2005 年度上期

オープンソースソフトウェア活用基盤整備事業

「OSS 性能・信頼性評価 / 障害解析ツール開発」

OS 層

～Alicia 編～

作成

OSS 技術開発・評価コンソーシアム

商標表記

- Alicia は、ユニアデックス株式会社の登録商標です。
- Asianux は、ミラクル・リナックス株式会社の日本における登録商標です。
- Intel、Itanium および Intel　Xeon は、アメリカ合衆国およびその他の国におけるインテルコーポレーションまたはその子会社の商標または登録商標です。
- Intel は、Intel Corporation の会社名です。
- Linux は、Linus Torvalds の米国およびその他の国における登録商標あるいは商標です。
- MIRACLE LINUX は、ミラクル・リナックス株式会社が使用許諾を受けている登録商標です。
- Pentium は、Intel Corporation のアメリカ合衆国及びその他の国における登録商標です。
- Red Hat は、米国およびその他の国で Red Hat, Inc. の登録商標若しくは商標です。
- Solaris は、米国 Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標です。
- SUSE は、米国 Novell, Inc.の一部門である SUSE LINUX AG.の登録商標です。
- Turbolinux は、ターボリナックス株式会社の商標または登録商標です。
- UNIX は、X/Open Company Limited が独占的にライセンスしている米国ならびに他の国における登録商標です。
- Windows は、米国およびその他の国における米国 Microsoft Corp.の登録商標です。
- その他記載の会社名、製品名は、それぞれの会社の商号、商標もしくは登録商標です。

目次

1 はじめに

1.1 背景

昨年度に引き続き、Linux のカーネルクラッシュダンプに関する話題がホットである。Linux World2005 でも、各ブースで、障害解析の話題では必ずダンプについて触れられていたように思う。また、ダンプ採取の実装に関しても、NTT データ・VA Linux Systems Japan による mkdump をはじめとして、富士通の diskdump、日立の Linux Tough Dump などが次々と発表されている。Linux カーネルのアップストリームでは、IBM インドの技術者による kdump が取り込まれるようである。それぞれの実装方式に関しては、参考文献の URL を参照されたい。

このように、ダンプ採取に関する話題は豊富であるが、ダンプ解析についての話題はまだまだ少ない。2005 年 9 月 16 日に、日本初の Linux Kernel Dump Summit が開催されるなど、今後、ダンプ解析についての議論が活発化していくことを実感している。

1.2 ダンプ解析とは

Linux カーネルのクラッシュダンプとは、Linux カーネルがある瞬間のメモリの内容をあるデバイスに吐き出したしたものである。カーネルは自身の不具合や矛盾の理由を見つけることができない場合、ハングやパニックといった状態に陥る。その原因を人間の目で診断するためにダンプが必要となる。

身近な例で表現してみよう。人間は急性な病気になった場合や体の調子が悪いと感じた場合、または家族や友人から健康状態について指摘された場合に病院に行き、病状によっては、レントゲンや CT を撮る場合がある。このレントゲンや CT スキャンを行うことが、ダンプ採取のイメージに近い。

その後、レントゲンの撮影データ(ネガ)を利用して専門の医師が分析を行い、どんな病気でどんな治療が必要かを診断する。クラッシュダンプにおいては、この撮影データの分析作業がダンプ解析に相当する。

2 Alicia の開発経緯

人間の病気の分野では、治療方法について様々な研究がなされており、ノウハウはとても充実している。しかし、Linux の障害について、特にダンプ解析の分野では、ノウハウが非常に不足している。ダンプ解析に関する議論がやっと始まったばかりである。

そこで、まずは、人間によるダンプ解析の補助をするツールの整備・統合と、ノウハウを蓄積するための仕組みを提供することから始め、徐々にノウハウを蓄積していくという方針を立てた。その方針を実現するためのツールが、Alicia(Advanced LInux Crash-dump Interactive Analyzer)である。

Alicia URL： http://alicia.sourceforge.net/

2.1 ダンプ解析ツールの現状

Linux カーネルのクラッシュダンプ解析ツールは、主に crash と lcrash の 2 つが現場で利用されている。それらに加え、Alicia が 2005 年 3 月にリリースされているという状況である。

2.1.1 crash

様々なダンプフォーマットに対応し、 GDB をラッピングしていることが特徴である。Red Hat の社員が新しいカーネルへの対応を行い頻繁にリリースしていることにより、とても鮮度が高い。また、拡張コマンドを作成することが可能であり、crash 本体の変更なく機能拡張が行える。

2.1.2 lcrash

lcrash は Linux のクラッシュダンプを解析するために、全ての設計・実装が一から作られている。他のツールに依存することがないため、他のアーキテクチャ(例えば IA64 等)で採取されたダンプを見ることもできる。また、sial(C 言語に良く似た言語仕様を持つ)マクロで、コマンドを簡単に拡張できる。

2.1.3 Alicia

Alicia は 2005 年 3 月にバージョン 1.0 としてリリースされたダンプ解析環境の統合化を実現するツールである。上記ダンプ解析ツール使用者は、簡単に Alicia の環境に移行することができ、さらに Alicia の機能拡張部分の恩恵を受けることができる。

特徴は、Perl 言語でダンプ解析スクリプト(LDAS)が手軽に記述できることであり、それが解析手順のスクリプト化を容易にしている。

Alicia の起動方法、API などは、付録 9.1 Alicia の実行方法を参照されたい。

2.2 Alicia1.0 からの課題

2.2.1 ダンプ解析ツールの統合化

Alicia には、ダンプ解析ツールの統合化を目指すという目的がある。Alicia1.0 は、crash をラッピングしたので、Alicia1.1 では lcrash のラッピングを行う。この対応により、ダンプ解析者は、Alicia の機能を crash と lcrash の両方に使うことができる。また、Alicia の API を使ってダンプ解析用のスクリプトを作成しておけば、ラッピングされるツールを意識することなく、いつも同じ結果を得ることができる。

2.2.2 LDAS の充実

Alicia の有効性の中で重要なものの一つに、使用者が簡単にスクリプトを作り、それを用いてダンプ解析を行えることが挙げられる。このことは、Alicia がダンプ解析ツールとして多くの解析者により成長を続けて行くタイプのツールであることを意味する。例えば、複数の構造体がポインタでチェーンされたリストをたぐりながら繰り返し処理をしたり、多くの情報から目的のデータを探したり、と言った簡単な機械的操作を行いたい場合に、ユーザが手入力のコマンドで会話的にダンプ解析するのに比べて、スクリプトを用いれば比較にならないほど、手間と時間の短縮になる。更に、作ったスクリプトは使い捨てではなく、保存して再利用することで、次回のダンプ解析に有効活用される。つまり、そのスクリプトは Alicia に新しいコマンドとして機能追加されたことに相当する。

そこで、今回は、以下の 2 つの LDAS を作成することで、LDAS を充実させる。

(1) 初期 LDAS

普段からカーネルのソースコードに親しんでいる技術者は、crash や lcrash などのダンプ解析ツールを立ち上げたときに、どんなコマンドを入力していけば解析を進めることができるかが分かっている。しかしながら、カーネルの開発者ではないシステム管理者などは、解析ツールを起動したはよいが、そこから解析を進めていくために何をすればよいかが分からない。

そこで、ダンプを解析するきっかけとなるコマンドを用意することができれば、ダンプ解析へのハードルをぐっと下げることができると考えた。

(2) LKST 連携 LDAS

ダンプとは、システムが停止する瞬間のメモリの内容であるため、過去のメモリの状態がどうなっていたのかを解析するのは困難である。しかしながら、LKST(Linux Kernel State Tracer:詳細は参考文献の URL を参照)が採取しているデータを利用すれば、過去に遡って解析することができる。

3 実施内容

「2.2 Alicia1.0 からの課題」に対応するため、以下の内容について Alicia の機能拡張を行った。

- lcrash 対応
- 初期 LDAS 作成
- LKST 連携

図 3-1 に Alicia の構成を示す。Alicia は、Wrapper モジュール部分と LDAS 部分で構成されており、今回は、Wrapper モジュールの lcrash 対応部分と LDAS に初期 LDAS と LKST 連携の LDAS を追加した。

