2004 年度

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」

ダンプデータ解析ツール(Alicia)の評価と考察

独立行政法人 情報処理推進機構

商標表記

- 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。
- Windows は、Microsoft Corp.の登録商標です。
- UNIX は、X/OPEN Company Ltd.が独占的な許諾権を持つ、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

この章では，Alicia の開発経緯と試用結果，および試用結果に基づく考察および課題について報告する。評価対象は Alicia のバージョン 1.0.0 である。

1.1 背景

近年、Linux のミッションクリティカル領域への進出が目立ってきたが、トラブルの解析にダンプが用いられる機会が少ない。システムハング等が発生しても、リブート後に現象が再現しなければ原因追求をしない、または、一部の有識者の経験に頼った解析を行っているのが現状である。
しかし、トラブルの早期解析に、ダンプデータを利用したいというニーズは、何年も前からあり、Linux のミッションクリティカル領域への進出とともに、ダンプ採取に対する意識が高まりつつある。

1.2 ダンプ解析とは

採取されるダンプデータには、ダンプの情報を含むヘッダー部分と、システムが停止した瞬間のメモリのデータ部分がある。そのデータ部分を Human Readable なデータとして可視化するには、それを翻訳するツールを必要とする。このように必要な情報を可視化することを「ダンプを編集する」と呼ぶ。
そして、ある一定の形式でダンプを編集するツールをいくつか組み合わせて、対話的に利用し、試行錯誤的に“当たり”を付けながら段階的に解析を進めていくことを、「ダンプを解析する」と呼ぶ。これは、ダンプの中にある情報のジャングルを、いろいろな手掛かりを頼りに探検しながら解(システムで生じたトラブルの原因やシステムが最適な状態で稼動しているかどうかの確認)に向かって突き進んでいくイメージである。
この解にいち早く近づくために利用できるツールを、ここでは「ダンプ解析ツール」と呼ぶことにする。

2 開発経緯

当 WG がダンプ解析ツールに取り組んだのは、Linux のミッションクリティカル領域への階段を 1 段上げるためである。ここでは、そのダンプ解析ツールの作成を始めた動機について述べていきたい。

2.1 現状分析

ダンプ解析の事例が少ないのは、Linux がミッションクリティカル領域とは異なるところから登場してきたため、ダンプを採取するという文化が育っていないことも理由の一つとして挙げられるだろう。Linux カーネルのデザインの中にダンプ採取に対する意識がないこともそれを表している。ダンプ採取への取り組みを行っているのは、ミッションクリティカル領域での業務経験がある大手 SIer やディストリビュータが主である。
その理由として、ダンプを採取しても解析するツールが充実していなかったり、解析する人や SIer がいなかったりすることが大きな理由であると考える。

2.2 ミッションクリティカルシステム

ミッションクリティカルシステムの目指すところは、業務を連続的に稼動させていなければならないシステムで、長期に渡って停止することが許されないシステムを構築することである。つまり、安定とトラブルの早期切り分け・解決が命題となっている。
これまで、Linux は、「安定」という部分に関して多大な労力が注ぎ込まれてきたが、もし障害が起きた場合にどうするのかという問いには答えられていない。
ミッションクリティカルな業務で使用されているシステムでは、例えば、メインフレーム、UNIX では当たり前のように、また、最近は Windows サーバにおいても、障害時のダンプ採取は必須となっている。

2.3 ダンプ採取

ダンプを採取してもらうためには、Linux のダンプに関する環境の向上が必要である。ダンプ採取についても様々な問題が提起されており、何年か前から活動している LKCD、Netdump というものから、最近では diskdump、mkexec なども立ち上がり、現状の問題点の克服について様々な解決方法を提示している。ここでは、現状のダンプ編集ツールについて紹介をしていく。

2.4 ダンプ編集ツール

2.1 の現状分析で、Linux のダンプ解析ツールが充実していないと述べた。確かに、現段階では、ダンプを「解析」するツールは存在しないといえる。ダンプを編集するツールはあっても、

ダンプの解析を補助するツール、環境が存在しないからである。そのため、我々は、今回、ダンプ解析ツールの第一段階として、ダンプ解析の環境を整えることを目標とした。
さて、その前に、現状のダンプ編集ツールについて簡単に説明しておこう。ここでは、ダンプ採取でも古くから活動をしている LKCD と Netdump で採取されたダンプの編集に利用されている lcrash と crash というツールについて紹介する。

2.4.1 crash

crash は、UNIX の crash ツールのインターフェースをベースに作成されたツールで、あるディストリビュータやデベロッパーが、netdump フォーマットに対応したダンプ編集ツールとして開発している。LKCD のダンプフォーマットにも対応している。
UNIX のツールを参考にしているため、UNIX のダンプ解析を行っていたユーザには親しみやすく、また GDB をラッピングしているため、GDB のコマンドもいくつか実行できることが、大きな特徴である。また、コマンドラインには GNU Readline を採用していたり、ページャーなども外部のコマンドを利用していたりと、lcrash のように独自で全てを抱え込むという仕様にはなっていない。
また、ダンプ編集のスピードアップの考慮や、動的ライブラリによる拡張コマンドの追加機能も備えている。

