DEBUG HACKS™

デバッグを極めるテクニック & ツール

吉岡 弘隆
大和 一洋、大岩 尚宏　著
安部 東洋、吉田 俊輔

O'REILLY®
オライリー・ジャパン

Debug Hacks

デバッグを極めるテクニック & ツール

吉岡 弘隆、大和 一洋
大岩 尚宏、安部 東洋、吉田 俊輔　著

O'REILLY®
オライリー・ジャパン

Debug Hacks 推薦の言葉

プログラムにはバグが付き物です。バグは人間の予想を超えたところからやってきます。世界最初のバグは、リレー式計算機の中にまぎれこんだ蛾だったそうです。あわれリレーの間に挟まれた蛾によってコンピュータの誤動作が引き起こされました。このエピソードがきっかけとなり、プログラムの間違いのことがバグと呼ばれるようになったのだそうです。この蛾は後に COBOL の開発者となるグレース・ホッパー女史の日記に記念として張りつけられていたと聞きます。

それから半世紀以上が過ぎ、あいかわらずバグは生み出され続けています。「プログラムは思った通りではなく、書いた通りに動く」というのはプログラマの中に伝わる「ことわざ」のひとつです。そして人間は間違えるものなので、プログラムの中にはバグが入り込みます。ある意味、バグとは人間の限界を見せつけてくれるものなのかもしれません。ほとんどのバグはちょっとプログラマを悩ませるくらいのかわいいものですが、最近はバグによって引き起こされた「事件」によって新聞をにぎわすようなこともたびたび起きています。

プログラマの仕事はプログラムを作ることですが、コンピュータが社会に浸透し、プログラムが複雑化するに従って、完全なプログラムを書くことは非常に困難になってきています。その結果、すべてのプログラマは必然的にバグに対処する必要があります。個人的にはプログラマの時間の大半が、バグを見つけることと、それらを直すことに費やされているのではないかと感じます。

しかし、ただやみくもにバグを探してもうまくいくわけはありません。バグにはバグの見つけ方があり、直し方があるのです。特にバグを見つけだし、特定するにはさまざまなテクニックが存在します。「デバッグ」という言葉は「バグを直すこと」のような印象がありますが、実際にはどこにあるのか特定されたバグはまったく恐ろしいものではなく、たいていはすぐに直すことができるものです。デバッグの神髄はバグの発見と特定にあるのです。本書は歴戦のプログラマが経験から獲得したバグの見つけ方・直し方が満載されています。特に普段はお目にかからないような Linux そのもののバグについての Hack は、貴重な情報ではないかと思います。いくつかの Hack は多くのプログラマが日常的に使う

ものではないかもしれませんが、それでもなおその発想は参考になります。特に gdb や valgrind や oprofile のような便利なツールについてきちんと解説してあるのがありがたいところです。また、2 つほど Ruby の「バグ」についても扱っていただいてます。ありがたいことです。

本書がプログラマの皆さんのバグへの戦いの日々が少しでも楽になるための「道標」となることを期待しています。

2009 年 3 月 羽田空港にて
まつもと ゆきひろ

クレジット

著者について

吉岡弘隆（Hiro Yoshioka）

ミラクル・リナックス所属のプログラマ。カーネル読書会という Linux の技術系勉強会を 1999 年から主宰している。慶応義塾大学大学院修了。日本ディジタルイクイップメント（DEC）研究開発センタ、日本オラクルを経てミラクル・リナックスを創業。大学を卒業以来、数々のソフトウェア製品（日本語 COBOL、DEC Rdb、Oracle 8、MIRACLE LINUX、Asianux 等）の開発を行ってきた。

JIS X0208:1990/X0212:1990 の標準化、U-20 プログラミングコンテスト審査委員、セキュリティ & プログラミングキャンプ、プログラミング部門主査。

2008 年、経済産業省商務情報政策局長感謝状、楽天テクノロジーアワード 2008 金賞受賞。

ブログ「ユメのチカラ」: http://blog.miraclelinux.com/yume/

「未来のいつか /hyoshiok の日記」: http://d.hatena.ne.jp/hyoshiok/

大和一洋（Kazuhiro Yamato）

ミラクル・リナックスで働くソフトウェアエンジニア。これまでは、Linux カーネルや GLIBC まわりの仕事が中心であったが、最近では、gstreamer などのメディア関連のハックにも精を出す。GUI ツールは苦手で、開発環境はもっぱら、vim + gcc + gdb。