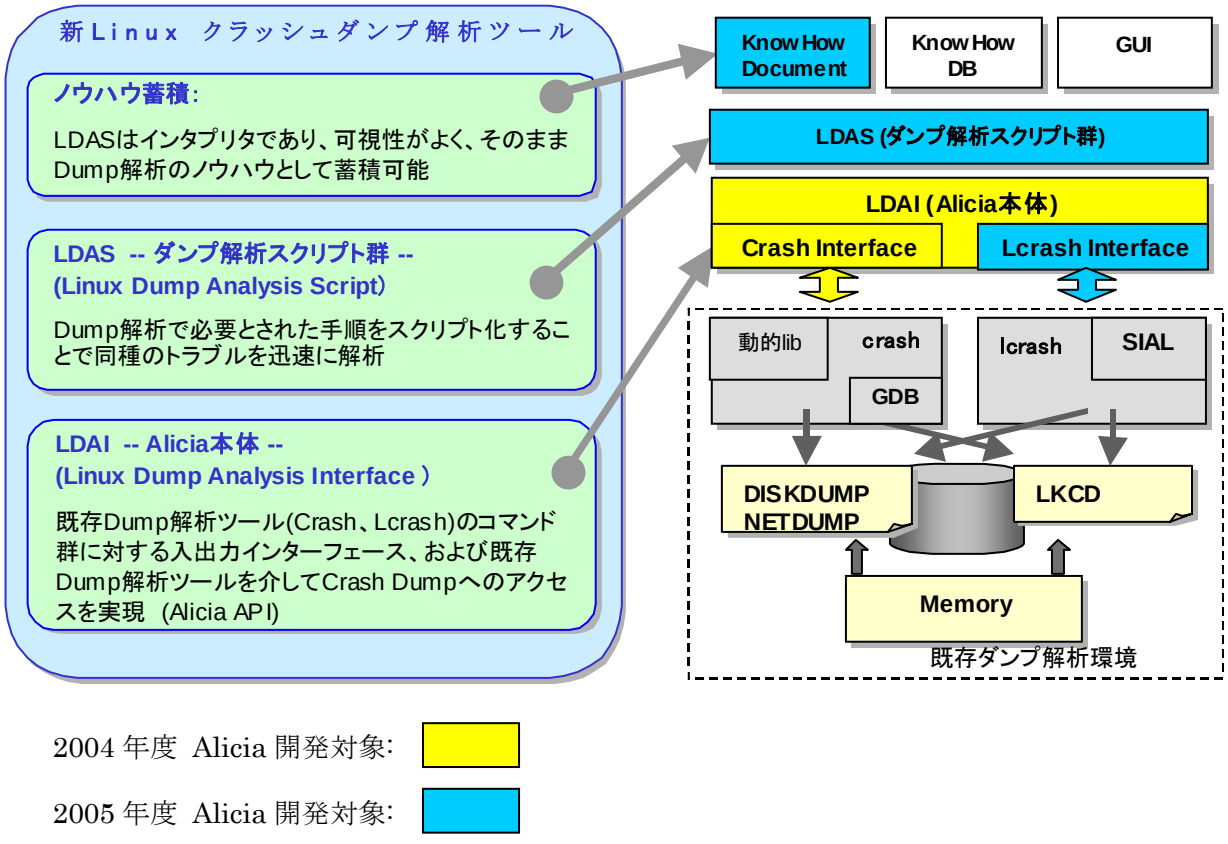

図 3-1 Alicia の構成

3.1 lcrash 対応

Alicia は今回リリースしたバージョン 1.1.0 より lcrash に対応した。ここでは、Alicia の各種 API、LDAS が crash、lcrash 共に同様の出力結果となることを確認する。また、API の crash、lcrash での実行時間を測定し比較する。

3.1.1 Alicia の起動方法

ここでは、lcrash モードでの Alicia の起動方法を示す。crash モードでの起動方法については、付録を参照されたい。

$alicia -lcrash [lcrash の引数に従う]

```
$ alicia -lcrash map dump kerntypes
```

次項より、Alicia の API が lcrash でも同じ結果を返すことを確認する。

3.1.2 kernel 関数

kernel 関数を使って、あるタスクの構造体 task_struct.comm を取得する。

ここでは、初期タスク init_task_union で確認する。

lcrash のコマンド"symbol"を利用して、初期タスクの task_struct 構造体のアドレスを得る。

```
alicia> symbol init_task_union;
```

```
       ADDR  OFFSET SECTION     NAME                TYPE
================================================================================
0xc049e000      0 GLOBAL_DATA  init_task_union      (unknown)
================================================================================
1 symbol found
```

kernel 関数を使って、task_struct のメンバ"comm"を表示する。

```
alicia> print kernel('c049e000', 'task_struct', 'comm', 'char *')
```

crash 側、lcrash 側とも同様の出力結果が得られる。

```
swapper
```

3.1.3 get_addr 関数

get_addr 関数を実行。Linux の外部参照静的変数"banner"のアドレスを得る。

```
alicia> print get_addr 'banner'
```

crash 側、lcrash 側とも同様の出力結果が得られる。

```
0xc04db300
```

3.1.4 get_mem 関数

get_mem 関数で、banner の内容を得る。

```
alicia> @banner = get_mem '0xc04db300', 15
```

リトルエンディアンのデータを文字列に変換して出力する。

```
alicia> print join '', map { pack "V", hex($_) } @banner
```

crash 側、lcrash 側とも同様の出力結果が得られる。

```
<6>NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
```

3.1.5 LDAS の実行

前節のオペレーションを、スクリプト化してみる。

シェルエスケープを使って、外部エディタを起動して、LDAS を編集する。

```
alicia>!vi banner.ldas
```

Alicia 上でのコマンド名をサブルーチン名としてスクリプトを作成する。

```
sub banner {
    my $addr = get_addr 'banner';
    print join '', map { pack "V", hex($_) } get_mem($addr, 15);
}
1;
```

作成した LDAS を Alicia 上にローディングする。

```
alicia> load 'banner.ldas'
```

LDAS を実行する。

```
alicia> banner
```

crash 側、lcrash 側とも同様の出力結果が得られる。

```
<6>NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
```

3.1.6 API の実行時間の比較

Perl の Benchmark モジュールを使用し、上記で実行した各 API を 10000 回実行して時間を測定する LDAS を作成する。

```
alicia> !vi benchmark.ldas
```

```
sub benchmark {
    use Benchmark;
```

```
    my $count = 10000;
    timethese($count,{
        'get_addr' => '&_get_addr_test',
        'get_mem' => '&_get_mem_test',
        'kernel' => '&_kernel_test',
        'ldas' => '&_ldas_test',
    });
};
sub _get_addr_test {
    get_addr 'banner';
};
sub _get_mem_test {
    get_mem '0xc04db300', 15;
};
sub _kernel_test {
    kernel('c049e000', 'task_struct', 'comm', 'char *');
};
sub _ldas_test {
    my $addr = get_addr 'banner';
    join '', map { pack "V", hex($_) } get_mem($addr, 15);
};
1;
```

benchmark.ldas の実行

```
alicia> load 'benchmark.ldas'
alicia> benchmark
```

以下に、benchmark.ldas による測定環境と結果を示す。

・ソフトウェア

カーネル： MIRACLE LINUX V3.0 SP1 (2.4.21-20.19AX)

Perl： 5.8.0

Alicia： 1.1.0

・ハードウェア

CPU： Pentium M 1300MHz

メモリ： 512MB

・対象ダンプ

カーネル： MIRACLE LINUX V3.0 SP1 (2.4.21-20.19AX)

crash の測定結果

```
mark: timing 10000 iterations of get_addr, get_mem, kernel, ldas...
  get_addr:  5 wallclock secs ( 0.57 usr +  0.06 sys =  0.63 CPU) @ 15873.02/s (n=10000)
   get_mem:  5 wallclock secs ( 1.85 usr +  0.08 sys =  1.93 CPU) @ 5181.35/s (n=10000)
    kernel: 173 wallclock secs ( 2.00 usr +  0.13 sys =  2.13 CPU) @ 4694.84/s (n=10000)
      ldas: 10 wallclock secs ( 2.97 usr +  0.09 sys =  3.06 CPU) @ 3267.97/s (n=10000)
```

lcrash の測定結果

```
Benchmark: timing 10000 iterations of get_addr, get_mem, kernel, ldas...
  get_addr:  1 wallclock secs ( 0.63 usr +  0.04 sys =  0.67 CPU) @ 14925.37/s (n=10000)
   get_mem:  4 wallclock secs ( 2.75 usr +  0.03 sys =  2.78 CPU) @ 3597.12/s (n=10000)
    kernel:  2 wallclock secs ( 0.93 usr +  0.01 sys =  0.94 CPU) @ 10638.30/s (n=10000)
      ldas:  5 wallclock secs ( 3.87 usr +  0.03 sys =  3.90 CPU) @ 2564.10/s (n=10000)
```

結果をまとめると表 3.1-1のようになる。(単位は秒)

表 3.1-1　Alicia API 実行時間

	get_addr	get_mem	kernel	ldas
crash	5	5	173	10
lcrash	1	4	2	5

lcrash は crash に比べ全体的に実行動作が早いことがわかる。

また、crash の kernel 関数は特に遅くなっている。これは、crash の kernel 関数が、crash 自前の機能ではなく、crash 内でラッピングしている GDB の機能を利用しているためのオーバヘッドと考えられる。

3.2 初期 LDAS

3.2.1 初期 LDAS とは

障害発生によって採取されたダンプを解析する際に、どのような情報から見ていくかといった手順は障害の内容に応じてまちまちであり、解析者がそれまでの経験と勘を駆使して試行錯誤的に、カーネル内の各種情報を調べていくといったケースが多い。

しかし、カーネルがおおよそどんな状態になっているのかについては、どんな場合でも個々の解析において初期段階で把握しておきたい。多くのユーザが同じような情報を編集して表示させるスクリプトを独自に開発するよりも、今回開発した初期 LDAS を使用すれば、必要最低限のカーネルの状態を簡単に見ることができる。

