ etc.

➶ standard et efficace.

➴ ne fonctionne pas si interruptions désactivées.

➴ limitation des actions.

NMI watchdog

Dans le cas de blocages ”hard”, il est impossible d’exécuter les SysRq.

Solution : le NMI (Non Maskable Interrupt) watchdog (voir fichier `Documentation/nmi_watchdog.txt`).

Détection de plantage processeur et affichage d’un oops.

➶ efficace dans les pires cas.

➴ il faut booter avec le paramètre `nmi_watchdog=x` (conflit possible avec outils de mesure des perfs).

Paramètres de compilation du noyau

Activables dans la section ”Kernel hacking” de la configuration du noyau.

Exemples :

- ✓ test dynamique du dépassement de pile ;
- ✓ vérification des allocations mémoires (malloc/free) et empoisonnement des zones libérées ;
- ✓ vérification des accès mémoire I/O sur des zones non réservées ;
- ✓ activation des séquences Magic SysRq ;
- ✓ débogage des `spinlock` ;
- ✓ débogage highmem ;
- ✓ intégration de ksymoops dans le noyau (2.5) ;
- ✓ compilation avec frame-pointers.

➶ efficacité selon choix.

➴ surcharge rendant nécessaire la désactivation en production et pouvant influencer les timings.

KDB

URL : http://oss.sgi.com/projects/kdb/ (ix86, ia64)

Débogueur fonctionnant sur la machine à déboguer (dans le handler du clavier).

Permet de :

- ✓ interrompre l'exécution du noyau ;
- ✓ mettre des breakpoints (exécution, lecture/écriture) ;
- ✓ exécuter le code en mode pas à pas ;
- ✓ connaître la pile d'appel (tâche courante, toutes les tâches) ;
- ✓ examiner/modifier la mémoire (adresses, listes chaînées etc) ;
- ✓ examiner/modifier les registres des processeurs.

➶ débogueur local à la machine (pas de matériel supplémentaire).

➴ débogueur au niveau assembleur.

KDB : exemple

```
Entering kdb (current=0xc02aa000, pid 0) due to Keyboard Entry
kdb> bp sys_open
Instruction(i) BP #0 at 0xc01300cc (sys_open)
    is enabled globally adjust 1
kdb> go
Instruction(i) breakpoint #0 at 0xc01300cc (adjusted)
0xc01300cc sys_open:         int3

Entering kdb (current=0xcacd6000, pid 416) due to Breakpoint @ 0xc01300cc
kdb> bt
0xcacd6000 00000416 00000001  1  000  run   0xcacd6270*sendmail
ESP        EIP        Function (args)
0xcacd7fc0 0xc01300cc sys_open (0x80c0544, 0x0, 0x1b6, 0x80f93a0, 0x80de260)
                              kernel .text 0xc0100000 0xc01300cc 0xc013014c
0xcacd7fc4 0xc01088eb system_call+0x33
                              kernel .text 0xc0100000 0xc01088b8 0xc01088f0
kdb> ps
Task Addr  Pid      Parent   [*] cpu  State Thread     Command
0xc121c000 00000001 00000000  1  000  stop  0xc121c270 init
[...]
0xcacd6000 00000416 00000001  1  000  run   0xcacd6270*sendmail
0xcac2a000 00000426 00000001  1  000  stop  0xcac2a270 sendmail
kdb> ss
SS trap at 0xc01300cd (sys_open+0x1)
0xc01300cd sys_open+0x1:     push   %ebx
kdb> ss
SS trap at 0xc01300ce (sys_open+0x2)
0xc01300ce sys_open+0x2:     pushl  0xc(%esp,1)
kdb> ss
SS trap at 0xc01300d2 (sys_open+0x6)
0xc01300d2 sys_open+0x6:     call   0xc0138da0 getname
kdb> bc 0
Breakpoint 0 at 0xc01300cc cleared
kdb> go
```

Retour
Fermer

KGDB

URL : http://kgdb.sourceforge.net/ (ix86, ia64, autres...)