2.4.2 lcrash

lcrash は、LKCD で採取されたダンプを編集するために作成されたツールで、数社の大手デベロッパーによって開発が進められている。しばらく更新がなかったのだが、ここ半年くらいで活動度があがり、新バージョンもリリースされ、LKCD ダンプフォーマット以外で Netdump フォーマットにも対応をするようになったと記述している。
大きな特徴は、ダンプ編集にあたって、独自の仕様を持っており、カーネルのシンボル情報を得るのにデバッグオプション付きでコンパイルされたカーネルを必要としない、また、異なるアーキテクチャ上で採取されたダンプにも対応できるようになっているところである。また、独自で sial というライブラリを抱え、C 言語に非常に似た記述方法で作成したスクリプトをインタプリタとして動かせるエンジンを持っている。この sial は C 言語と 100%互換性があるわけではないので、若干癖はあるが、カーネルのヘッダーファイルをインクルードすることにより、カーネルの変数や構造体にアクセスできるなど、大変有効なユーティリティである。

2.5 現状の課題と改善が必要な点

ここまで、crash と lcrash というツールを紹介してきたが、両者とも素晴らしいツールでありながら、どちらも独自の道を進んでいるため、悪いところは補足し、良いところは取り入れるというような歩みよりが難しくなっているように思える。
そこで、我々は、両者を統合的に使えるツール、また両者に不足している部分を補えるツールを作成していくことを決定した。作成するにあたって、新規にもう一つのダンプ編集ツールを作成するのではなく、両者をラッピングすることによって、両者の良い部分を生かしたツールを作成することが一番の近道であると考えた。

表 2.5-1 に課題をいくつか列挙している。4 章で、それを Alicia がどのように解決していったかを見ていく。

表 2.5-1 現状の課題

課題	内容
コマンドライン	crash では、GNU Readline を一部採用しており、馴染みやすい。lcrash は独自のコマンドライン編集である。
インターフェース	lcrash と crash では、コマンド体系が異なる。また、lcrash は netdump フォーマットのダンプには対応していなかった（最新バージョンでは対応）。 また、それぞれのコマンドの表示形式を変更するだけでも、ソースコードを修正しコンパイルしなおすか、拡張コマンドを作成する必要があった。
簡易解析スクリプト(インタプリタ)	lcrash では、sial 言語で記述可能な環境を用意している。カーネルの変数、構造体にアクセスできるのは便利だが、簡易にすぐに記述するのは難しい。crash では拡張コマンドとして C 言語で記述する動的ライブラリを用いる方法しかない。
解析の共有化	ダンプ解析者によるそれぞれのツールのコマンドを使った解析の手順がライブラリ化されていない。

3 Alicia とは

3.1 概要

Alicia は、Linux のカーネルダンプを解析するための快適な環境や、Perl で書かれた新たな解析コマンドを動的に追加する機能を提供するツールである。これらの機能は、既存ダンプ編集ツールである crash(lcrash にも対応予定)を Perl のインターフェースによりラッピングし、それらのコマンドに Alicia 独自の関数を加えることによって実現している。

3.2 ダンプ解析環境での Alicia の位置付け

現在のダンプ解析環境に Alicia が加わった場合、ダンプ解析環境は、図 3.2-1 のようになる。この図の右側の網掛け(青色)部分が、Alicia 本体と Alicia 上で利用できるダンプ解析スクリプトである。この解析スクリプトのことを LDAS: Linux Dump Analysis Script と呼ぶ。

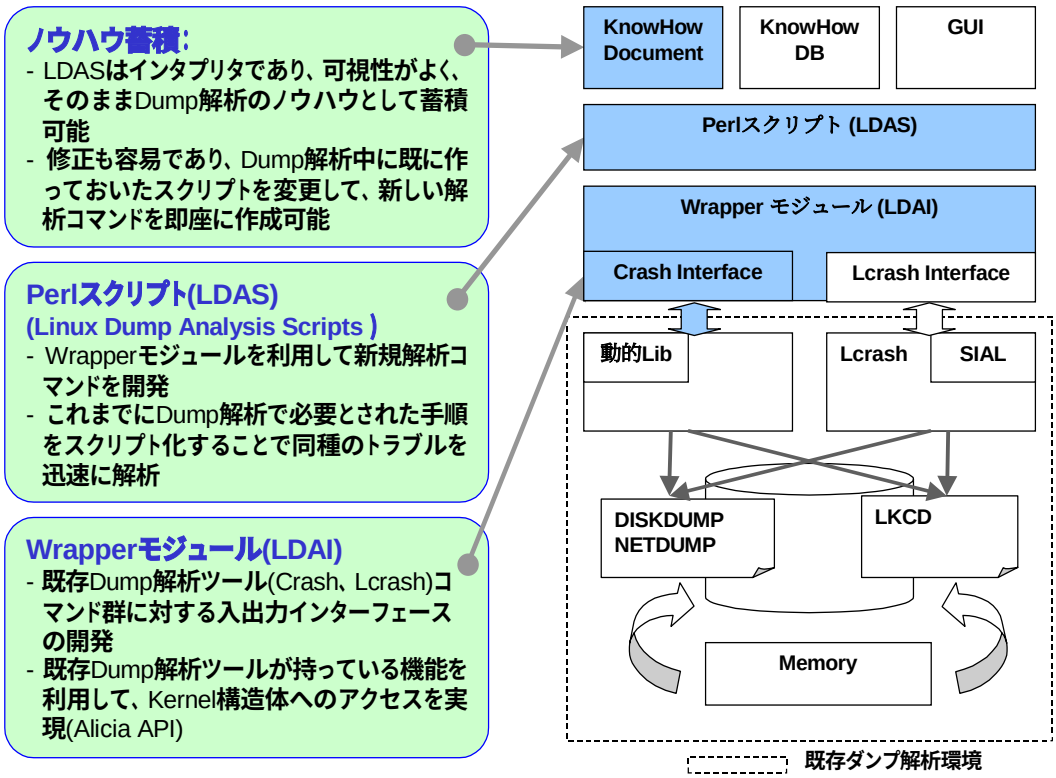

図3.2-1　ダンプ解析環境におけるAliciaの位置付け

3.3 Alicia の機能

Alicia は、大きく分けて 4 つの機能を持っている。ダンプ解析者は、それらの機能を利用して、ダンプ解析環境の向上、解析時間の短縮を手に入れることができる。以下に、それぞれの機能について、簡単に説明する。