図 3.2-1 に障害発生から、ユーザが初期 LDAS の解析結果を得るまでの構成図を示す。

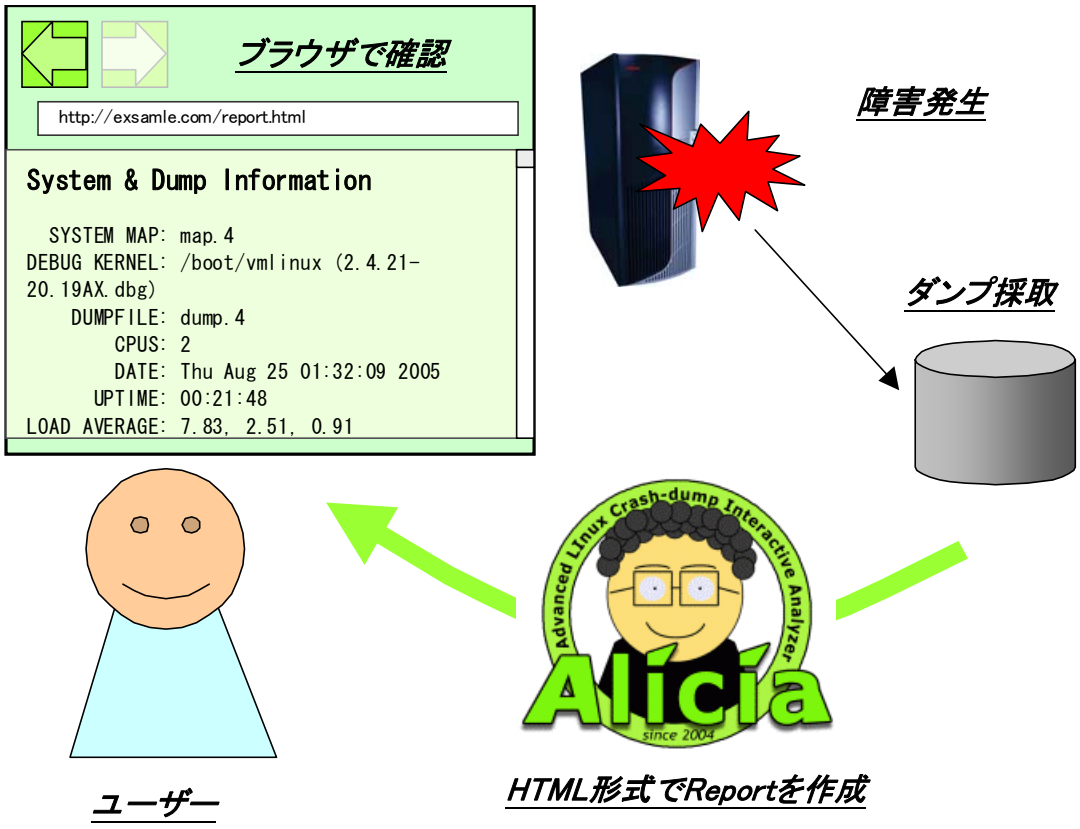

図 3.2-1 初期 LDAS 構成

表 3.2-1 は、初期 LDAS で採取される情報の一覧である。

表 3.2-1　初期 LDAS で採取される情報

情　報	項
システムおよびダンプ情報	3.2.3
停止情報	3.2.4
スタック情報	3.2.5
停止関数の情報	3.2.6
プロセス情報	3.2.7
ランキュー情報	3.2.8
メモリアロケーション情報	3.2.9
スワップ情報	3.2.10
ロック情報	3.2.11
マウント情報	3.2.12
ロードモジュール情報	3.2.13
ネットワーク情報	3.2.14
プロック情報	3.2.15

3.2.2 実行方法

通常、LDAS は load コマンドによって Alicia 上に読み込むが、初期 LDAS は Alicia 起動時に読み込まれているため、load の実行の必要はない。

DUMP 解析のサマリ情報を指定したディレクトリ/ファイル名に、詳細情報を、指定したディレクトリ/MMDDhhmmss/”下に出力する。以下の例では、/demo/report.html に出力する。

注）詳細情報が格納されるディレクトリ名は実行時の時間
MM:月　DD:日　hh:時　mm：分　ss:秒

```
alicia> report(output=>'/demo/report.html')
```

出力中の詳細情報のファイル名が表示される。

```
Writing to log.txt
Writing to irq.txt
Writing to io_dev.txt
Writing to task.txt
Writing to file_info.txt
Writing to all_task.txt
Writing to slab_info.txt
alicia>
```

次項よりサマリ情報で表示される項目を項目毎に見ていく。

3.2.3 システムおよびダンプ情報

CPU の数や DUMP ファイル名、DUMP の採取日時等の情報が表示される。

```
System & Dump Information

  SYSTEM MAP: map.4
DEBUG KERNEL: /boot/vmlinux (2.4.21-20.19AX.dbg)
    DUMPFILE: dump.4
        CPUS: 2
        DATE: Thu Aug 25 01:32:09 2005
      UPTIME: 00:21:48
LOAD AVERAGE: 7.83, 2.51, 0.91
       TASKS: 84
```

```
     NODENAME: miracle3
      RELEASE: 2.4.21-20.19AXsmp
      VERSION: #1 SMP Wed Jan 5 05:02:09 EST 2005
      MACHINE: i686  (2392 Mhz)
       MEMORY: 2 GB
        PANIC: "Fatal exception"
```

3.2.4 停止情報

停止時のレジスタの内容、スタック情報や停止原因メッセージ等の停止情報が表示さる。

```
Stop Log

 Oops: 0002
 audit e1000 floppy sg lkst microcode dump_gzip dump keybdev mousedev hid input usb-uhci
usbcore ext3 jbd mptscsih mptbase diskdumplib sd_mod scsi_mod
 CPU:    1
 EIP:    0060:[<c01a7490>]    Not tainted
 EFLAGS: 00010292

 EIP is at sys_flock [kernel] 0x0 (2.4.21-20.19AXsmp/i686)
 eax: 0000008f   ebx: ed96c000   ecx: f8c9a780   edx: 00000000
 esi: c04a00d1   edi: 0953d2c0   ebp: bfebd578   esp: ed96dfc0
 ds: 0068   es: 0068   ss: 0068
 Process perl (pid: 4642, stackpage=ed96d000)
 Stack: c04a0036 00000003 00000002 00ede5a8 00000002 0953d2c0 bfebd578 0000008f
        0000002b 0000002b 0000008f 0022f2c1 00000023 00000246 bfebd54c 0000002b
 Call Trace:
 Code: 00 00 56 53 bb f7 ff ff ff 83 ec 38 8b 44 24 4c 8b 6c 24 50

 Kernel panic: Fatal exception



 dump: Dumping to device 0x802 [sd(8,2)] on CPU 1 ...
 dump: Compression value is 0x2, Writing dump header

 dump: Pass 1: Saving Kernel Pages:
 dump: Memory Bank[0]: 0 ... 7ffcffff:
```

3.2.5 スタック情報

プロセスのスタックトレースおよび停止時のレジスタの内容等のスタック情報が表示される。

Stack Trace & Register

```
PID: 4642   TASK: ed96c000  CPU: 1   COMMAND: "perl"
bt: cannot resolve stack trace:
 #0 [ed96dbf0] LKST_ETYPE_PROCESS_CONTEXTSWITCH_HEADER_hook at c012ecb2
 #1 [ed96dbf4] build_tree at f88d8196
 #2 [ed96dc1c] build_bl_tree at f88d897d
 #3 [ed96dc30] _tr_flush_block at f88d9153
bt: text symbols on stack:
    [ed96dbf4] build_tree at f88d819b
    [ed96dc10] copy_block at f88d9b17
    [ed96dc1c] build_bl_tree at f88d8982
    [ed96dc30] _tr_flush_block at f88d9158
    [ed96dc58] deflate_slow at f88d78c9
    [ed96dc80] deflate at f88d6833
    [ed96dc90] deflateEnd at f88d6a56
    [ed96dca8] dump_compress_gzip at f88dd0c8
    [ed96dcb4] cleanup_module at f88dd162
    [ed96dcdc] dump_compress_gzip_alloc at f88dcf40
    [ed96dce0] dump_compress_gzip_free at f88dcfd0
    [ed96dd08] dump_add_page at f88c9688
    [ed96dd24] dump_kernel_write at f88c93e5
    [ed96dd6c] dump_execute_memdump at f88ca19c
    [ed96ddc0] dump_execute at f88ca607
    [ed96ddd8] LKST_ETYPE_0_PANIC_HEADER_hook at c01334f8
    [ed96dde8] die at c010d7ef
    [ed96de2c] LKST_ETYPE_0_PANIC_HEADER_hook at c0133402
    [ed96de4c] die at c010d7ef
    [ed96de6c] locks_alloc_lock at c01a4f18
    [ed96de84] handle_mm_fault at c01599a2
    [ed96de98] LKST_ETYPE_OOPS_PGFAULT_HEADER_hook at c01255d1
    [ed96deac] dput at c01a9070
    [ed96df20] dentry_open at c01832b0
    [ed96df34] sys_flock at c01a7490
    [ed96df44] do_page_fault at c0125260
    [ed96df54] do_page_fault at c0125260
```

```
    [ed96df5c] error_code at c04a022c
    [ed96df64] do_page_fault at c0125260
    [ed96df78] do_page_fault at c0125260
    [ed96df80] error_code at c04a023d
    [ed96df98] no_timing at c04a00d1
    [ed96dfb4] sys_flock at c01a7490
    [ed96dfc0] LKST_ETYPE_SYSCALL_ENTRY_HEADER_hook at c04a0036
bt: possible exception frames:
  KERNEL-MODE EXCEPTION FRAME AT ed96de04:
    EAX: 00000001  EBX: c0328e43  ECX: c0400f40  EDX: 0000000d  EBP: c032afb1
    DS:  0068      ESI: ed96df8c  ES:  0068      EDI: c03fefbc
    CS:  0060      EIP: c0133402  ERR: 00001222  EFLAGS: 00000092
  KERNEL-MODE EXCEPTION FRAME AT ed96df8c:
    EAX: 0000008f  EBX: ed96c000  ECX: f8c9a780  EDX: 00000000  EBP: bfebd578
    DS:  0068      ESI: c04a00d1  ES:  0068      EDI: 0953d2c0
    CS:  0060      EIP: c01a7490  ERR: ffffffff  EFLAGS: 00010292
  USER-MODE EXCEPTION FRAME AT ed96dfc4:
    EAX: 0000008f  EBX: 00000003  ECX: 00000002  EDX: 00ede5a8
    DS:  002b      ESI: 00000002  ES:  002b      EDI: 0953d2c0
    SS:  002b      ESP: bfebd54c  EBP: bfebd578
    CS:  0023      EIP: 0022f2c1  ERR: 0000008f  EFLAGS: 00000246
```

3.2.6 停止関数の情報

停止した時に実行していたカーネル関数の全アセンブルコードが表示される。

```
Last Function

Last function is sys_flock <c01a7490>.
----
0xc01a7490 <sys_flock>:        add    %al,(%eax)
----
0xc01a7492 <sys_flock+2>:      push   %esi
0xc01a7493 <sys_flock+3>:      push   %ebx
0xc01a7494 <sys_flock+4>:      mov    $0xfffffff7,%ebx
0xc01a7499 <sys_flock+9>:      sub    $0x38,%esp
0xc01a749c <sys_flock+12>:     mov    0x4c(%esp),%eax
0xc01a74a0 <sys_flock+16>:     mov    0x50(%esp),%ebp
0xc01a74a4 <sys_flock+20>:     call   0xc0186b90 <fget>
0xc01a74a9 <sys_flock+25>:     test   %eax,%eax
0xc01a74ab <sys_flock+27>:     mov    %eax,%edi
```

```
0xc01a74ad <sys_flock+29>:      je     0xc01a7591 <sys_flock+257>
0xc01a74b3 <sys_flock+35>:      test   $0x20,%ebp
0xc01a74b9 <sys_flock+41>:      je     0xc01a76d0 <sys_flock+576>
0xc01a74bf <sys_flock+47>:      mov    %ebp,%esi
0xc01a74c1 <sys_flock+49>:      and    $0xe0,%esi
0xc01a74c7 <sys_flock+55>:      test   %esi,%esi
0xc01a74c9 <sys_flock+57>:      mov    %esi,%ebx
＜中略＞
0xc01a7708 <sys_flock+632>:     jmp    0xc01a76eb <sys_flock+603>
0xc01a770a <sys_flock+634>:     movl   $0x42,(%esp)
0xc01a7711 <sys_flock+641>:     call   0xc01336a0 <__out_of_line_bug>
0xc01a7716 <sys_flock+646>:     lea    0x0(%esi),%esi
0xc01a7719 <sys_flock+649>:     lea    0x0(%edi),%edi
```