大岩尚宏（Naohiro Ooiwa）

ミラクル・リナックス株式会社勤務のソフトウェアエンジニア。携わった業務は開発よりもカーネルの調査・解析が多く、ドライバ・ネットワーク・ファイルシステムなど数多くの幅広いバグを担当。バグであれば何でも引き受ける。最近では Linux における省電力を検証している。絵は得意だが、日本語・英語が苦手。上司にはコミュニティにパッチを出すよう急かされる。日本を代表する Linux 開発者に会う機会が多い。

安部東洋（Toyo Abe）

ミラクル・リナックス所属のソフトウェアエンジニア。Hello World プログラムもロクに書けない状態から Linux カーネルの世界に入ってしまい、泣きそうになった経験を持つ。今まで x86 アーキテクチャだけだったので、執筆が一段落したら ARM に挑戦しようと企てている。

吉田俊輔（Shunsuke Yoshida）

ミラクル・リナックス所属のシステムエンジニア。どこにでもいる自称、一般人。地方ソフトウェア企業からメーカー系 SI 企業を経てミラクル・リナックスに入社。OS/DataBase/Network/ 仮想化等のインフラ系 SE。小江戸らぐ /YLUG（横浜 LinuxUsersGroup）/USAGI 補完計画等、関東近郊の OSS コミュニティに参加。イベント参加 / 出展や原稿執筆を行っている。

ブログ「第三のペンギン」: http://blog.miraclelinux.com/thethird/

コントリビュータについて

島本裕志（Hiroshi Shimamoto）

ソフトウェアエンジニア。主に問題対応を通じて Linux を学ぶ。特技は core・crash 解析。x86、スケジューラ、リアルタイムなどの分野で Linux カーネルコミュニティ活動を行っている。

美田晃伸（Akinobu Mita）

フィックスターズ社のプログラマ。デバッガの使い方がよく分からず、主に printf デバッグしている[†]。

謝辞

本書は『BINARY HACKS —— ハッカー秘伝のテクニック 100 選』という素晴らしい著作にインスパイヤされ企画されました。本書の執筆にあたっては、ミラクル・リナックスの社員ばかりではなく、コントリビュータとして、島本裕志さん、美田晃伸さんに、興味深い Hacks を寄稿していただきました。ここに記して感謝の意を表したいと思います。

また、推薦の言葉をプログラミング言語 Ruby の生みの親で著名なまつもとゆきひろさんに頂きました。どうもありがとうございました。

—— Hiro Yoshioka

† Linux カーネルの Fault Injection の作者でありメンテナ。

はじめに

本書はプログラマがプログラムをするときに避けて通ることのできないデバッグというプロセスについて記したものです。デバッグはプログラミング言語や開発環境に依存しない、いかなるプログラミングでも避けて通ることができない作業にもかかわらず、あまりまとめられることがなく、適当な参考書がほとんどない分野でした。

プログラミング入門書はあまたあるのに、何でデバッグ入門書がほとんどないのでしょうか？

プログラマの作業を設計、コーディング、テスト、デバッグなどのプロセスで考えると、多くの時間をデバッグで費しているということは、少なくありません。実際、ソフトウェア開発のコストの多くは、ソフトウェアを新規に作成するのではなく、ソフトウェアを拡張したり、変更したり、不具合を修正したりすることに費やされていると言われています。

プログラムを新規に作るより、デバッグの方が時には難しいと感じることがあります。設計やコーディング、あるいはテストに関するベストプラクティスが、書籍という形で広く流通しているにもかかわらず、ソフトウェア開発における重要なフェーズであるデバッグについての入門書が、あまり見当らないのはちょっと奇妙な感じがします。

デバッグという作業はプログラマが 10 人いれば 10 通りのデバッグ方法があるかのような極めて属人的な作業です。そして、デバッグの達人もいれば、そうでない人もいる。軽やかに、それこそ鼻歌まじりに魔法のようにバグを見つけ出し、直すハッカーもいます。

今回『Debug Hacks』を著すにあたって、心がけたことのひとつに、わたしたちが出会った事例を中心に、具体的なデバッグ方法を明らかにするということです。

わたしたちは、わたしたちのデバッグ手法を記すことによって、自分たちのデバッグの方法について、自分たちが理解したいと考えました。どうして、このコマンドを利用して、デバッグをしたのだろう。そもそも、どうやってこのバグを見つけたのだろう。そのような自問自答の繰り返しの中で、デバッグのプロセスを炙り出すことを試みました。