(1) Wrapper 機能
既存ダンプ解析ツールをラッピングする。単にラッピングして、Alicia 独自のコマンド体系にしてしまうのはなく、既存ツールが持っているコマンドセットもそのまま使用できるようにしている(一部のコマンドはサポートしない)。

(2) Interactive Perl
対話的に解析を進めることができる。そのときに、Perl の構文が利用できるため、Alicia に入力したコマンドの結果を変数に格納し、それをさらに別のコマンドへの入力にするという使い方が可能である。

(3) Scripting 機能
作成しておいた LDAS をロードし、コマンドとして対話的に実行することができる。また、その LDAS を関数として利用することで、別の LDAS の中に組み込むことも可能である。

(4) Alicia API
スクリプトの中で利用しやすいいくつかの独自の関数(コマンド)を持つ。それらは、ラッピングするツールに依存することなく同じ結果を返す。これにより Alicia API を利用した LDAS もラッピングするツールに依存しない。
以下は、代表的な API である。

a) kernel('*address*', '*structure*', '*member*[.*member*]'[, '*type*'])
指定されたカーネルの構造体のメンバを得る。

b) get_mem('*address*'[, *number*])
仮想アドレスの内容を返す。

c) get_addr('kernel_variable')
カーネルのシンボルを仮想アドレスに変換する。

d) get_value('kernel_variable')
シンボルが指すアドレスの内容を返す。

3.4 Alicia の操作

Alicia のオペレーション概要を図 3.3-1 に示す。以下に、図中で番号付けされた各アクティビティについて順に説明する。

(1) インタラクティブに標準コマンドでダンプ解析をする。
①使用者は Wrapper モジュール(LDAI)を利用した Alicia プログラムを起動する。

②使用者は LDAI に登録してある標準ダンプ解析コマンドを実行する。
③LDAI は入力されたコマンドを Crash のコマンド形式に変換し、コマンドを Crash に向けて発行する。
④LDAI は Crash コマンドの結果を解析し要求された出力形式で使用者に結果を返す。
⑤使用者はその結果(出力データ)を変数に格納する。
⑥使用者はその結果を利用して別の標準コマンドを実行、または、その結果を出力する。

(2) インタラクティブに独自関数を作成し、ダンプ解析を行なう。
①を実行する。
⑦独自関数をその場で作成する。
⑧独自関数を実行する。
③～⑥が実行される。

(3) 独自の解析スクリプト(LDAS)を利用して、ダンプ解析を行なう。
①を実行する。
⑨使用者は用意しておいた LDAS を読み込ませ起動する。
⑩LDAI は、そのスクリプトを実行する。
③～⑥が実行される。