3.2.7 プロセス情報

プロセス ID、実行 CPU 番号、タスクストラクト・アドレス、使用メモリサイズ等のプロセス情報が表示される。

Process

PID	PPID	CPU	TASK	ST	%MEM	VSZ	RSS	COMM
0	0	0	c049e000	RU	0.0	0	0	[swapper]
0	1	1	c4cd8000	RU	0.0	0	0	[swapper]
1	0	1	f7fa4000	RU	0.0	2916	512	init
2	0	0	f7fa0000	IN	0.0	0	0	[migration/0]
3	0	1	c4ede000	IN	0.0	0	0	[migration/1]
4	1	0	c343e000	IN	0.0	0	0	[keventd]
5	1	0	c343c000	IN	0.0	0	0	[ksoftirqd/0]
6	1	1	c4cd6000	IN	0.0	0	0	[ksoftirqd/1]
7	1	0	f7f88000	RU	0.0	0	0	[kswapd]
8	1	0	f7f86000	RU	0.0	0	0	[kscand]
9	1	1	c4ee4000	IN	0.0	0	0	[bdflush]
10	1	1	c4ee2000	RU	0.0	0	0	[kupdated]
11	1	0	c4cb8000	IN	0.0	0	0	[mdrecoveryd]
21	1	0	f63dc000	RU	0.0	0	0	[kjournald]
82	1	1	c376c000	IN	0.0	0	0	[khubd]
1292	1	1	c35e6000	IN	0.0	0	0	[kjournald]
1293	1	1	f619a000	IN	0.0	0	0	[kjournald]
2131	1	0	c34c4000	RU	0.0	2692	620	syslogd
2135	1	1	c4c7a000	RU	0.0	2928	472	klogd

```
    2145      1   0  c4c72000  RU   0.0    3416    452  irqbalance
    2157      1   1  c4c68000  IN   0.0   84048    608  auditd
    2192      1   1  f63ce000  IN   0.1    5052   1572  sshd
    2206      1   0  f6196000  RU   0.5   22928   9604  httpd
    2215      1   0  c4d30000  RU   0.0    2852    644  crond
    2243      1   1  c4d38000  RU   0.2    6468   3692  xfs
    2252      1   1  c4d46000  RU   0.0    2692    580  atd
<中略>
    4573   3237   1  eec36000  IN   0.0    2520    472  gcc
    4574   4573   0  ee58c000  RU   0.8   19716  17516  cc1
    4575   4573   0  ee82e000  IN   0.0    3244    896  as
    4621   2647   0  ef2be000  IN   0.0    2444    468  gcc
    4622   4621   1  ed086000  RU   0.4   12564   8492  cc1
    4623   4621   0  ed0ca000  IN   0.0    3388    864  as
    4628      1   0  ecefa000  IN   0.1    7704   1752  wget
    4630      1   1  ed4bc000  IN   0.1    8504   1748  wget
    4631   3291   0  eea44000  RU   0.3    8404   5896  wiki.cgi
    4632   2269   1  ecfd8000  RU   0.3    8356   6256  wiki.cgi
    4633      1   1  ecf18000  IN   0.1    7756   1752  wget
    4636      1   1  ed7de000  IN   0.1    7928   1752  wget
    4639   2275   0  ef444000  RU   0.1    6484   1068  apache_test.sh
    4640   4267   0  ecf7c000  RU   0.2    7392   4280  wiki.cgi
    4641   2268   1  ed18a000  RU   0.2    7424   4964  wiki.cgi
>   4642   2498   1  ed96c000  RU   0.1    7512   1368  perl
    4650   4639   0  ed146000  IN   0.1    7888   1752  wget
    4652   4639   0  ed1ee000  RU   0.1    7712   1424  wget
    4653   4639   0  ed4e0000  RU   0.1    7744   1468  wget
    4654   2265   1  ed458000  RU   0.1    5012   2464  wiki.cgi
>   4657   4639   0  ecf38000  RU   0.0    4704    704  wget
```

3.2.8 ランキュー情報

実行待ちのプロセスの PID、タスクストラクト・アドレス、実行 CPU 番号等のランキュー情報が表示される。

```
Run Queue

RUNQUEUES[0]: c04eb280
 ACTIVE PRIO_ARRAY: c04eb71c
  [115] PID: 2135   TASK: c4c7a000  CPU: 1   COMMAND: "klogd"
        PID: 8      TASK: f7f86000  CPU: 0   COMMAND: "kscand"
```

```
        PID: 2145   TASK: c4c72000  CPU: 0   COMMAND: "irqbalance"
        PID: 2206   TASK: f6196000  CPU: 0   COMMAND: "httpd"
        PID: 7      TASK: f7f88000  CPU: 0   COMMAND: "kswapd"
        PID: 2131   TASK: c34c4000  CPU: 0   COMMAND: "syslogd"
        PID: 10     TASK: c4ee2000  CPU: 1   COMMAND: "kupdated"
        PID: 21     TASK: f63dc000  CPU: 0   COMMAND: "kjournald"
        PID: 2271   TASK: c4e06000  CPU: 1   COMMAND: "httpd"
        PID: 2243   TASK: c4d38000  CPU: 1   COMMAND: "xfs"
        PID: 2252   TASK: c4d46000  CPU: 1   COMMAND: "atd"
        PID: 2215   TASK: c4d30000  CPU: 0   COMMAND: "crond"
        PID: 2496   TASK: f60ce000  CPU: 0   COMMAND: "sshd"
        PID: 2452   TASK: f60ee000  CPU: 0   COMMAND: "sshd"
  [121] PID: 4654   TASK: ed458000  CPU: 1   COMMAND: "wiki.cgi"
  [124] PID: 4657   TASK: ecf38000  CPU: 0   COMMAND: "wget"
        PID: 4639   TASK: ef444000  CPU: 0   COMMAND: "apache_test.sh"
        PID: 4574   TASK: ee58c000  CPU: 0   COMMAND: "cc1"
        PID: 4622   TASK: ed086000  CPU: 1   COMMAND: "cc1"
  [125] PID: 4652   TASK: ed1ee000  CPU: 0   COMMAND: "wget"
 EXPIRED PRIO_ARRAY: c04eb2a4
  [120] PID: 4642   TASK: ed96c000  CPU: 1   COMMAND: "perl"
  [121] PID: 4632   TASK: ecfd8000  CPU: 1   COMMAND: "wiki.cgi"
        PID: 4640   TASK: ecf7c000  CPU: 0   COMMAND: "wiki.cgi"
        PID: 4641   TASK: ed18a000  CPU: 1   COMMAND: "wiki.cgi"
        PID: 4631   TASK: eea44000  CPU: 0   COMMAND: "wiki.cgi"
  [125] PID: 4653   TASK: ed4e0000  CPU: 0   COMMAND: "wget"
RUNQUEUES[1]: c04ebc80
 ACTIVE PRIO_ARRAY: c04ebca4
 EXPIRED PRIO_ARRAY: c04ec11c
```

3.2.9 メモリアロケーション情報

カーネルメモリの割り当てページ、メモリサイズ、使用%数等のメモリアロケーション情報が表示される。

```
Memory Allocation

              PAGES        TOTAL      PERCENTAGE
 TOTAL MEM   513658          2 GB         ----
      FREE   286617        1.1 GB   55% of TOTAL MEM
      USED   227041      886.9 MB   44% of TOTAL MEM
    SHARED    27856      108.8 MB    5% of TOTAL MEM
```

```
    BUFFERS    11113       43.4 MB    2% of TOTAL MEM
     CACHED   199414        779 MB   38% of TOTAL MEM
       SLAB     9598       37.5 MB    1% of TOTAL MEM

TOTAL HIGH   294864        1.1 GB   57% of TOTAL MEM
 FREE HIGH    90603      353.9 MB   30% of TOTAL HIGH
 TOTAL LOW   218794      854.7 MB   42% of TOTAL MEM
  FREE LOW   196014      765.7 MB   89% of TOTAL LOW

TOTAL SWAP   562273        2.1 GB         ----
 SWAP USED        0             0    0% of TOTAL SWAP
 SWAP FREE   562273        2.1 GB  100% of TOTAL SWAP
```

3.2.10 スワップ情報

スワップ領域の装置名、サイズ、等のスワップ情報が表示される。

```
Swap

FILENAME          TYPE         SIZE      USED   PCT  PRIORITY
/dev/sda2        PARTITION    2249092k       0k   0%     -1
```