わたしたちが実際に仕事で遭遇した事例をもとに、それぞれの Hack を記しました。例

によっては、説明を単純化するために、新たにテストプログラムなどを書き下ろしたものもありますが、そのどれもが、わたしたちが遭遇したバグをベースに説明するようにしました。事例を用いることによって、机上の空論ではなく経験をベースとした記述になっていると思います。

そのようなデバッグのプロセスの記述は、わたしたちのデバッグ方法が仮に改良の余地が多々あるとしても、明示的に記述したことによって、それをベースに他の方法を議論したり、もっと良い方法を発見するヒントになるという意味で地味だけど重要な作業だと思います。むしろ、そのようなドロ臭い作業の積み重ねこそがデバッグの方法の進歩に繋ると信じています。

わたしたちの方法よりもっと良い方法がきっとあると思います。良いデバッグ方法について語るためにはベースとなるたたき台が必要だと考えます。それが、この『Debug Hacks』となることを願ってやみません。特にハッカー（ベテランプログラマ）の皆さんには自分のスタイルと対比の上、読み解いていただければと思います。わたしたちの方法の適用範囲、長所短所などさまざまな観点から議論をいただければ、それがわたしたちプロフェッショナルなプログラマのデバッグに対するより深い理解になると考えます。

デバッグの方法はこれまで明示的に書き記されるものより、それぞれの経験によって培った、ある種秘伝のようなものでした。プログラマとして研鑽を積んでいくうえで、デバッグのテクニックを身につけることが、わたしたちプロフェッショナルなプログラマにとっての基礎体力となると考えています。

また、今回のわたしたちのように、自分のデバッグ方法を広く公開するというスタイルが一般化すれば、より多くの人たちとベストプラクティスを共有でき、それがわたしたちプログラマにとっての貴重な財産となると思います。

デバッグ方法は、ツールや開発環境の進歩によって、今後も変化していくと考えます。そして、わたしたち自身のデバッグやプログラミングスタイルの理解によっても、どんどん進化していくと思います。それを能動的に学ぶためにも、本書を参考に多くのプログラマの皆様と一緒に鍛錬していきたいと考えています。

本書で必要となる知識と想定する読者

本書は、主にC/C++などのプログラミング言語で開発するアプリケーションプログラマやLinuxカーネル開発者などを対象としています。特に言語や開発環境は想定しませんが、例としてLinux環境を利用しています。低レベルでのデバッグの場合、コンピュータアーキテクチャの基礎知識、プログラミング言語の基礎知識などを必要とします。また開発環境としてUnix系のプログラミング環境の基礎知識を必要とします。それ以外の知識は特に仮定していません。

想定している読者は、自分でプログラムの設計、実装、テスト、デバッグなどを行う初

級から中級プログラマです。自分のプログラミングスキルをもっと伸ばしたいと願っている人たちに向けて記しました。C/C++ プログラマだけではなく、Perl/PHP/Python/Ruby などスクリプト言語でプログラムを書いている人たちにとっても、わたしたちが記した方法の多くは、たとえ言語や道具だてが異なっているとしても考え方は参考になると思います。また Windows や Mac など異なるプラットフォームでプログラムしている人たちにとっても、同様にその考え方は参考になると思います。

特に学生の皆さんには、本書を読んでいただきたいと願っています。プログラミング言語の入門書は一通り読んだけれど、もっともっとプログラミングを極めたいと考えている皆さんには、本書で書いた Hacks が参考になると思います。この本が自分の学生時代にあったら読みたかったなあと思いながら編集しました。

スクリプト言語でプログラムを書いている人はコンピュータアーキテクチャや機械語を意識することは日頃ほとんどないことです。しかし、例えば Ruby の処理系がセグメンテーションフォルトで突然クラッシュした時、それを修正する必要にかられた場合、本書が扱うような知識やテクニックが必要になってきます。プログラマとしての幅をもう少し広げたい人にとって、本書はそのきっかけになると考えます。

また独自のスタイルを持つバリバリのハッカー（ベテランプログラマ）の皆さんにもぜひ読んでいただき、忌憚のないご意見をいただきたいと思います。特に Linux カーネルのデバッグについて正面切って取り上げた参考書がほとんどありませんので、わたしたちの設定したスコープ、想定読者像を含めた本書の構成そのものについても、自分だったらこうするという観点からのコメントなどを頂ければ幸いです。

本書で扱うこと扱わないこと

わたしたちは主に Linux 上でのアプリケーションや Linux カーネルそのものを例題に選びましたが、それはたまたまわたしたちが、そのような分野で仕事をしているからに他なりません。