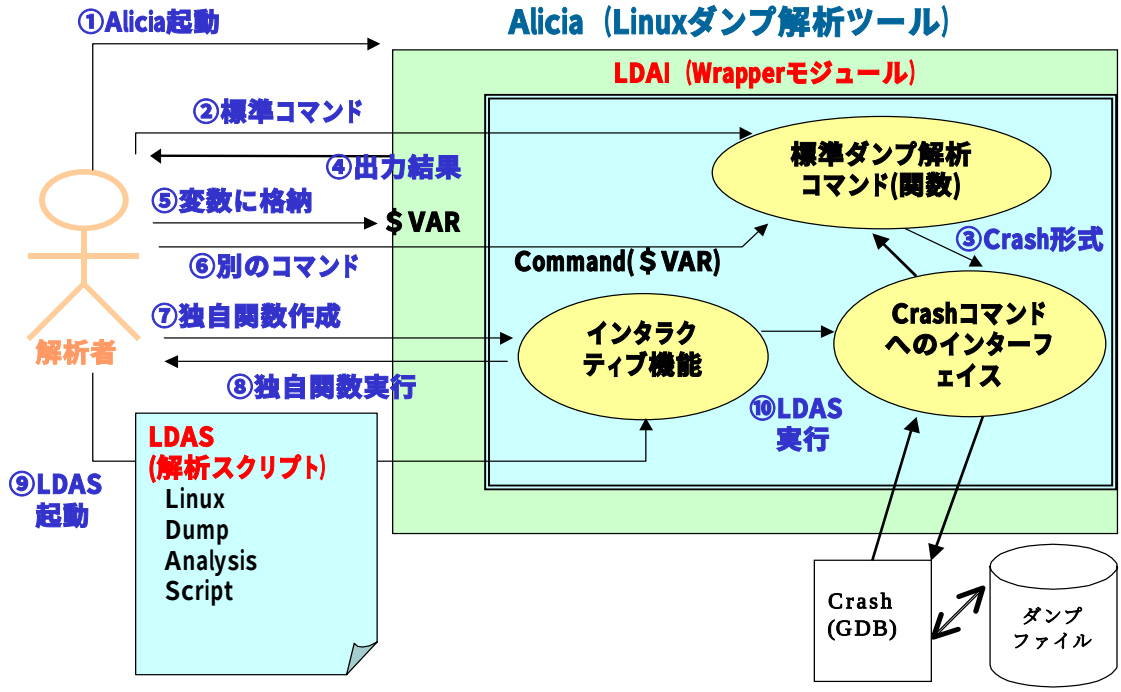

図3.3-1　Aliciaのオペレーション概要

4 Alicia による課題解決

Alicia は、2.5 節で挙げた課題に対して、スクリプト言語としてメジャーである Perl を解析環境に導入することで解決をはかっている。具体的に、どのように解決したかについて、Alicia を実際に使用してみることで、順に報告していく。下記説明の中で、左端が太い枠(図)は入力操作を表し、普通の枠(図)は出力結果を表している。
Alicia のバージョン 1.0.0 では、LCKD と Netdump のフォーマットに対応している crash のみをラッピングしている。次期メジャーバージョンでは、lcrash に対応を予定している。

4.1 コマンドライン

GNU Readline を採用しているため、コマンドのヒストリ機能、コマンド名やファイル名の補完機能などを bash 等のシェルと同様の操作感で、Alicia 上でダンプ解析を進めることができる。表 4.1-1 に、GNU Readline の代表的な操作を記述しておく。

表 4.1-1 GNU Readline の代表的な操作

機能(キーバインディング)	内容
ラインの先頭(C-a)	編集ラインの先頭にカーソルを移動
ラインの終了(C-e)	編集ラインの一番後ろにカーソルを移動
前ヒストリ(C-p、↑)	一つ前に実行したコマンドを表示させる
後方ヒストリ検索(C-r)	遡ってヒストリのインクリメンタル検索を行う
補完(TAB)	crash コマンド、Alicia コマンドの補完、およびファイル名の補完を行う

また、シェルエスケープの機能も持っており(crash も同様)、Alicia のシェルをエスケープして、外部エディタによるダンプ解析スクリプト(LDAS)の作成や、その他 bash コマンドなどが実行可能となっている。ダンプ解析スクリプト(LDAS)については、「4.3 節 簡易解析スクリプト」を参照のこと。

```
alicia> !vi sample.ldas
```

```
alicia> !ls -l
```

図 4.1-1 シェルエスケープ機能の例

Alicia は独自のコマンド群(API)の他に、crash のコマンドをそのまま実行できる。
ただし、行の最後には、セミコロン(;)が必要である。

例えば、図 4.1-2 のように、crash コマンドの sys を実行してみる。

```
alicia> sys;
```

```
  SYSTEM MAP: map
DEBUG KERNEL: vmlinux (2.4.21-9.35AX.reboot3smp)
    DUMPFILE: dump
        CPUS: 32
        DATE: Sat Oct  9 01:13:21 2004
      UPTIME: 02:04:25
LOAD AVERAGE: 65.99, 754.32, 975.01
       TASKS: 2166
    NODENAME: mlnx1
     RELEASE: 2.4.21-9.35AX.reboot3smp
     VERSION: #1 SMP Wed Sep 15 11:44:07 JST 2004
     MACHINE: i686  (2800 Mhz)
      MEMORY: 3.9 GB
       PANIC: "Fatal exception"
```

図 4.1-2 sys コマンドの実行

Alicia 独自の機能として、Perl の構文をコマンドライン編集に使用できることが大きな特徴になっている。例えば、図 4.1-3 のように変数にコマンドの実行結果を格納することが可能である。

```
alicia> $var1 = pass_through('sys');
alicia> print $var1;
```

※pass_through()関数は、crash のコマンドをそのまま実行する Alicia の標準関数(API)である。

```
  SYSTEM MAP: map
DEBUG KERNEL: vmlinux (2.4.21-9.35AX.reboot3smp)
    DUMPFILE: dump
        CPUS: 32
        DATE: Sat Oct  9 01:13:21 2004
      UPTIME: 02:04:25
LOAD AVERAGE: 65.99, 754.32, 975.01
       TASKS: 2166
    NODENAME: mlnx1
     RELEASE: 2.4.21-9.35AX.reboot3smp
     VERSION: #1 SMP Wed Sep 15 11:44:07 JST 2004
     MACHINE: i686  (2800 Mhz)
```

```
      MEMORY: 3.9 GB
       PANIC: "Fatal exception"
```

図 4.1-3 Perl 構文をコマンドライン編集に使用する例

また、その結果に対して Perl の構文による処理が可能である。図 4.1-4 は、crash のコマンド ps の結果を物理メモリでソートして表示（ページャーを使用）させている。

```
Alicia> $text = pass_through('ps');
alicia> $text =~ s/>/ /g;
alicia> @pss = split(/¥n/, $text);
alicia> shift @pss;
alicia> @mps = sort { (split(/¥s+/,$a))[8]<=>(split(/¥s+/,$b))[8]; } @pss;
alicia> less join("¥n", @mps);
```

```
   PID    PPID  CPU   TASK    ST  %MEM     VSZ    RSS  COMM
     0      0   0  c04a6000  RU   0.0       0      0  [swapper]
     0      1   1  c659e000  RU   0.0       0      0  [swapper]
     0      1   2  c659a000  RU   0.0       0      0  [swapper]
     0      1   3  c6598000  RU   0.0       0      0  [swapper]
     0      1   4  c6396000  RU   0.0       0      0  [swapper]
                              (省略)
```