3.2.11 ロック情報

ファイル・ロック情報とグロバールロックのアドレス、メモリ内容等のロック情報が表示される。

```
Lock


File Lock

1: FLOCK 2546  perl           lockf
1: ->FLOCK 2547  perl           lockf
1: ->FLOCK 2548  perl           lockf
2: POSIX 2252  atd            atd.pid
3: FLOCK 2214  UNKNOWN        crond.pid

Global Lock

              global name   address     content
 ======================  ==========  ====================
            atm_dev_lock  0xc049b43c  0x00000001  0x00000000
```

```
              dcache_lock  0xc0501780  0x00000001  0x00000000
             dev_base_lock  0xc04575c8  0x01000000  0x00000000
                 die_lock  0xc03fefbc  0x00000000  0x00000001
            dma_spin_lock  0xc0400c20  0x00000001  0x00000000
               files_lock  0xc044a07c  0x00000001  0x00000000
           global_bh_lock  0xc0401034  0x00000001  0xc032b715
               i8253_lock  0xc03ffd34  0x00000001  0x00002e9c
              i8259A_lock  0xc03ff740  0x00000001  0x00000000
         inet_peer_idlock  0xc0495aa0  0x00000001  0x00000000
    inet_peer_unused_lock  0xc0495b00  0x00000001  0x00000000
             inetdev_lock  0xc0499404  0x01000000  0xc02d21a0
          io_request_lock  0xc0454588  0x00000001  0x00000000
               ip_ra_lock  0xc0495b88  0x01000000  0x00001770
              mmlist_lock  0xc04ff300  0x00000001  0x00000000
             modlist_lock  0xc0401004  0x00000001  0x00000001
      net_big_sklist_lock  0xc0494854  0x01000000  0x00000080
            notifier_lock  0xc0423250  0x01000000  0x00000000
                oops_lock  0xc0400b98  0x00000000  0xc0400b9c
pagecache_lock_cacheline  0xc0445c00  0x00000001  0x00000000
      proc_alloc_map_lock  0xc044b680  0x00000001  0x00000000
          qdisc_tree_lock  0xc0495300  0x01000000  0x00000000
               raw_v4_lock  0xc0496dc0  0x01000000  0x00000000
                  rtc_lock  0xc03ffd30  0x00000001  0x00000001
                   sb_lock  0xc044a2e8  0x00000001  0x00000001
              shmem_ilock  0xc0449b08  0x00000001  0x0000003e
                 swaplock  0xc0445f94  0x00000001  0x00000000
     task_capability_lock  0xc0402160  0x00000001  0x00000000
            tasklist_lock  0xc04ff280  0x01000000  0x00000000
          timerlist_lock  0xc042320c  0x00000001  0x00000094
             tqueue_lock  0xc0423200  0x00000001  0x00000000
         tux_module_lock  0xc0494820  0x00000001  0x00000000
           udp_hash_lock  0xc0497e60  0x01000000  0x00000000
         unix_table_lock  0xc049ae84  0x01000000  0x00000004
             vcache_lock  0xc0449ddc  0x00000001  0x00000000
             vmlist_lock  0xc0445c94  0x01000000  0x00000000
```

3.2.12 マウント情報

マウントされているファイルのスーパブロック、デバイス名、ファイルタイプ等のマウント情報が表示される。

Mount

```
VFSMOUNT SUPERBLK TYPE   DEVNAME   DIRNAME
c4ed5100 c4ed3000 rootfs rootfs    /
c4ed5ec0 f77e4c00 ext3   /dev/root /
c4ed5e80 c4ed3800 proc   /proc     /proc
c4ed5e40 c4eb4400 devpts none      /dev/pts
c4ed5f00 c34cfc00 usbdevfs usbdevfs  /proc/bus/usb
c4eb6e80 f77e4400 ext3   /dev/sda1 /boot
c4eb6e40 f77e4800 tmpfs  none      /dev/shm
c4eb6e00 c4d8c000 ext3   /dev/sdb1 /mnt/sdb
```

3.2.13 ロードモジュール情報

カーネルのモジュール名、サイズ等のロードモジュール情報が表示される。

Load Module

```
 MODULE   NAME           SIZE  OBJECT FILE
f880d000  scsi_mod     126376  (not loaded)  [CONFIG_KALLSYMS]
f882f000  sd_mod        13712  (not loaded)  [CONFIG_KALLSYMS]
f8836000  diskdumplib    5228  (not loaded)  [CONFIG_KALLSYMS]
f883b000  mptbase       45280  (not loaded)  [CONFIG_KALLSYMS]
f884e000  mptscsih      53264  (not loaded)  [CONFIG_KALLSYMS]
f885f000  jbd           64820  (not loaded)  [CONFIG_KALLSYMS]
f8872000  ext3          98184  (not loaded)  [CONFIG_KALLSYMS]
f888d000  usbcore       91360  (not loaded)  [CONFIG_KALLSYMS]
f88a7000  usb-uhci      32524  (not loaded)  [CONFIG_KALLSYMS]
f88b2000  input          6368  (not loaded)  [CONFIG_KALLSYMS]
f88b7000  hid           22596  (not loaded)  [CONFIG_KALLSYMS]
f88c0000  mousedev       6040  (not loaded)  [CONFIG_KALLSYMS]
f88c5000  keybdev        2976  (not loaded)  [CONFIG_KALLSYMS]
f88c9000  dump          37056  (not loaded)  [CONFIG_KALLSYMS]
f88d6000  dump_gzip     42764  (not loaded)  [CONFIG_KALLSYMS]
f89fd000  microcode      7136  (not loaded)  [CONFIG_KALLSYMS]
f8a02000  lkst          79008  (not loaded)  [CONFIG_KALLSYMS]
f8a3d000  sg            42604  (not loaded)  [CONFIG_KALLSYMS]
f8a53000  floppy        62544  (not loaded)  [CONFIG_KALLSYMS]
f8a70000  e1000         87624  (not loaded)  [CONFIG_KALLSYMS]
f8af0000  audit         94776  (not loaded)  [CONFIG_KALLSYMS]
```

3.2.14 ネットワーク情報

ネットデバイスのアドレス、デバイス名、IP アドレス等のネットワーク情報が表示される。

```
Network

NET_DEVICE  NAME   IP ADDRESS(ES)
 c0457420   lo     127.0.0.1
 f367a800   eth0   172.24.33.86
 c4e4f000   eth1
 f367a000   eth2
```

3.2.15 プロック情報

/proc/sys/kernel、/proc/sys/vm、/proc/sys/fs、/proc/sys/net に格納されている情報が表示される。

```
Proc file

 The files that is supported are:
 acct cmdline file_max ip_forward max_map_count msgmax msgmnb msgmni page_cluster
pid_max rmem_default rmem_max sem tcp_ecn tcp_keepalive_time tcp_rmem threads_max uptime
wmem_default wmem_max
 /proc/sys/kernel
 acct =
 ACCT PARM[0] = 0x00000004
 ACCT PARM[1] = 0x00000002
 ACCT PARM[2] = 0x0000001e

 msgmax = 0x00002000
 msgmnb = 0x0000ffff
 msgmni = 0x00000b3e
 pid_max = 32768
 sem =
 SEMMSL = 0x00000100
 SEMMNS = 0x00007d00
 SEMOPM = 0x00000064
 SEMMNI = 0x0000008e

 threads_max = 14336

 /proc/sys/vm
```

```
max_map_count = 0x00010000
page_cluster = 0x00000003
/proc/sys/fs
file_max = 0x20000
/proc/sys/net
ip_forward = 0
rmem_default = 65535
rmem_max = 131071
tcp_ecn = 0
tcp_keepalive_time = 7200
tcp_rmem = 4096
wmem_default = 65535
wmem_max = 131071
```

3.2.16 詳細データ

指定したディレクトリ下に保存される詳細情報は次の通りである。

① ログ情報ファイル（log.txt）
HIGHMEM、LOWMEM のサイズ、ZONE のページ数、CPU の型番等のカーネルメッセージが出力される。

② IRQ ファイル（irq.txt）
各 IRQ のインタラプト情報が出力される。

③ I/O 情報ファイル（io_dev.txt）
各 I/O デバイスの I/O 要求数、待ち状態の I/O キュー、などの情報が出力される。

④ タスク情報ファイル（task.txt）
CPU に割り当てられていたプロセスのタスク･ストラクト内情報が出力される。

⑤ ファイル情報ファイル（file_info.txt）
システム内にオープンされている全ファイル INODE、DENTRY、ファイルストラクト・アドレスパス名等のファイル情報が出力される。

⑥ 全タスク情報ファイル（all_task.txt）
システム内の全プロセスのタスク・スラクト内情報が出力される。

⑦ メモリ slab 情報ファイル（slab_info.txt）
カーネル slab アロケータの統計情報が出力される。

3.3 LKST データの解析

ダンプに残された LKST のデータを解析するためのインフラを Alicia 上で LDAS として開発した。ダンプが採取される直前までのデータを参照できるので、ダンプが採取された

瞬間だけではなく過去に遡って解析を実施できる。今回は、インフラの動作を確認するために、ランキューの数の変遷とシステムコールの使用状況を調査するアナライザを作成した。

3.3.1 解析環境

・ソフトウェア

カーネル： MIRACLE LINUX V3.0 SP1 (2.4.21-20.19AX)

Perl： 5.8.0

Alicia： Alicia-1.1.0

・ハードウェア

CPU： Pentium M 1300MHz

メモリ： 512MB

・対象ダンプ

カーネル： MIRACLE LINUX V3.0 SP1 + lkst2.21 対応
(2.4.21-20.19AX.lkst1smp)

3.3.2 LKST データ採取

MIRACLE LINUX V3.0 では、LKST がデフォルトでインストールされているため、カーネルへのパッチ等の作業がいらない。実際のデータはデフォルトでは採取されないため、以下のように LKST のバッファとサイズを 1M、イベントマスクはデフォルトを使用して LKST データの採取を行う。

```
# modprobe lkst
# lkstbuf create -s 1M
# lkst buf ls
# lkstbuf jump -b 1
# lkstbuf read -f /dev/null
```

3.3.3 ダンプ採取

以下の手順でダンプを採取する。

```
# echo 1 > /proc/sys/kernel/sysrq
# echo c > /proc/sysrq-trigger
```

3.3.4 LKST データの取得

採取されたダンプから、LKST のデータを収集する。

```
# cd /var/log/dump/0
# alicia -crash map.0 /usr/src/linux/vmlinux dump.0
alicia> $lkst = new Lkst
alicia> $lkst->collect(id=>1)
```