Web アプリケーション、組み込み、ゲーム、ミドルウェア、などなどプログラムと言ってもさまざまな応用分野があります。それぞれの分野に特有なデバッグ手法というのがあるかと思いますが、本書では取り上げていません。すべてを網羅するオールマイティのデバッグ手法というのは存在しないと思いますが、本書ではより一般的なデバッグ手法に焦点をあてています。そのために多くの場合、その考え方は活用できると思います。

扱うこと

本書では、デバッグの基本的な考え方、方法を紹介します。アプリケーションプログラムのデバッグだけではなく、OS（Linux カーネル）のデバッグについても扱います。また gdb のようなデバッガの使いかたや、ダンプの読みかた、crash の使いかた、kprobes

や oprofile などのデバッグに便利なツールについても触れます。

本書で触れたツール以外でも多くの優れたツールがあります。例えば、ftrace、LTTng、dmalloc、blktrace、lockdep、kgdb、KDB、utrace、lockmeter、mpatrol、e1000_dump、git-bisect、kmemcheck などについては触れられませんでした。これらのツールについて、読者の皆様の Debug Hacks をぜひ、伺いたいです。

扱わないこと

本書ではプログラミング一般、例えば、ソフトウェアの設計、デバッグしやすいコーディング作法やテスト方法論などは扱いません。TDD（テスト駆動開発）はテスト、デバッグを表裏一体の開発プロセスとしていますが、本書の範囲の外です。

また一般にトラブルシューティングとして知られる、何らかのトラブルが発生したときの問題の切り分け、ワークアラウンド（回避策）の提示なども本書の対象外としました。

本書では、不具合（バグ）を認識した後に、それを修正するという狭義のデバッグについて焦点をあてます。

本書の構成

「1章　こころがまえ（warmingup）」は、デバッグとはどのようなプロセスかを概説します。また本書『Debug Hacks』の全体像を記しています。

「2章　デバッグ前に知っておくべきこと」は、デバッグの基本として、デバッガ（GDB）の使い方、Intel アーキテクチャの基本、スタックの基礎知識、関数コール時の引数の渡され方、アセンブリ言語の勉強方法などを記しています。

「3章　カーネルデバッグの準備」は Linux カーネルのデバッグ方法の基本を記しています。Oops メッセージの読み方、シリアルコンソールの使い方、ネットワーク経由でのカーネルメッセージの取得、SysRq キー、各種ダンプの取得方法、crash コマンドの使い方、IPMI および NMI watchdog でのクラッシュダンプの取得、カーネル特有のアセンブリ言語などなど、カーネルデバッグの基本について記しています。

「4章　実践アプリケーションデバッグ」は、ユーザアプリケーションの実践的なデバッグ方法について記しています。スタックオーバーフローによるセグメンテーションフォルト（SIGSEGV）、バックトレースが正しく表示されない、配列の不正アクセスによるスタック破壊、ウォッチポイントを活用した不正メモリアクセスの検知、malloc()/free() での障害、アプリケーションのストールなどさまざまな事例によるデバッグ方法を記しています。

「5章　実践カーネルデバッグ」は、カーネル障害のデバッグ方法について記しています。カーネルパニック（NULL ポインタ参照、リスト破壊、レースコンディション）、カーネルストール（無限ループ、スピンロック、セマフォ、リアルタイムプロセス）、動作のスローダウン、CPU 負荷が高くなる不具合についてデバッグ方法を記しています。

「6章　差がつくデバッグテクニック」は、デバッグするにあたってのさまざまなツールの紹介やちょっとしたノウハウなど広範囲なものを集めました。紹介しているツールやテクニックもstrace、objdump、Valgrind、kprobes、jprobes、KAHO、systemtap、procファイルシステム、oprofile、VMware vprobe、フォルト・インジェクション、Xenなど多岐にわたります。その他、OOM Killerの動作と仕組み、GOT/PLTを経由した関数コールの仕組みと理解、initramfs、RT Watchdogを使ってリアルタイムプロセスのストールを検知する方法、手元のx86マシンが64ビット対応かどうか調べる方法まで記しています。

「付録 Debug Hacks 用語の基礎知識」は、本書に登場する用語の解説です。各Hackを読み進める中でわからない用語に出会ったときは、この付録を参照してみてください。

本書の利用法

本書は、1章以外、特に読む順番を仮定していません。前提知識をお持ちの方は、興味を引く項目をランダムに読んでも構いません。読み方はもちろん自由です。基礎的な知識を把握したいのならば、1章、2章をじっくり読んで、参考文献などにも目を通すとよいでしょう。バリバリのカーネルハッカーの方は、本書で取り上げたツールの使い方など、まだまだ序の口と感じるかもしれません。その場合は、ハッカー流のツッコミを教えていただけると幸いです。