Modus operandi :

- ✓ deux machines reliées par un câble série (null-modem) ;
- ✓ machine à déboguer (target) : noyau patché avec `kgdb` ;
- ✓ machine de développement : copie des sources + binaires ;
- ✓ machine de développement : utilisation `gdb` standard (+ `ddd` etc.).

Permet de :

- ✓ interrompre l'exécution du noyau ;
- ✓ mettre des breakpoints (exécution, lecture/écriture) ;
- ✓ exécuter le code en mode pas à pas ;
- ✓ connaître la pile d'appel (tâche courante, toutes les tâches) ;
- ✓ examiner/modifier la mémoire (adresses, variables, listes chaînées etc) ;
- ✓ ... toute fonctionnalité de gdb standard ...

➶ débogueur au niveau source.

➴ deux machines nécessaires.

KGDB : exemple

```
Program received signal SIGTRAP, Trace/breakpoint trap.
breakpoint () at gdbstub.c:1177
1177    }
(gdb) break sys_open
Breakpoint 2 at 0xc0131ef7: file open.c, line 783.
(gdb) cont
Continuing.

Breakpoint 2, sys_open (filename=0x80c0544 "/proc/loadavg", flags=0, mode=438)
    at open.c:783
783             tmp = getname(filename);
(gdb) list
778             int fd, error;
779
780     #if BITS_PER_LONG != 32
781             flags |= O_LARGEFILE;
782     #endif
783             tmp = getname(filename);
784             fd = PTR_ERR(tmp);
785             if (!IS_ERR(tmp)) {
786                     fd = get_unused_fd();
787                     if (fd >= 0) {
(gdb) next
...
786                     fd = get_unused_fd();
(gdb) next
787                     if (fd >= 0) {
(gdb) print fd
$1 = 5
(gdb) where
#0  sys_open (filename=0x80c0544 "/proc/loadavg", flags=0, mode=438)
    at open.c:787
#1  0xc0107323 in system_call () at af_packet.c:1891
(gdb) clear sys_open
Deleted breakpoint 2
(gdb) cont
Continuing.
```

IKD : Integrated Kernel Debugging

URL : ftp://ftp.kernel.org/pub/linux/kernel/people/andrea/ikd/

Collection d'outils réunis par Andrea Arcangeli :

- ✓ test dynamique du dépassement de pile ;
- ✓ outil de traçage noyau ;
- ✓ détecteur de fuites mémoire ;
- ✓ détecteur de deadlock par spinlock ;
- ✓ NMI watchdog ;
- ✓ débogueur kdb ;
- ✓ ...

➶ quelques outils intéressants.

➴ partiellement obsolète pour les nouveaux noyaux.

Retour

Fermer

UML : User Mode Linux

URL : http://user-mode-linux.sourceforge.net/

Virtualisation du noyau Linux.

Visible comme une architecture supportée par Linux : `um`.

Processus standard permettant l'utilisation "normale" de `gdb`, `gprof` etc.

- facilité de débogage.
- pas de débogage matériel.

UML : exemple

```
$ ./linux debug
GNU gdb Red Hat Linux (5.2.1-4)
...
(gdb) break sys_open
Breakpoint 4 at 0xa003d384: file open.c, line 808.
(gdb) cont
Continuing.

Breakpoint 4, sys_open (filename=0xa01cbcca "/dev/console", flags=2, mode=0)
    at open.c:808
808             tmp = getname(filename);
(gdb) list
803             int fd, error;
804
805     #if BITS_PER_LONG != 32
806             flags |= O_LARGEFILE;
807     #endif
808             tmp = getname(filename);
809             fd = PTR_ERR(tmp);
810             if (!IS_ERR(tmp)) {
811                     fd = get_unused_fd();
812                     if (fd >= 0) {
(gdb) next
...
811                     fd = get_unused_fd();
(gdb) next
812                     if (fd >= 0) {
(gdb) print fd
$2 = 0
(gdb) cont
Continuing.
```

LKCD : Linux Kernel Crash Dump

URL : http://lkcd.sourceforge.net

Permet de sauvegarder l'état du noyau en cas de crash.

Pilotable aussi par séquences SysRq.

Image mémoire sauvegardée en swap, puis lors du reboot automatiquement dans `/var/log/dumps`.