3.3.5 ランキュー情報

アナライザに「runqueue」、出力を「runqueue.txt」に指定する。

```
alicia> $lkst->analyze(analyzer=>'runqueue', id=>1, output=>'runqueue.txt')
```

```
Analyzing ...

cpuid,cpu,start,delta_value
CPU-0,runqueue,2005/08/31 14:28:29.834028747,2,0
CPU-0,runqueue,2005/08/31 14:28:29.834051105,1,2590
CPU-0,runqueue,2005/08/31 14:28:29.836046939,2,0
CPU-0,runqueue,2005/08/31 14:28:29.837789892,1,2814
CPU-0,runqueue,2005/08/31 14:28:29.838234673,2,0
CPU-0,runqueue,2005/08/31 14:28:29.838407861,1,2416
＜以下略＞
```

左から CPU 番号、アナライザ名、時刻、ランキューのサイズ
(PROCESS_CONTEXTSWITCH イベント発生時)、スイッチする前のプロセス ID。

3.3.6 システムコール情報

アナライザに「syscall」、表示形式に「stat」(統計情報)、出力ファイルに「syscall.txt」を指定する。

```
alicia>  $lkst->analyze(analyzer=>'syscall',  format=>'stat',  id=>1,
output=>'syscall.txt')
```

```
sysno,syscall_name,count,average,max,min
3,read,395,8794239,969339124,4792
4,write,215,605914,21632036,5444
5,open,236,31480,227400,12560
6,close,236,4421,39384,1124
7,waitpid,8,15182,32736,3100
11,execve,19,229510,1423456,6876
```

```
13,time,3,3434,4100,2108
20,getpid,4,1088,1272,936
33,access,13,14172,29056,7456
41,dup,4,4556,5528,4064
42,pipe,1,32396,32396,32396
45,brk,127,4918,30812,512
54,ioctl,5,5996,16072,1844
63,dup2,1,7004,7004,7004
64,getppid,2,1352,1560,1144
65,getpgrp,1,1664,1664,1664
77,getrusage,1,3920,3920,3920
85,readlink,1,117708,117708,117708
90,mmap,52,13411,26576,5832
91,munmap,20,35352,52772,15580
120,clone,3,496556,638324,413796
122,uname,6,3368,3960,3092
140,_llseek,21,2744,5768,1544
142,_newselect,7,390471303,2392212430,14244
174,rt_sigaction,39,2748,11432,1164
175,rt_sigprocmask,40,2228,5828,956
183,getcwd,2,11518,12396,10640
191,ugetrlimit,1,1640,1640,1640
192,mmap2,123,2464486,300575836,5856
195,stat64,41,18182,60168,4472
197,fstat64,218,2782,12828,1104
199,getuid32,2,982,992,972
200,getgid32,2,622,676,568
201,geteuid32,2,578,656,500
202,getegid32,2,518,524,512
221,fcntl64,2,3160,3640,2680
243,set_thread_area,5,3352,4192,2456
258,set_tid_address,1,948,948,948
```

左から、システムコール番号、システムコール名、実行回数、平均値、最大値、最小値(単位は、CPU サイクル数)。

4 考察

4.1 lcrash 対応の有効性

今回の lcrash 対応で、Alicia から lcrash のコマンドも実行できるようになった。これによって、今まで lcrash しか使ったことがないユーザでも、Alicia の機能を使うことができるようになった。

lcrash には、crash に比べて以下の優位性があり、これらの優位性も Alicia に取り込まれたこととなる。

- デバック情報を含んだカーネルイメージがいらない
  crash は、シンボル名や構造体情報の取得のため、実行にはデバッグカーネルを必要とするが、lcrash はダンプ解析に独自の仕様を持っており、デバッグカーネルを必要とせず単体で実行できる。そのため、LKCD で採取されたダンプの解析をすぐに始められる。
- 異なるアーキテクチャのダンプを解析できる
  crash は、異なるアーキテクチャで採取されたダンプを解析するためには、ダンプを採取した環境を必要とする。　　例えば、IA-32 アーキテクチャの環境でコンパイルされた lcrash で IA-64 アーキテクチャで採取されたダンプを解析できる。

また、3.1.6 項の比較から、全体的に lcrash の動作が早いことが分かる。特に kernel 関数に関しては数十倍の処理速度の違いが出ている。これは、kernel 関数の処理が、crash 内でラッピングされている GDB の機能を利用しているためと考える。

今回の lcrash 対応により、Alicia API の実行速度が crash に比べ非常に早くなることがわかった。lcrash に対応したダンプは crash、lcrash 両方で開くことが可能であるが、大量の処理を実行するような場合には、lcrash モードで Alicia を実行したほうが効率的といえる。

4.2 初期 LDAS の有効性

初期 LDAS のメリットは、サマリ的なアウトプットが容易に得られるという点と、決められたスクリプトを実行することにより、カーネルやダンプに関する知識の無い人でも簡単に、ある程度の情報を編集して目に見える形にできるという点である。

例えば、遠隔地で発生した障害で、採取された資料をメールなどで解析部隊に送付する必要がある場合、生のダンプファイルではサイズが大きすぎるためメールに添付して送ることができない場合が多い。それに対して初期 LDAS で得られたファイルのサイズは 200～300k バイトなので、メールに添付して解析部隊に送付することが可能である(図 4.2-1)。これにより、障害の早期解析に着手し、いち早く原因を切り分けることが可能となる。

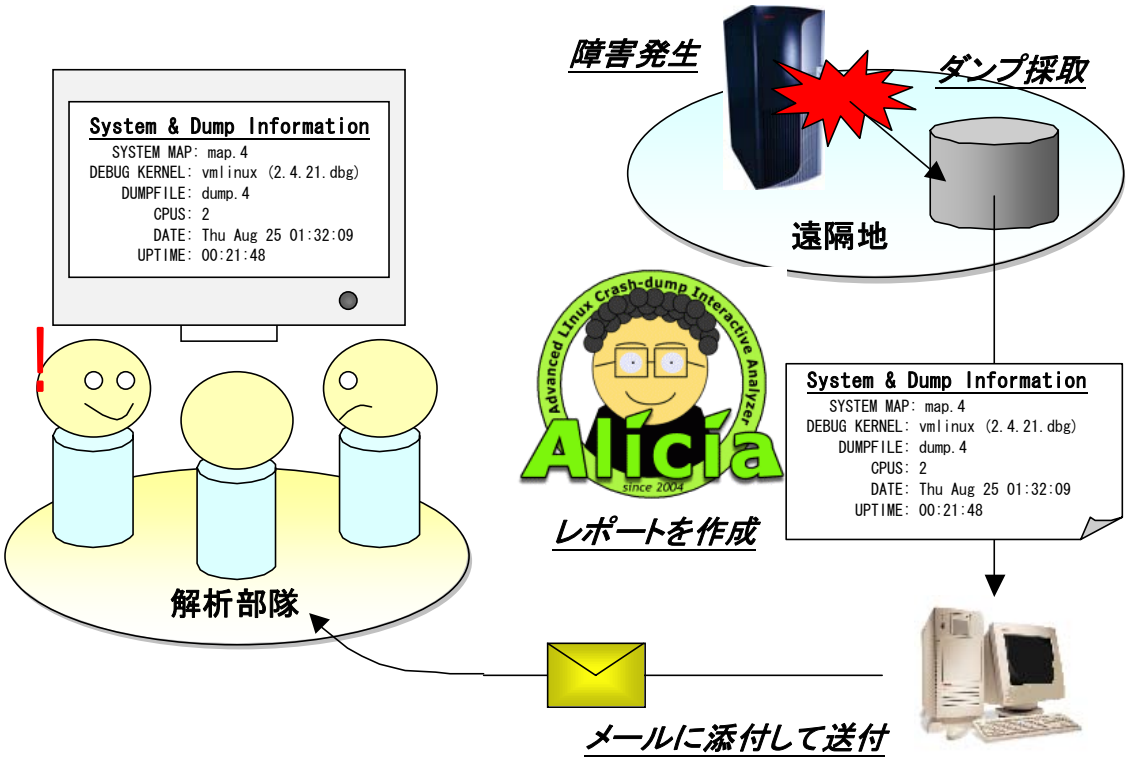

図 4.2-1 初期 LDAS 利用方法

また，障害の原因が既知の問題であれば、初期 LDAS のリストの情報から、すぐに既知の問題に該当するか判別できる場合もある。この時点で既知の問題と判別できれば，早急な対応が可能となる。

お客様の Linux システムに Alicia および初期 LDAS を適用していたなら、上記のようなサポートが可能となる。障害の原因が早期に切り分けられれば、問題点が修正されるまでの間にも、障害の再発生を防止したり、障害の影響を低減したり局所化するといったことを実現できる。これは、ミッションクリティカルなシステムをサポートしていく上で極めて重要なことである。

このように、一時解析資料を自動的に採取する初期 LDAS の機能は障害解析の迅速化に大きな効果を発揮することが期待できる。

次に、初期 LDAS を使用せずに同様の資料を採取する場合の手間を比較してみる。

初期 LDAS は、一つのコマンドで 13 種類の情報を含むレポートファイルと、7 種類の詳細情報ファイルを出力する。このうち約 7 割は crash コマンドをそのまま使用することにより表示可能であるが、残りの 3 割は、その情報を保持している変数や構造体を調べたり、繰り返し構造体等をたどる必要のある情報である。

例えば、プロック情報（3.2.15 項参照）などはソースコードの解析を行い、どの変数に情報が保持されているかなどを調査するところからはじめる必要がある。さらに、複数の変数や構造体やメンバを組み合わせて計算を行ったりする必要があるものもある。初期 LDAS を使用しなければ、いちいち構造体のオフセット位置を調べたりしながらメンバをたどっていくことになる。一つの例として、/proc/sys/fs/file-max を crash を使用して調べるステップを以下に示す。（既にソースからどの変数を調べるかは調査済みとする）

```
# crash  map.0 /usr/src/linux/vmlinux dump.0

crash> whatis files_stat
struct files_stat_struct files_stat;

crash> struct files_stat_struct
struct files_stat_struct {
    int nr_files;
    int nr_free_files;
    int max_files;
}
SIZE: 12

crash> rd files_stat 3
c03730e0:  00000154 00000045 00020000
```