本書での表記

等幅（sample）

ファイル名、サンプルコード、出力、コマンドなどを示しています。

等幅太字（sample）

ユーザ入力などを示しています。

ヒント、アドバイスなどを示しています。

注意ないし警告などを示しています。

各 Hack の左隣にある温度計アイコンは、それぞれの Hack の相対的な難易度を示しています。

 初級　  中級　  上級

意見と質問

本書の内容については、最大限の努力を持って検証および確認を行っていますが、誤りや不正確な点、誤解や混乱を招くような表現、誤植などもあるかと思います。本書を読んで気づかれたことがありましたら、今後の版で改善できるようにお知らせいただけると幸いです。

株式会社オライリー・ジャパン
〒 160-0002　東京都新宿区坂町 26 番地 27　インテリジェントプラザビル 1F
電話　03-3356-5227
FAX　03-3356-5261
電子メール　japan@oreilly.co.jp

本書に関する技術的な質問や意見については次の宛先に電子メールを送ってください。

japan@oreilly.co.jp

本書の Web ページには、サンプルコード†、正誤表、追加情報が掲載されています。

http://www.oreilly.co.jp/books/9784873114040/

オライリーに関するその他の情報については、次の Web サイトを参照してください。

http://www.oreilly.co.jp/（日本語）
http://www.oreilly.com/（英語）

† このサンプルコードは、筆者が執筆するときに使用したプログラムであり、さまざまな環境において動作を保証するものではありません。また、予告なしに変更されることがあります。なお、サンプルコードについての対応はできかねますのでご了承下さい。

目　次

1章 こころがまえ（warmingup）
Hack #1-3

HACK #1 デバッグとは
デバッグのプロセスの基礎から応用まで、本書では説明します。

そもそもデバッグというのは、どのような作業なのでしょう。プログラマなら誰でも行う作業なのに、そのプロセスについて詳細に記述した文書がほとんどないことには驚かされます。プログラマが10人いれば10人分の多彩な方法があります。しかし、その方法について明確に記されたことはほとんどありません。

本書では、わたしたちが、自らのデバッグ体験に基づいてそのデバッグ方法を語っています。これがベストな方法であるとか、これ以外の方法がないとは到底言えませんし、言うつもりもありません。それでも、このようにデバッグの方法を明示的に記すことには何がしかの価値はあるでしょう。それは、明示的に記述することにより、経験の浅いプログラマにとっては誰にも教えてもらえなかったデバッグの方法について直接的に学ぶ機会になるでしょうし、経験豊富なプログラマにとっても、自分のデバッグ方法についての考え方を再確認するきっかけになることでしょう。特に経験豊富なプログラマの皆さんにはぜひ本書を読んでいただき、自分の方法との差分を考察していただければと思います。また、その差分を明らかにし積み重ねることにより、われわれプログラマがよりよいプログラミングやデバッグについての知見を得ることができると思います。

プログラミングプロセス

プログラマがコードを書いて、それを完成させていくまでのプロセスは、要求定義、設計、コーディング、テスト、デバッグというような段階を踏むと思います。ここでは特にコーディング、テスト、デバッグのあたりについて詳細に議論することにします。

コードを書いているとき、ある程度、実装ができたなと感じたら、コンパイル、ビルドなどをして、コンパイルエラーやビルドエラーがなくなる程度まで修正します。

コンパイルエラーがなくなれば通常はまがりなりにも動きますので、とりあえず動作させてみて、期待する振る舞いになるか確認します。これは正式なテストというよりも、と

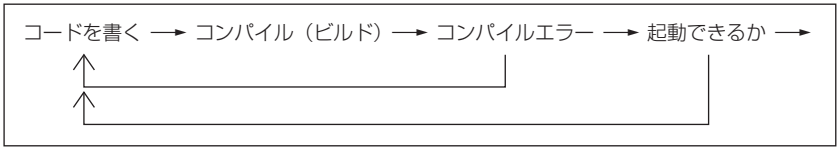

図 1-1 プログラミングプロセス

もかく起動できるか、動くかの確認です。この時点で、そもそも起動できない、クラッシュするなどのバグはテストのフェーズに入る前に取り除きたいものです。

デバッグとテスト

インフォーマルなテスト（起動できるかなど最小限の機能の確認）の後に、テストとデバッグの作業が始まります。

ここで、テストとデバッグについて定義をしておきます。

テストというのはプログラムの動作が仕様どおりであるか確認する作業です。動作が期待する動作（仕様）と異なるとき、それをバグと呼ぶことにします。テストはプログラムのバグを探すプロセスです。