Analyse post-mortem grâce à l'outil `lcrash`.

- pas de surcharge, utilisable en production.
- sauvegarde de l'état, pas des événements.

LKCD : exemple

```
=======================
LCRASH CORE FILE REPORT
=======================
GENERATED ON:
    Wed Jan 22 16:44:03 2003

TIME OF CRASH:
    Wed Jan 22 16:42:01 2003

PANIC STRING:
    sysrq
...

====================
CURRENT SYSTEM TASKS
====================

      ADDR    UID    PID   PPID  STATE     FLAGS CPU  NAME
============================================================
0xc022e000      0      0      0      0         0   0  swapper
0xc121e000      0      1      0      1     0x100   0  init
...
0xca676000      0    545      1      1     0x100   0  mingetty
0xca55c000      0    548    540      1     0x100   0  bash

===========================
STACK TRACE OF FAILING TASK
===========================

============================================================
STACK TRACE FOR TASK: 0xc022e000 (swapper)

 0 default_idle+35 [0xc0106ccf]
 1 cpu_idle+64 [0xc0106d38]
 2 start_kernel+257 [0xc0230669]
 3 is386+74 [0xc010018c]
============================================================
```

Retour
Fermer

Netdump

URL : http://www.redhat.com... dans kernel.src.rpm

En cas de crash, envoie l'image mémoire à un serveur de dumps par le réseau.

Dump directement utilisable par `gdb`.

`netdump` fait partie de l'infrastructure `netconsole` qui peut être utilisée pour logguer les messages noyau par le réseau (`oops`).

- ↗ moins de couches à traverser que LKCD, donc plus fiable.
- ↗ dumps centralisés sur un serveur de dumps (+ déclenchement d'événements).

- ↘ uniquement certaines cartes réseau supportées (tulip, eepro100).
- ↘ nécessité d'avoir un serveur de dumps.

Netdump : exemple

```
# ls /var/crash/
10.20.10.4-2003-01-22-17:31 magic scrips
# ls /var/crash/10.20.10.4-2003-01-22-17\:31
log vmcore
# cat /var/crash/10.20.10.4-2003-01-22-17\:31/log
Unable to handle kernel NULL pointer dereference at virtual address 00000000
 printing eip:
cc88c060
*pde = 00000000
Oops: 0002
oops netconsole eepro100 mousedev keybdev hid input usb-uhci usbcore ext3 jbd
CPU:    0
EIP:    0010:[<cc88c060>]    Tainted: GF
EFLAGS: 00010286

EIP is at oops_init [oops] 0x0 (2.4.18-19.8.0)
eax: cc88c060   ebx: fffffffea   ecx: c02e1800   edx: c02e1808
esi: 00000000   edi: 00000000   ebp: cc88c000   esp: c889bf28
ds: 0018   es: 0018   ss: 0018
Process insmod (pid: 775, stackpage=c889b000)
Stack: c011a022 00000000 08072945 0000006d 00000060 00000060 00000004 c8c177e0
       c7e9e000 c5ab2000 00000060 cc887000 cc88c060 0000024c 00000000 00000000
       00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
Call Trace: [<c011a022>] sys_init_module [kernel] 0x4a2 (0xc889bf28))
[<cc88c060>] oops_init [oops] 0x0 (0xc889bf58))
[<c0108d23>] system_call [kernel] 0x33 (0xc889bfc0))

Code: c7 05 00 00 00 00 00 00 00 00 31 c0 c3 6f 6f 70 73 00 5f 5f

...

# gdb /usr/src/linux/linux /var/crash/10.20.10.4-2003-01-22-17\:31/vmcore
...
```

Retour
Fermer

Dprobes : Dynamic Probes

URL : http://www-124.ibm.com/linux/projects/dprobes

Infrastructure permettant de modifier dynamiquement l'exécution du noyau (ou applications).

Permet l'activation de points d'arrêt dynamiques à une adresse mémoire, avec insertion de code avant/après.

Les handlers sont définis dans un pseudo-langage (ou C), et sont activables avec un outil fourni.