上記の例で、例えば調査時間が数分かかったとして、その後上記 3 つのコマンドを入力して初めて求める情報が採取可能となる。

このように、単純に複数のコマンドを入力するだけではなく、調査や判断などを必要とする情報も多く含まれている。初期 LDAS を使用すると調査時間が不要であり、たった 1 つのコマンドで多くの有効な情報が採取可能となる。

今回、Alicia 本体の開発時からその必要性を問われていた「初期 LDAS」の開発を実施したが、開発作業に伴って得られた副産物的なメリットも幾つかあった。Linux の客先サポートを主とし、カーネルのソースコード読みやダンプ解析をしたことがないメンバを LDAS 開発に参加させることで、Linux のソースコード読みへの入り口を開き、サポート技術の底上げに貢献することができた。逆の見方をすれば、ダンプ解析に不慣れな人間でも LDAS を組むことが出来ることの実証にもなった。

また、初期 LDAS の設計段階では、「どのような情報を表示すれば良いか」という点に関して、Linux だけではなく、異なるプラットフォームの OS で、それぞれダンプ解析に関わってきたメンバの意見を集め、ダンプ解析時に表示されるべき情報をまとめた。それ

らが Linux ではどういう情報に相当するかを調べ、更に Linux 固有の情報を追加していった。これらの作業はまさに「Linux のダンプ解析ではまずどのような情報が必要となるか」という課題を解決することに他ならない。そして、各情報の重要度や、各 LDAS 開発の難易度、工数、などを検討して現時点の仕様が決められた。

初期 LDAS の内容をスタディすることにより、LDAS 作成の一般的ノウハウと Linux カーネルで処理される各種のデータ構造に関するノウハウを読み取ることが可能である。初期 LDAS はカーネルスキルを習得するための教材となりえる点でも大きな存在価値を持つと言えるだろう。

4.3 LKST 連携の有効性

Alicia から LKST が採取したデータを参照できるようになったことで、ダンプデータから、過去に遡ってどんなイベントが起きていたかを見ることができるようになった。これは、原因追求のための手段が増えたことを意味する。特に、ハングアップ等の障害の原因が停止直前ではなく、少し前の事象に起因しているような場合に有効であると考える。
また、3.3 節で示したように、ダンプ採取直前のシステムコールの統計データを編集することができ、システムがどのような状態にあるかを推測するのに役に立つことが分かる。現在の Alicia では、統計データのアナライザとしてシステムコール情報の解析機能しか用意できていないが、将来はこのインフラを利用することにより LKST が採取している様々な情報を解析できるようになるだろう。

LKST データの参照は、カーネルに LKST のパッチが適用されており、さらに LKST のデータ採取が指示されている場合にのみ発生する。このため、客先での実運用中の障害に対してダンプ上に LKST データが残されているケースは稀であり、実例で LKST 連携の有効性を示すのは難しい。しかし、再現テストを行う際に LKST データを採取できれば、Alicia による LKST 連携の効果を実感できるだろう。なぜならば、ダンプ解析においてトレース情報は原因を特定するための決め手となり得る情報だからである。

前節では初期 LDAS の教育的効果について説明したが、LKST のデータとダンプのデータを教育的な観点で利用できないかについても検討を行った。例題として、システムを停止に導いたプロセスの軌跡を見ることでシステムに何が起こっているかを見てみる。まずは、LKST のイベントレコードから指定したプロセスのイベントをリストアップする LDAS を作成する。

```
sub eventlog {
    my $lkst = shift || die "Usage: eventlog(lkst instance, pid)";
    my $pid = shift || 1;
    my $recs = $lkst->get_records;
    my $ana = $lkst->get_analyzer(analyzer=>'runqueue');
```

```
    my @events =
        $ana->filter_by_attribute({log_pid=>$pid},
        $ana->sort_by_recid(@$recs));

    print "  address  event     arg1       arg2      arg3      arg4¥n";
    print $_->myaddr, " ",
            sprintf("0x%03x %10x %10x %10x %10x", $_->log_event_type,
            $_->log_arg1, $_->log_arg2, $_->log_arg3, $_->log_arg4), "¥n"
            for @events;
}
1;
```

この LDAS に対して最後にアクティブであったプロセスの PID を引数として指定する。

```
alicia> task | grep PID
PID: 29149  TASK: c3d9e000  CPU: 0   COMMAND: "bash"
alicia> $lkst = new Lkst
alicia> $lkst->collect(id=>1)    # CPU0のみ
alicia> eventlog($lkst, 29149)
```

```
  address  event     arg1       arg2      arg3      arg4
0xc89d76c0 0x093    c029d92d   c01b5007         0         0
0xc89d7700 0x002    c1aaa000          1         0         0
0xc89d7740 0x093    c0299704   c01b5028         0         0
0xc89d7780 0x020    c011a8b0          0  c01ecabb         0
0xc89d77c0 0x093    c0296f8c   c011aaae         0         0
0xc89d7800 0x093    c0296d1c   c011aac5         0         0
＜中略＞
0xc89d89c0 0x093    c0295fb0   c010c0a3         0
0xc89d8a00 0x093    c0295fb0   c010c1c8         0
0xc89d8a40 0x092    c0295782   c010c1fd         0         0
0xc89d8a80 0x093    c029727d   c0120b07         0         0
0xc89d8ac0 0x002    c6d04000          3         0
0xc89d8b00 0x001    c3d9e000   c1aaa000         2
0xc8a0ff40 0x001    c3d9e000   c5f26000         2         0
0xc8a1b880 0x001    c3d9e000   c5650000         2         0
0xc8a1cb80 0x001    c3d9e000   c03c8000         2         0
0xc8a1d0c0 0x001    c3d9e000   c03c8000         2         0
```

① イベントタイプ 0x092
die 関数から panic("Fatal exception")が呼ばれた。

② イベントタイプ 0x093
panic 関数から printk("Kernel panic: xxx)が呼ばれた。

```
0xc8a1de80 0x001    c3d9e000    c03c8000          2          0
0xc8a1e5c0 0x001    c3d9e000    c03c8000          2          0
0xc8a1e840 0x001    c3d9e000    c03c8000          2
0xc8a1ef00 0x001    c3d9e000    c5f26000          2
0xc8a20e00 0x001    c3d9e000    c5f26000          2
0xc8a24b00 0x093    c02a7854    c011a89b          0
0xc8a24b40 0x093    c0295fb0    c0120b56          0          0
```

③ イベントタイプ 0x001
このプロセスから swapper プロセスへの切り替えが行われた。

上記の実行結果でピックアップしたイベントを詳細に見てみる。

① イベントタイプ 0x092

・イベントタイプ 0x092 の定義 (include/linux/lkst_etype.h)

```
LKST_ETYPE_DEF(0x092, NORMAL, O_PANIC, "panic", ¥
        "address of argument", ¥
        "call address", ¥
        NULL, ¥
        NULL)
```

・イベントタイプ 0x092 の記録 (kernel/panic.c)

```
NORET_TYPE void panic(const char * fmt, ...)
{

(中略)

    LKST_HOOK(LKST_ETYPE_O_PANIC,
                    LKST_ARGP(fmt),
                    LKST_ARGP(__builtin_return_address(0)),
                    LKST_ARG(0),
                    LKST_ARG(0));

(以下、省略)
```

上記より、イベントタイプ 0x092 は、panic 関数で LKST_HOOK マクロが、O_PANIC のイベントを記録したものであることが分かる。また、LKST_HOOK により、panic 関数が呼ばれたときの引数のアドレス(arg1)と、panic 関数を呼んだアドレス+1 命令(arg2)が記録されている。arg1、arg2 の内容を見てみる。

・arg1 (c0295782)

```
alicia> rd c0295782 4
```

arg1
引数で指定されたのは、文字列(Fatal exception.)のアドレスである。

```
c0295782:  61746146 7865206c 74706563 006e6f69   Fatal exception.
```

・arg2 (c010c1fd)

```
alicia> dis 0xc010c1f1 5
0xc010c1f1 <die+129>:   movl   $0xc0295782,(%esp)
0xc010c1f8 <die+136>:   call   0xc0120a70 <panic>
0xc010c1fd <die+141>:   lea    0x0(%esi),%esi
0xc010c200 <die+144>:   movl   $0x0,(%esp)
0xc010c207 <die+151>:   call   0xc011a850 <bust_spinlocks>
```

arg2
die 関数の先頭から 136 バイト目の call 命令より、panic 関数が呼ばれた。arg2 には、戻り番地が記録されている。

② イベントタイプ 0x093

・イベントタイプ 0x093 の定義 (include/linux/lkst_etype.h)

```
LKST_ETYPE_DEF(0x093, NORMAL, O_PRINTK, "printk", ¥
        "address of argument", ¥
        "call address", ¥
        NULL, ¥
        NULL)
```

・イベントタイプ 0x093 の記録 (kernel/panic.c)

```
asmlinkage int printk(const char *fmt, ...)
{

(中略)

        LKST_HOOK(LKST_ETYPE_O_PRINTK,
                        LKST_ARGP(fmt),
                        LKST_ARGP(__builtin_return_address(0)),
                        LKST_ARG(0),
                        LKST_ARG(0));

(以下、省略)
```

上記より、イベントタイプ 0x093 は、printk 関数で LKST_HOOK マクロが、O_PRINTK のイベントを記録したものであることが分かる。また、LKST_HOOK により、printk 関数が呼ばれたときの引数のアドレス(arg1)と、printk 関数を呼んだアドレス+1 命令(arg2)が記録されている。arg1、arg2 の内容を見てみる。

・arg1 (c029727d)

```
alicia> rd c029727d 4
```

```
alicia> rd c029727d 8
c029727d:  4b3e303c 656e7265 6170206c 3a63696e   <0>Kernel panic:
c029728d:  0a732520 6e617000 632e6369 3e303c00    %s..panic.c.<0>
```