デバッグは、何らかの方法で発見されたバグを修正するプロセスです。

テストとデバッグは全く目的が違うプロセスだということを最初に理解しておきましょう。良いテストとは単位時間あたり、多くのバグを発見するもので、良いデバッグというのは単位時間あたり多くのバグを修正するものです。

バグは正式なテストのプロセスを踏むだけではなく、日常何気なく利用している時に見つかったり、自分以外の利用者によって発見されたり、さまざまな方法で見つかります。

デバッグはいずれにせよバグを認識した時から始まります。

バグだと正式に認定されなくても、なんとなく動きが変だぞとか、単なる問い合わせから確認作業に入る場合もあり、そのような作業も含めて広義にはデバッグと言う立場の人もいますが、本書では、何らかの方法で見つかった（見つけた）バグを修正するプロセスのことをデバッグと言うことにします。

バグを見つけるということ

良いテストは多くのバグを見つけます。テストをするときのこころがまえとして、より多くのバグを見つけたいものです。

バグを見つけたテストのことを成功したテストという立場があります。これは、プログラムの動作としては仕様どおりではないので、「失敗」なのですが、テストとしてはバグを見つけたのですから「成功」であるという立場です（参考文献）。

バグが発見されたと受動態で言うのではなく、バグを発見したと能動態で言うという立

場があります。テストによってバグを（自分が）発見するのであるという立場です。積極的にバグを見つけてやろうという意気ごみが感じられます。

バグは自然発生するのではなくプログラマが自らバグを書くわけですから、受動態ではなく能動態でバグを発見し、直したいものです。

プログラマとしては誰かにバグを発見されるより少しでも前に自らバグを発見したいものですね。

またプログラムのコードを書く以前にテストプログラムを作成するという方法論があります。これを TDD（テスト駆動開発）と呼びます。プログラムをテストするのではなく、すべてのプログラムにはあらかじめテストがあるという方法論です。本書では詳述しませんが、テストとプログラミングそしてデバッグを表裏一体化した開発方法論と言うことができます。

バグの分類

プログラムの動作を下記のように分類します。

① 期待する動作をして終了する。

② 期待する動作をしないで終了する。

③ 終了しない。

ここでプログラムを入力を与えて出力を得るものだと単純化して考えると、テストはいくつかの入力の組に対して出力を確認するプロセスとなります。

①は入力に対応する出力の組が期待する出力（仕様）と等しくかつ終了した場合です。この時点ではバグを見つけていないのでデバッグの必要性はありません。ただし、テストがバグを見つけなかっただけで、バグが存在しないことを保証するものではありません。

②は入力に対しあらかじめ期待する値を出力しない場合です。われわれはバグを発見したことになります。

③は無限ループやデッドロックのような場合で入力に対し期待する値を出さないで終了もしない場合です。

②と③について、われわれはデバッグをすることになります。本書では、そのようなバグをどのようにデバッグするか説明します。

デバッグのプロセス

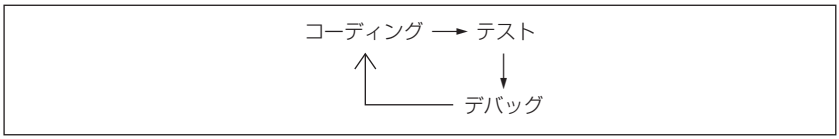

図 1-2 デバッグのプロセス

バグは先に記したようにテストによって発見する場合もあれば、第三者によって発見される場合もあります。デバッグのプロセスは下記のようになります。

① バグの再現

② デバッグ

③ 動作の確認（テスト）

④ 期待する動作の場合は、終了。期待する動作でない場合は、②へ。

本書では、さまざまなバグについてのデバッグ方法について記述しています。

参考文献

『ソフトウェア・テストの技法第 2 版』（マイヤーズ他著、近代科学社刊、978-4-7649-0329-6）

—— Hiro Yoshioka

HACK #2

Debug Hacks マップ

典型的なバグの切り分け方法と本書の Hack との対応を示します。

バグがどのような種類のものなのか、最初は判断が難しいものです。ここでは発生したバグがどのような種類のものなのか分類し、本書のどの Hack が問題解決のヒントとなるのかを示します。