図 4.1-4 Perl 構文の実行結果に対して Perl 構文で処理する例

4.2 インターフェース

Alicia は、既存のダンプ編集ツールをラッピングしているツールであり、そのラッピングされたツールが crash であろうと lcrash であろうと、どちらのツールを使っているか意識することなく解析スクリプトを作成できるようにすることを目指している。(バージョン 1.0.0 では crash のみの対応となっているが、lcrash にも対応する予定である)
もちろん、既存のダンプ編集ツールには、素晴らしいコマンドがたくさんあるため、それらを利用するインターフェースも実装している。しかも、それらのコマンド群の実行結果を加工できるため、既存ツールの独自のコマンドが無駄になってしまうこともない。
また、コマンドの出力結果を加工して、その一部をさらに別のコマンドの引数として渡すということもできる。

4.3 簡易解析スクリプト

Alicia はインタプリタの記述言語として、Perl を採用している。そのため、sial のような独自

な言語を使用することなく、解析スクリプトを記述できるようになっている。図 4.3-1 は、Perl で記述した解析スクリプトである。

```
sub psmem {
    my ($text, @pss, $t1, @mps);
    $text = pass_through('ps');
    $text =~ s/>/ /g;
    @pss = split(/¥n/, $text);
    $t1 = shift @pss;
    @mps = sort { (split(/¥s+/,$a))[8]<=>(split(/¥s+/,$b))[8]; } @pss;
    unshift (@mps, $t1);
    return join("¥n", @mps);
}
1;
```

図 4.3-1 解析スクリプトの例

これは、4.1 節で示した例を解析スクリプト化したものである。このように解析手順をスクリプト化して残すことにより、次回のトラブル解析にも利用できる。
それでは、実際のダンプ解析の場面を想定して、解析スクリプトの作成例を示してみる。

4.3.1 ダンプ採取直前のデータ

ここでは、Alicia の既存の LDAS と既存の crash コマンドを使用して、カーネルがクラッシュする直前の/var/log/message に、まだ書き込まれていないイメージを採取する解析手順をスクリプト化してみる。
手順を以下に示す。
(1) 指定されたパス名のハッシュを計算し、dentry 構造体のアドレスを算出する。
(2) dentry 構造体から、inode 構造体のアドレスを得る。
(3) inode 構造体にリンクされている dirty バッファのアドレスを得る。
(4) dirty バッファが存在するなら、その page 構造体のアドレスを得る。
(5) page 構造体から、該当するページのアドレスを得る。
(6) そのページの内容をダンプする。

表 4.3-1 に使用する関数群を表にする。

表 4.3-1 使用する関数の一覧

関数名(コマンド名)	内容
rd	指定されたメモリの内容を 16 進数と ASCII で表示する crash コマンド
pass_through	ラッピングしているツールのコマンドをそのまま実行する Alicia のコマンド(標準関数)
kernel	カーネルの構造体メンバにアクセスする Alicia のコマンド(標準関数)
get_dentry	入力されたパスの dentry 構造体のアドレスを返す作成済みの LDAS
list_c	指定された構造体の list_head リンクリストとなっているメンバ構造体の、リンク上の構造体アドレスをすべて配列で返す作成済みの LDAS

```
require 'list_c.ldas';
require 'get_dentry.ldas';
package LDAS::Data;
sub get_dirty_data {
    my $path = shift;
    my ($page, $virtual, $size);
    my @data = ();
    my %Pgs = get_dirty_pages($path);
    while (($page, $size) = each (%Pgs)) {
        $virtual = Alicia::kernel($page, 'page', 'virtual');
        push(@data, Alicia::pass_through("rd $virtual $size"));
    }
    return wantarray ? @data : "@data";
}

sub get_dirty_pages {
    my $path = shift;
    my ($dentry, $inode, $os, $om, $cs, $cm, @bhs, $count);
    $dentry = Alicia::get_dentry($path);
    $os = 'inode';
    $om = 'i_dirty_data_buffers';
    $cs = 'buffer_head';
    $cm = 'b_inode_buffers';
    $inode = Alicia::kernel($dentry, 'dentry', 'd_inode');
```

```
    @bhs = Alicia::list_c($inode,$os,$om,$cs,$cm);
    shift(@bhs);
    my %Pgs = ();
    foreach $bh (@bhs) {
        $count = Alicia::kernel($bh, 'buffer_head', 'b_count.counter');
        if ($count) {
            $page = Alicia::kernel($bh, 'buffer_head', 'b_page');
            $Pgs{$page} = Alicia::kernel($bh, 'buffer_head', 'b_size');
        }
    }
    return %Pgs;
}
1;
```