・arg2 (c0120b07)

```
alicia> dis -r c0120b07 5
0xc0120aee <LKST_ETYPE_O_PANIC_HEADER_hook+44>: ca
0xc0120af3 <LKST_ETYPE_O_PANIC_HEADER_hook+49>: mo
0xc0120afb <LKST_ETYPE_O_PANIC_HEADER_hook+57>: movl   $0xc029727d,(%esp)
0xc0120b02 <LKST_ETYPE_O_PANIC_HEADER_hook+64>: call   0xc0121440 <printk>
0xc0120b07 <LKST_ETYPE_O_PANIC_HEADER_hook+69>: mov    0xc03c1818,%esi
```

arg2

0xc0120b02 の call 命令より、printk 関数が呼ばれた。dis コマンドは、関数名を直近の外部参照名を表示するため、panic 関数内ではあるが、関数名を LKST_ETYPE_O_PANIC_HEADER_hookとしている。

③ イベントタイプ 0x001

・イベントタイプ 0x001 の定義 (include/linux/lkst_etype.h)

```
LKST_ETYPE_DEF(0x001, NORMAL, PROCESS_CONTEXTSWITCH, "context_switch", ¥
        "pointer to task_struct prev ", ¥
        "pointer to task_struct next ", ¥
        "process state", ¥
        "process count")
```

・イベントタイプ 0x001 の記録 (kernel/sched.c)

```
static inline task_t * context_switch(runqueue_t *rq, task_t *prev, task_t *next)
{

(中略)

        LKST_HOOK(LKST_ETYPE_PROCESS_CONTEXTSWITCH,
                        LKST_ARGP(prev), LKST_ARGP(next),
                        LKST_ARG(prev->state),
                        LKST_ARG(0));

(以下、省略)
```

上記より、イベントタイプ 0x001 は、context_switch 関数で LKST_HOOK マクロが、PROCESS_CONTEXTSWITCH のイベントを記録したものであることが分かる。また、LKST_HOOK により、プロセス切り替え前の task_struct 構造体のアドレス(arg1)と、プロセス切り替え後の task_struct 構造体のアドレス(arg2)が記録されている。arg1、arg2 の内容を見てみる。

・arg1 (c3d9e000)

```
alicia> task c3d9e000 | head -2
PID: 29149  TASK: c3d9e000  CPU: 0   COMMAND: "bash"
struct task_struct {
```

・arg2 (c03c8000)

```
alicia> task c03c8000 | head -2
PID: 0      TASK: c03c8000  CPU: 0   COMMAND: "swapper"
struct task_struct {
```

arg2
プロセスが swapper に切り替えられた。

このように、ダンプ採取時にシステムに存在するプロセスのイベントとそのイベント発生時に保存されるデータから、カーネル内でのプロセスの遷移を観測することができる。

4.4 Alicia 開発規模

開発規模を示す一指標として、今回のAlicia開発における開発ステップ数を表 4.4-1に示す。

表 4.4-1 Alicia の開発規模

項番	開発項目	開発言語	ステップ数
1	本体	Perl	3670
2	lcrash 対応	Perl	731
3	初期 LDAS	Perl	1193
4	LKST 連携	Perl	1152
5	その他(新規 LDAS、本体機能拡張、バグ FIX)	Perl	1401

5 今後の課題と計画

5.1 今後の課題

初期 LDAS で表示する情報については、現在の初期 LDAS が数多くのダンプ解析において利用され、情報の過不足に関する経験値を集約することによって、多くの改良を積み重ねていく必要があるだろう。

更に、障害時のためにカーネル内部で採取している情報の足りなさを検討し、LKST と同じような立場から、各種の内部トレースや記録を残しておけるようなカーネルへの機能追加までも視野に入れて、ダンプ情報の充実を図りたい。

これらの改良を重ねていくには、カーネルに対する深い知識が必要となるが、カーネル・ソースのスタディもさることながら、ショート・プログラム的にあれこれと LDAS を組んでみることが、改良を促進する上でも有効だと考える。

また、OS 層障害解析ツールの調査から、Solaris の MDB コマンド群との差分が明らかとなった。MDB には、crash/lcrash で提供されていないコマンド機能が多数あり、その辺りを補完するような LDAS の作成も課題の一つである。

5.2 計画

LDAS を世界中で共有する場合、LDAS のソースコードを共有できるだけではなく、用途やカーネルのシンボルなどで検索できる仕組みが必要である。今回、初期 LDAS により、広範囲にわたるダンプ情報の収集を実現しており、まずはこれを例題に共有・検索できるシステムを作る計画を持っている。

また、現在、LDAS の作成は、Perl 言語で記述しなければならないが、他の言語(C 言語、Ruby、Python)で記述するためのバインディングを作成することで、Alicia 利用者（ダンプ解析者）の裾野を広げていきたい。

6 おわりに

Alicia は、「独立行政法人 情報処理推進機構 オープンソースソフトウェア活用基盤整備事業」に係る委託業務の一環として開発したものである。

日本 OSS 推進フォーラム URL：
http://www.ipa.go.jp/software/open/forum/development/index.html

参考文献

[1] mkdump (Mini Kernel Dump) http://mkdump.sourceforge.net/ (2005)

[2] diskdump http://sourceforge.net/projects/lkdump/ (2005)

[3] Linux Tough dump http://www.hitachi.co.jp/Prod/comp/linux/products/solution.html (2005)

[4] kdump http://lse.sourceforge.net/kdump/ (2005)

[5] LKCD (Linux Kernel Crash Dump) http://lkcd.sourceforge.net/ (2005)

[6] crash (Linux crash analysis utility) http://people.redhat.com/anderson/crash_whitepaper/ (2003)

[7] LKST (Linux Kernel State Tracer) http://lkst.sourceforge.net/ (2005)

[8] Solaris MDB (Solaris Modular Debugger Guide) http://docs.sun.com/app/docs/doc/816-3983/ (2004)

7 付録

7.1 Alicia の実行方法

ここに、MIRACLE LINUX V3.0 SP1(以下、MIRACLE LINUX V3.0)で、Alicia を起動する方法について簡単に記しておく。

7.1.1 Alicia の実行に必要な Perl モジュールの導入

以下の手順に従い、CPAN からダウンロード後、インストールする。CPAN とは、Perl 関連のコードの総合アーカイブサイトである。

```
# perl -MCPAN -e shell
cpan> install Term::ReadKey
cpan> install Term::ReadLine::Perl
```

7.1.2 crash、lcrash の導入

MIRACLE LINUX V3.0SP1 では、crash、lcrash とも導入済みであるため、ここでは導入方法については割愛する。それぞれは、以下の URL からダウンロードできる。

```
crash: ftp://people.redhat.com/anderson/
lcrash: http://lkcd.sourceforge.net/
```

7.1.3 Alicia の導入

Alicia は、以下の URL からダウンロードできる。

http://sourceforge.net/projects/alicia/

lcrash 対応版であるバージョン 1.1.0 をダウンロードする。以下の手順でインストールする。

```
# tar zxvf Alicia-1.1.0.tar.gz
# cd Alicia-1.1.0
# make
# make install
```

7.1.4 Alicia の設定

インストール後に作成される /etc/alicia.conf に crash、lcrash のコマンドパスを設定する。

デフォルトでは、以下の設定になっている。

```
Crash = /usr/bin/crash
Lcrash = /sbin/lcrash
```

7.1.5 ダンプ解析のための準備

ダンプを解析するためには、解析するためのダンプはもちろん、シンボル情報などの解決に必要なカーネルイメージ(vmlinux)が必要である。これらは、MIRACLE LINUX V3.0 では簡単に作成することができる。

・ デバッグ情報を含むカーネルイメージの作成

まず、カーネルのソースコードを/usr/src/linux に展開し、そのディレクトリに移動する。そして、Make ファイルの CFLAGS に g オプションを追加し、make するだけでよい。

1. Makefile を編集する

```
# vi Makefile
コンパイルフラグCFLAGSにgオプションを追加する
CFLAGS        := -g -Wall -Wstrict-prototypes -Who-trigraphs
```

2. make を実施する

```
# make clean
# make
/usr/src/linuxにvmlinuxが作成される。
```

・ ダンプを採取する

MIRACLE LINUX V3.0 では、デフォルトで LKCD(ダンプ採取ツールの一つ)がインストールされているので、以下の手順を実施すれば、/var/log/dump にダンプが採取される。ただし、ダンプは一度 Swap パーティションに保存されるので、Swap パーティションのサイズをメモリーサイズ以上に取っておく必要がある。

1. ダンプ採取を可能にする

```
# echo 1 > /proc/sys/kernel/sysrq
```

2. ダンプを採取する

```
# echo c > /proc/sysrq-trigger
```

/var/log/dump/n/ (n はダンプ採取毎のカウントアップされるシーケンス番号)にダンプが保存される。

7.1.6 Alicia の起動

上記の手順によって作成されたダンプを開いてみる。以下の書式で起動する。

$ alicia [mode] [その mode に従った引数]
mode は、-crash または -lcrash を指定する。
以下は、crash を利用してダンプを開くための例である。

```
$ cd /var/log/dump/0
$ alicia -crash map.0 /usr/src/linux/vmlinux dump.0
                        :
alicia>
```

alicia というプロンプトが表示されれば、Alicia を使ってダンプ解析が可能である。

7.1.7 API の紹介

(1) pass_through('既存ダンプ解析ツールのコマンド')
この関数は既に登場している。crash コマンドの結果を文字列で返してくれる関数である。以下は、crash の ps コマンドから task_struct 構造体のアドレスの一覧を取得する例である。

```
alicia> @pss = map {(split /¥s+/)[4]} split(/¥n/, pass_through('ps'))
```

(2) get_addr('カーネルのシンボル名')
カーネルのシンボル名のアドレスを返す。

```
alicia> $addr = get_addr('saved_command_line') # $addrは(3)の例で使用
```

(3) get_mem('アドレス', ワード数)
指定されたアドレスからワード数分のメモリの内容を返す。

```
alicia> @outs = get_mem($addr, 256/4)
alicia> print join '', map { pack("V", hex($_)) } @outs # ASCIIに変換
ro root=LABEL=/ hda=ide-scsi
```

(4) kernel('アドレス','構造体名','メンバ変数名','型')

指定されたカーネル構造体のメンバ変数を返す。

```
alicia> $utsname = get_addr('system_utsname');
alicia> print kernel($utsname,  'new_utsname', 'release', 'char *');
2.4.21-9.30AXsmp
```