図 1-3 と**図 1-4** では、障害の種類を「異常終了する」、「終了しない」、「その他の現象」に分類し、それぞれについて原因の切り分け目安と関係する Hack 番号を示しています。

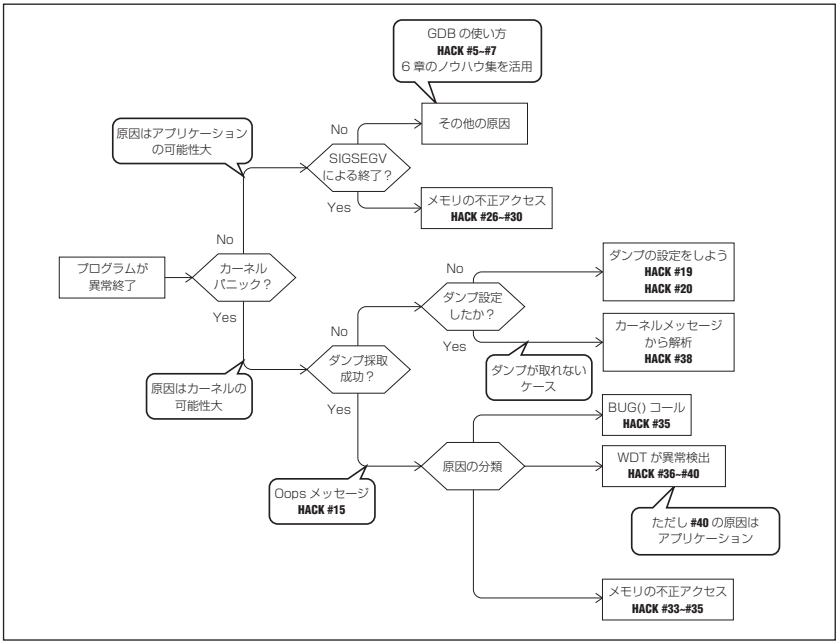

図 1-3　プログラムが異常終了した際の参照 HACK

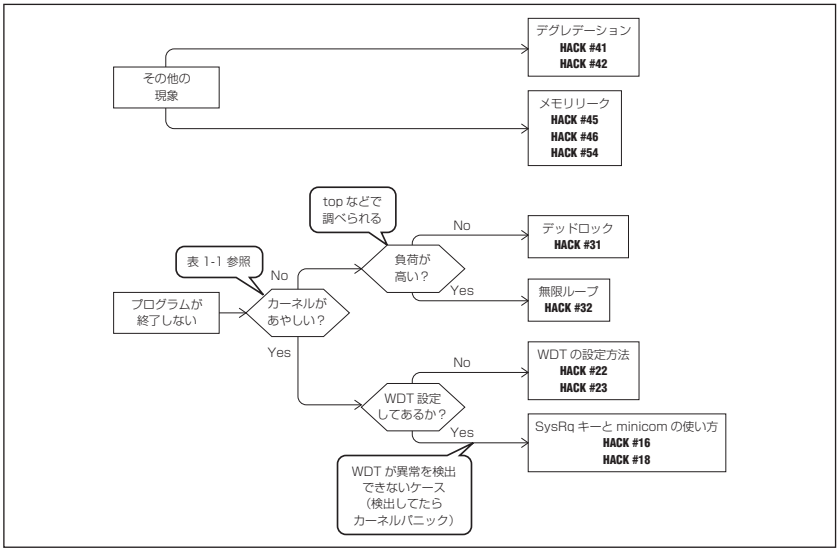

図 1-4　プログラムが終了しない際の参照 HACK

表 1-1 カーネルがあやしいと思われる現象

切り分け方法	結果
ps	表示が途中で止まる ステータスがD
ping	応答が返らない
キーボード	キー入力できない
kill -9	終了させられない
strace	アタッチできない（応答がない）
gdb	アタッチできない（応答がない）
カーネルメッセージを見る	softlockup などのメッセージ出力がある

まとめ

典型的なバグの切り分け方法と本書の Hack との対応を示しました。本 Hack が本書を読み進める上での助けになればと思います。

—— Kazuhiro Yamato、Toyo Abe、Naohiro Ooiwa

HACK #3 デバッグの心得

デバッグ前や解析における心得を紹介します。

何かのバグを見つけた場合は、デバッグの方向性を決定するためにも、情報を集め、現状の把握をしなくてはなりません。

バグの検出はいくつかのパターンがあります。

① テストで検出

② テスト以外で検出

①については確認の意味が大きく、テスト項目に沿って評価をしていることもあると思います。再現でき、また自分で開発したプログラムであれば修正も早いと思います。

②がやっかいなところです。②は自分で検出することもあれば、第三者によって検出される場合もあります。そのためテスト環境と異なった環境で検出されたり、複数の機能を総合した試験や負荷試験などで検出されるバグも多いと思います。