図 4.3-2 LDAS の作成例

※ このスクリプトはサンプルであり、Miracle Linux V3.0(2.4.21-9.30AX)以外での動作を考慮していない。

作成した LDAS を、get_dirty_data.ldas という名前で保存し、図 4.3-3 のように Alicia 上にロードし、実行する。

```
alicia> load 'get_dirty_data.ldas';
alicia> @bufs = LDAS::Data::get_dirty_data('/var/log/messages');
alicia> less @bufs;
```

```
ff081000:  69686320 7920646c 676e756f 6f207265    child younger o
ff081010:  7265646c 74634f0a 20392020 303a3130   lder.Oct  9 01:0
ff081020:  30333a39 6e6c6d20 6b203178 656e7265   9:30 mlnx1 kerne
ff081030:  69203a6c 2074696e 20202020 20202020   l: init
ff081040:  43205320 42463430 20303832 20202020    S C04FB280
ff081050:  20202034 20312020 20202020 20203020   4     1      0
ff081060:  34332020 20202020 32202020 20202020     34       2
ff081070:  28202020 4c544f4e 4f0a2942 20207463      (NOTLB).Oct
ff081080:  31302039 3a39303a 6d203033 31786e6c   9 01:09:30 mlnx1
ff081090:  72656b20 3a6c656e 6c614320 7254206c    kernel: Call Tr
ff0810a0:  3a656361 5b202020 3130633c 37623932   ace:   [<c0129b7
ff0810b0:  205d3e34 65686373 656c7564 656b5b20   4>] schedule [ke
ff0810c0:  6c656e72 7830205d 20343533 66783028   rnel] 0x354 (0xf
ff0810d0:  31376637 29303465 74634f0a 20392020   7f71e40).Oct  9
ff0810e0:  303a3130 30333a39 6e6c6d20 6b203178   01:09:30 mlnx1 k
ff0810f0:  656e7265 5b203a6c 3130633c 66303334   ernel: [<c01430f
```

```
ff081100:  205d3e35 65686373 656c7564 6d69745f   5>] schedule_tim
ff081110:  74756f65 656b5b20 6c656e72 7830205d   eout [kernel] 0x
ff081120:  28203536 37667830 65313766 0a293862   65 (0xf7f71eb8).
ff081130:  2074634f 30203920 39303a31 2030333a   Oct  9 01:09:30
ff081140:  786e6c6d 656b2031 6c656e72 3c5b203a   mlnx1 kernel: [<
ff081150:  39313063 31646266 5f205d3e 6c6f705f   c019fbd1>] __pol
ff081160:  6961776c 6b5b2074 656e7265 30205d6c   lwait [kernel] 0
:
```

図 4.3-3 get_dirty_data.ldas の実行方法と実行結果

4.3.2　リソース調査

あるプロセスが大量に CPU を使用していないか調査する。
いくつか方法が考えられるが、ここでは、ps コマンドの結果に、それぞれのプロセスの CPU 使用時間を付加し、使用時間順にソートしたものを表示するスクリプトを利用してみる。

作成した関数は、以下の二つである。

(1) LDAS::Cputime::cputime
　指定されたプロセスの CPU 時間を得る。
　CPU 時間(秒:小数点 2 位まで) = LDAS::Cputime::cputime('task_struct 構造体のアドレス')

(2) LDAS::Cputime::cputime_all
　全てのプロセスの CPU 時間を ps コマンドのイメージに付加する。
　ps イメージのエントリ(配列) = LDAS::Cputime::cputime_all()

図 4.3-4 は、上記の(2)の関数を利用して、全てのプロセスの CPU 時間を表示している。

```
alicia> load 'cputime.ldas';
alicia> @cpu = LDAS::Cputime::cputime_all();
alicia> $output = join("¥n", @cpu);
alicia> less $output;
```

```
  PID   PPID  CPU   TASK    ST  %MEM     VSZ    RSS  Cputime  COMM
 2090      1    0  f42dc000  RU   0.0    3044   1672   22.79 oprofiled
28657  28233  25  ef2da000  UN   0.0    2328   1340   11.82 multitask
30210  28233  21  e9ff2000  RU   0.0    3352   1404   10.68 multitask
 1271   1264   5  f47f6000  IN   0.2   49384   6932   10.58 ocssd.bin
29229  28233   6  f0ec4000  RU   0.0    3336   1388   10.11 multitask
28480  28233  23  e9bd0000  UN   0.0    2328   1340    9.78 multitask
29582  28233   3  ebcc6000  UN   0.0    2328   1340    9.78 multitask
29004  28233  24  e8fc4000  UN   0.0    2328   1340    9.51 multitask
```

```
  28686  28233   0  f2776000  UN   0.0    2328   1340    9.42 multitask
  30188  28233   3  e9f68000  RU   0.1    3308   2320    9.30 multitask
  30051  28233   3  f0bda000  UN   0.0    2328   1340    9.06 multitask
  28660  28233  24  e9d3a000  UN   0.0    2328   1340    8.94 multitask
  29730  28233  16  f2d0e000  UN   0.0    2328   1340    8.91 multitask
  28524  28233  11  ea5fa000  UN   0.0    2328   1340    8.85 multitask
  29013  28233  22  ed1bc000  RU   0.0    3296   1348    8.79 multitask
  29534  28233  20  e8efe000  UN   0.0    2328   1340    8.76 multitask
  29807  28233  27  edb44000  RU   0.0    3320   1372    8.67 multitask
  28943  28233   8  f3642000  UN   0.0    2328   1340    8.61 multitask
  30062  28233  16  f24f8000  RU   0.0    3296   1348    8.58 multitask
  29728  28233   5  ee714000  RU   0.0    3316   1368    8.55 multitask
  28926  28233  11  ea488000  UN   0.0    2328   1340    8.46 multitask
  29565  28233   3  e956e000  UN   0.0    2328   1340    8.46 multitask
:
```