Possibilité de coupler **dprobes** avec **LTT**.

- ➶ pas de modification du code source.
- ➶ peu d'influence sur la performance du système.
- ➴ gros patch monolithique.

Kprobes : Kernel Dynamic Probes

URL : http://www-124.ibm.com/linux/projects/kprobes

Version simplifiée des **dprobes**.

Permet l'activation de points d'arrêt dynamiques à une adresse mémoire, avec insertion de code avant/après.

Les handlers sont définis en C, compilables en tant que module noyau (interface mode-utilisateur prévue).

- pas de modification du code source.
- modulable, permet d'obtenir tout type de débogage.
- peu d'influence sur la performance du système.

- nécessite d'écrire du code noyau.
- jeune, noyau 2.5.




LTT : Linux Trace Toolkit

URL : http://www.opersys.com/LTT/index.html

Infrastructure permettant de suivre l'exécution du noyau :

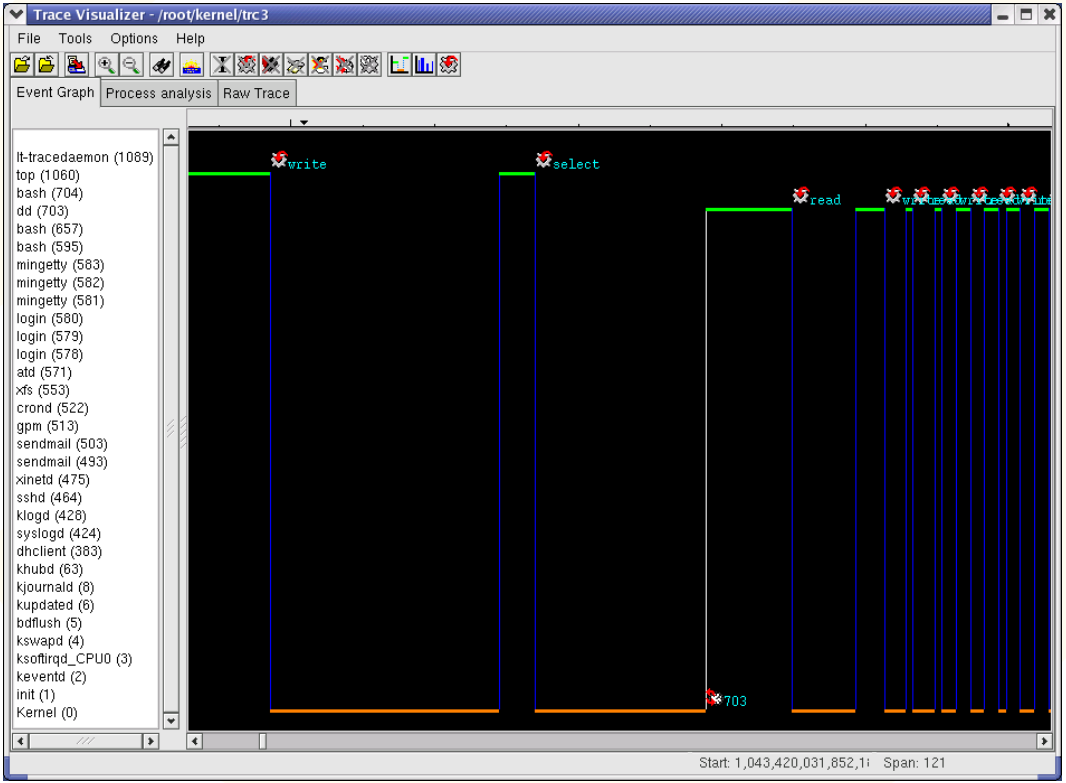
- ✓ activation de **LTT** : module noyau `tracer` ou built-in ;
- ✓ sélection des événements à tracer (syscall, timer, schedule etc) ;
- ✓ lancement du `tracedaemon` ;
- ✓ lecture graphique avec `tracevisualizer`.

Utilisable aussi avec **RTAI**.

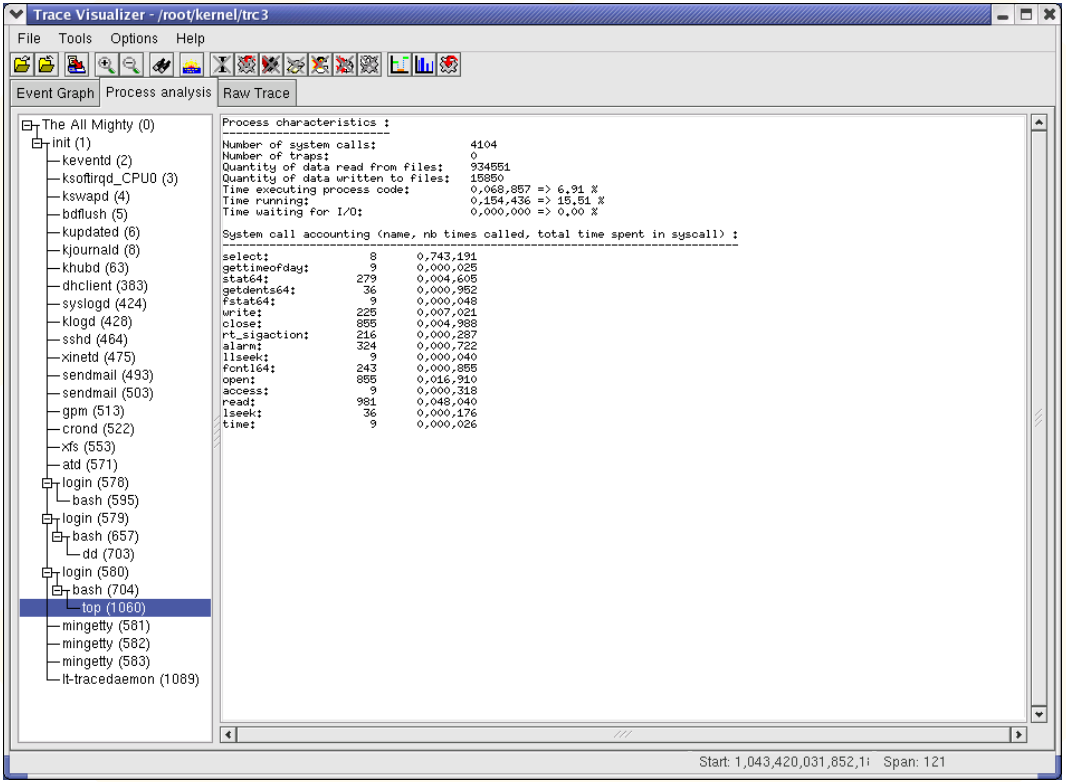
- ⬈ solution solide et prouvée.
- ⬈ interface graphique aidant l'interprétation.

- ⬊ 48 points de trace fixes (mais intégration possible avec **dprobes**).
- ⬊ architecture "non-optimale" (intégration noyau).