図 4.3-4 cputime の実行方法と実行結果

4.4 解析手順の共有化

今までは、ダンプの解析を行うために、既存のダンプ編集ツールを起動し、そのツールが装備しているコマンドをたたき、その結果を元にメモリの内容を追いかけ、原因を追求するという手順であった。しかし、この障害追及手順は、解析者本人しか分かっておらず、また解析者本人でさえ、この手順を再現するのに同じ操作を一から行う必要がある。
そのため、同じようなダンプが採取された場合に、多くの人々が、同じ手間を、一から始めなければならない。crash や lcrash にも、障害追及手順をプログラミングできる環境は整っているが、Perl 言語でプログラミングするような手軽さはない。
そこで、ダンプ解析用の便利なスクリプトが公開されたなら、世界中のダンプ解析の手助けとなり、また、ダンプ解析に詳しくないシステム管理者でも障害の追求における一つのアイテムとなるはずである。
現在、Alicia では、Perl 言語による手軽なダンプ解析スクリプトの作成を支援する環境を提供しているが、今後、解析スクリプトを共有するためのデータベースの仕組みなどを、Alicia の開発を続ける中で考えていきたい。

4.5 課題の整理

ここで、2 章の課題に対して、Alicia が解決した部分、また Alicia のバージョンが上がることで解決する部分を表 4.5-1 にまとめてみた。

表 4.5-1 Alicia による課題解決

課題	内容	Alicia バージョン
コマンドライン	Alicia のシェルを採用することにより、GNU Readline の機能を持ち、crash、lcrash 間の差はなくなった。	バージョン 1.0.0 で対応済み
インターフェース	インターフェースが Alicia に統一されることで、crash、lcrash の操作の違いに悩まされることはなくなる予定である。さらに既存の Perl 関数を利用することができ、解析の幅が広がった。また、結果の表示も自由にカスタマイズ可能になった。	バージョン 1.1.0 で対応予定 (lcrash をラッピングする)
簡易解析スクリプト(インタプリタ)	既存ツールの拡張コマンド作成機能は残したまま、新たに Perl でスクリプトを作成することができるというオプションが加わった。	バージョン 1.0.0 で対応済み
解析の共有化	Alicia を使うことで簡易に解析スクリプトを記述できるようになった。しかし、そのスクリプトをどのように共有していくかのシナリオを示していく必要がある。	どのように共有していくかについてはこれから議論をしていく

5 考察

5.1 Alicia の有効性

1 次障害解析は、時間との勝負である。原因が分からないまでも、どの操作を行ったからその障害が発生したのか、何を実行しなければカーネルのバグを回避できるのか、すぐに切り分ける必要がある。
また、ダンプは生き物と呼ばれるように、その瞬間で違う顔を見せる。ここで必要とされるのは、素早く簡単に解析スクリプトをその場で作成できる環境である。そのような環境を、Alicia は用意する。そして、Alicia を使って解析をすればするほど、解析者のノウハウが解析スクリプトとなって蓄積されていくことになる。
Linux のダンプ解析に Alicia があることを前提に　あるメーカのメインフレームのダンプ解析で使用しているツールや環境と比較すると、Linux に不足しているのは、既存解析スクリプトの数、種類とダンプ解析の経験、実績である。今後 Alicia を使用したダンプ解析の実績を積むことと、その中で作成された有効な LDAS を蓄積することによって、メインフレームと同等のダンプ解析レベルに達することが可能になる。

5.1.1 Alicia の使用による解析手順の高速化

今回、4 章で用いたような解析スクリプトを作成すると、既存のツールを使った場合と比較して、格段に解析時間が短くなる。ここでは、「4.3.2 リソース調査」で紹介した解析スクリプトである cputime_all を使った場合と既存ツールを使った場合を、作業時間に関して比較してみる。（図 5.2-1）
cputime_all は、システム・クラッシュ発生時に稼動していた全てのプロセスを CPU の使用時間の多いものから順番に表示させるための補助ツールである。全てのプロセスは、init_task_union を起点として、そのリンクリストを辿ることによって得られるが(cputime.ldas とうファイルにはこちらの方式の例題も用意している)、cputime_all では、ps コマンドの出力結果を流用している。そして、それぞれのプロセスの task_struct 構造体から CPU 利用時間を抽出し、ps コマンドの出力に付加している。
これを既存のツールで実施した場合で作業時間を試算してみる。task_struct 毎の CPU 利用時間と次の task_struct 構造体を求めるのに、少なくとも 10 秒程度は必要である。1000 プロセスが稼動していたときに採取されたダンプで、上記手順を実施した場合、10000 秒、すなわち 2 時間 40 分以上かかることになる。ちなみに、Alicia を開発するときに使用していたダンプでは、2166 プロセス存在しているため、既存ツールをそのまま利用しただけでは、5 時間以上かかる計算となる。これに対して、cputime_all を使用した場合の実行時間は約 1 分程度である。このスクリプトの作成に、カーネルの調査を含めても、1 時間程で作成できたので、解析時間の短縮に関してこの差は非常に大きいと考える。

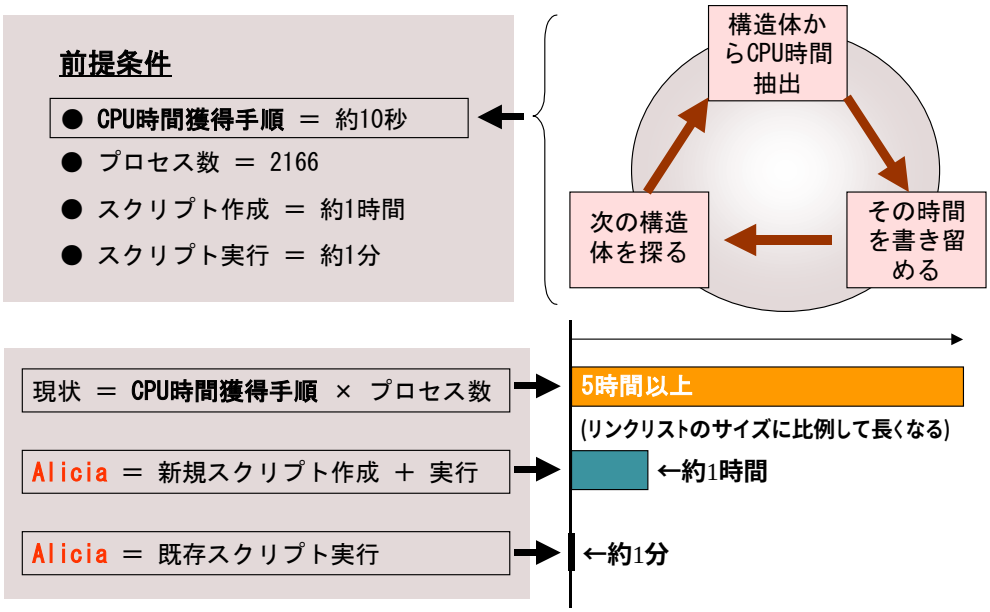

図5.2-1 Aliciaによる解析手順の高速化例 – 全プロセスのCPU時間の獲得

今後、Linux がより大規模なシステムに適用されれば、リンク構造を持つリストのサイズも大きくなり、このような解析スクリプトの効果がさらに増すことになる。また、スクリプトの容易なカスタマイズ性、再利用可能性は、crash の定型の出力が単純に使えないケース(例えば、リンクリストがさらにリンクリストの構造を持つ場合など)で特に顕著であり、Alicia で稼動する解析スクリプトは、その作成時、実行時に大きな効果を発揮することが分かる。再利用性が高いため、様々な場面においてダンプ解析の高速化を期待できる。

5.1.2 Alicia の活用方法

このように、そのノウハウが共有化されれば、世界中のユーザにとってダンプ解析が格段に楽になるに違いない。しかも、その共有データは、Linux カーネルの勉強の逆引き辞典としても使える可能性を秘めている。カーネルのソースだけではなく、実データとともに追っていけば、より理解が深まるはずである。

また、お客様のシステムに Linux が導入され、そのシステムに Alicia を適用していたなら、障害の発生でダンプを採取後、1 次ダンプ解析スクリプト(LDAS)を実施し、その結果を送付してもらうことで早期に障害を切り分ける、という利用方法が考えられる。もちろん、1 次ダンプ解析スクリプトをどのようなものにするかについては、多くの改良を積み重ねていく必要があるだろう。

5.2 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 5.4-1 に示す。Alicia 開発における最大のポイントは Alicia 全体の設計作業にあった。また、品質を高めるためのテスト作業にも重点を置き、表 5.4-1 に示した開発項目以外にも 466 ステップのテストプログラム群を開発している。

表 5.4-1　Alicia の開発ステップ数

項番	開発項目	開発ステップ数
1	Perl モジュール	2159
2	Alicia（実行、コンフィグ）	53
3	モジュールインストール	200
4	README	335
5	LDAS	923
	合計	**3670**

5.3 今後の課題

今回の開発作業、および試行作業において、Alicia の有効性は十分に確認できたと考える。Alicia により、ダンプ解析に対するインフラは整いつつあるが、それと並行して、ダンプに採取されるデータとして何が有効であるかを議論していかなければならない。現段階では、ダンプ解析のために意識的にデータを残そうという試みがほとんどない。そのため、ダンプにはシステムが停止する瞬間のメモリの内容しかない。現在のところ、LKST がインフラとしてこの分野に適用可能と考えられる。LKST は、システム稼動中のデータを LKST のバッファに蓄えており、そのバッファをうまく利用すれば、過去にさかのぼってダンプの解析が可能である。例えば、ディスクやテープ装置に関連する障害には、過去の IO がどのように行われたのかについて、カーネル内のイベントとしてメモリに(ダンプに)情報が残されていれば、原因の特定に非常に有効である。

今後は、LKST のような別のコンポーネントとの連携を考慮して、Linux のトラブル追求に貢献していきたい。

6 計画

近い将来(バージョン 1.1)に、lcrash への対応を予定している。また、遠い将来には、ノウハウ DB、GUI 化を考えている。

7 おわりに

Alicia の開発は、ユニアデックス、NTT データ、ミラクル・リナックスの三社協同による体制で行った。デザイン、コーディング、テスト、マニュアル作成は主にユニアデックスで実施したが、それぞれの工程のチェックポイント、および月に 1 回の会議でのコメント、また、NTT データ社からの詳細な試用報告を受けて、Alicia のブラッシュアップをしていくことができた。また、Alicia の開発と並行して、ミラクル・リナックス社によって、既存の Linux カーネルダンプ編集ツールの詳細な調査が実施された。この調査により、 今回ラッピングした crash の詳細な構造と、crash の拡張コマンド作成に必要な情報を得ることができた。これは、ダンプ解析環境の向上への多大なる貢献と考える。また、Alicia と既存ツールとの住み分けが明確化し、Alicia の開発方針に間違いがないことも確認できた。
協同体制による開発により、各社が持っている力を単純に足し合わせて得られる成果以上のものを引き出せたと考える。

8 参考文献

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

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

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

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

本書は、独立行政法人 情報処理推進機構から以下の８社への委託開発の成果として作成されたものです。
委託先企業：（株）日立製作所（幹事会社）
（株）SRA、（株）NTT データ、新日鉄ソリューションズ（株）、
住商情報システム（株）、（株）野村総合研究所、ミラクル・リナックス（株）、
ユニアデックス（株）　（五十音順）