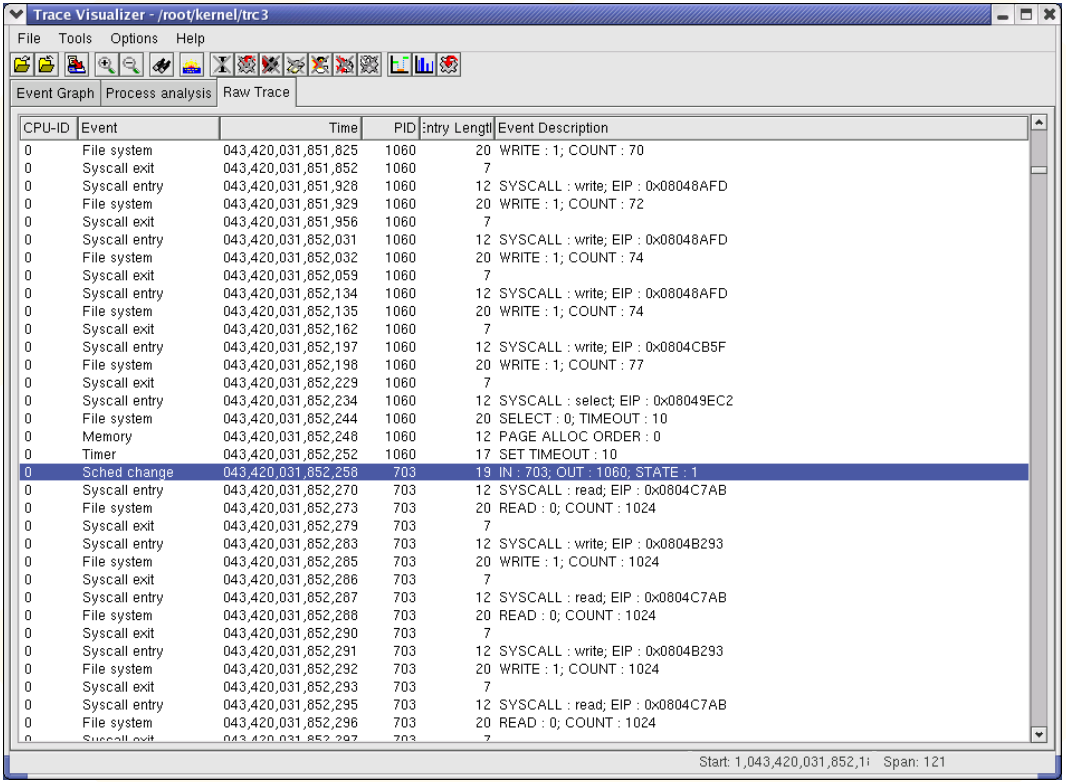
LTT : exemple



Retour
Fermer



Retour
Fermer



Retour
Fermer

Profile

Profiler inclus dans le noyau : booter avec `profile=2`.

Informations de profiling écrites dans `/proc/profile`.

Récupération avec l'utilitaire `readprofile`.

Informations :
- ✓ *PC sampling* : nombre de samples par fonction ;
- ✓ *normalized load* : ratio samples / longueur de la fonction ;

Le sampling est effectué à chaque tick du timer.

➹ standard, inclus dans tous les noyaux.

➴ nécessite un reboot.
➴ ne fonctionne pas quand les interruption sont désactivées.

Kernprof

URL : http://oss.sgi.com/projects/kernprof/

Architectures supportées : i386, ia64, sparc64, et mips64.

Réécriture du mécanisme standard, extension aux nouveaux modes :

- ✓ *PC sampling* ;
- ✓ *Call graph* ;
- ✓ *Annotated call graph* ;
- ✓ *Scheduler call graph* ;
- ✓ *Call count* ;
- ✓ *Call backtracing*.

Sampling effectué :

- ✓ *time domain* : tick du timer ;
- ✓ *PMC domain* : compteurs de performance processeur ;

↗ plusieurs modes, avec des surcoûts différents.

↗ plusieurs architectures supportées.

↘ nécessite options de compilation du noyau (`mcount`, `frame-pointers`) diminuant les performances.

Oprofile

URL : http://oprofile.sourceforge.net/

Permet de profiler tout :

- ✓ noyau ;
- ✓ modules noyau ;
- ✓ librairies partagées ;
- ✓ applications.

Sampling effectué par :

- ✓ compteurs de performance inclus dans les processeurs
- ✓ le RTC.

- ➶ très complet (et complexe).
- ➶ activable ’à la volée’.
- ➶ inclus dans le noyau 2.5 (et noyaux Red Hat $\geq$ 8.0).
- ➶ surcoût de 3-8% uniquement.

- ➴ x86 uniquement.
- ➴ jeune...

Conclusion

☛ Points forts :

✓ beaucoup de solutions adaptés à différents besoins ;
✓ possibilité d'implémenter sa propre solution ;
✓ aide de la communauté ;
✓ ... peu de bugs.

☛ Points faibles :

✓ difficulté de choisir la solution adaptée ;
✓ solutions souvent en état *alpha* / *beta* ;
✓ pas de solution officielle ;
✓ peu de documentation.




Retour
Fermer

Liens

- Alcôve
  http://www.alcove.com

- Stelian Pop
  http://popies.net
  <stelian.pop@fr.alcove.com>
  <stelian@popies.net>