いずれにせよバグを修正するわけですが、本 Hack では難易度の高い②について、以下に心得をまとめます。

1. 再現させる前の心得

2. 再現させたあとの心得
3. 解析時における心得
4. 問題の原因が不明であるときの心得

1. 再現させる前の心得

バグが検出されると、ダンプなどをヒントに再現させるのが重要となります。

ここでは再現させる前の準備や心得について説明します（再現テストプログラム（TP）の作成については「**カーネルパニック（リスト破壊編）**」**[HACK #34]** を参照してください。ダンプ採取については「**diskdump を使ってカーネルクラッシュダンプを採取する**」**[HACK #19]**、「**Kdump を使ってカーネルクラッシュダンプを採取する**」**[HACK #20]** を参照してください）。

環境構築

バグが検出されるとデバッグに入りますが、検証するときにハードウェアやバージョンが違うと動作が異なる場合があります。

再現環境は実機で確認するのがベストですが、できないときにはできる限り問題が発生した環境とそろえます。ハードウェアもソフトウェアもできる限り同じ環境が望ましいです。まずは同じハードウェアを使い、別なハードウェアで再現させる場合でも NIC や CPU の数などできる限り近づけます。ソフトウェアであれば OS、バージョン、パーティションやスワップのサイズなど細かいところまでそろえます。

筆者の経験では以下のような環境をそろえることで再現させることができました。

- マウントやモジュールなどのパラメータ・オプション
- ネットワークの通信先のハードウェア（NIC）
- ディスクの H/W メーカー
- 設定ファイルの記述

試験用の内部ネットワークではバグが再現しないのですが、他の環境だと再現するということがありました。内部ネットワークだと NTP サーバにアクセスできず、それが再現させることができない理由だったことがありました。
また違う例では、/etc/sysconfig/network-script/ifcfg-eth* の記述が同じなのに再現しないということがありました。このときは /etc/sysconfig/network の設定不足が原因でした。

ヒアリング

第三者が検出したバグを再現させる場合に疑問点や必要な情報があればまとめて聞きましょう。細かい質問を何度も聞くと相手が疲れますので、一通り必要な作業をしてから質問、懸念事項をまとめて質問しましょう。また相手がバグには関係ないと思い込み、重要な情報が伝わらないこともあります。このようなことも考慮してヒアリングを行いましょう。

以前に、他の人が作成した再現 TP をもらい実行しましたがなかなか再現しないということがありました。1 週間続けましたが再現せず、最後に今まで数分ごとに実行していたのを、何気なく数秒で実行したところ再現したという経験があります。再現 TP の実行タイミングは確認はしていたのですが、答えていた人と実際に再現 TP を作って実行していた人が違っていて、実行タイミングが数秒だというのが私にまで伝わらなかったようです。ちなみに内容は I/O のキャッシュが問題でした。I/O を初めて行うときにその再現 TP を実行させなければならないというものでした。

設定の見直し

バグの解決を急ぐあまり、設定など簡単なミスをしがちです。再現しない場合は、ケーブルや設定内容を確認し、さらにコマンドの出力でしっかりと確認しましょう。

2. 再現させたあとの心得

再現させたあとの心得について説明します。

現象の確認

再現したように見えて、実は違う現象ということがあります。現象が本当に再現したということを確認します。

再現率、時間の確認

100% 再現するのか、それとも何かタイミングがあったときのみ再現するのかによって、デバッグの手法や時間の使い方が変わります。

また 30 分実施しても再現しないが、10 時間実施すると再現することもあります。再現する時間、確率を把握しておきましょう。

3. 解析時における心得

解析の切り口や、問題の切り分けにおける心得を説明します。

現象を目で確認

例えば何かをするとリブートするとします。何かを実行したあとパニックしてリブートするのか、数十秒後に watchdog によるものなのか確認します。リブート前にはキー操作ができるか、ping が通るのかも確認すると、そのあとのデバッグに役立ちます。再現するまでの時間が毎回同じであれば、その時間も確認します。その値があとのデバッグで役に立つことがよくあります。

シェルで sleep 1 と実行すると明らかに 1 秒以上スリープする現象がありました。time コマンドで計測しても、そもそもカーネルのタイマにバグがあれば、コマンドの出力は参考になりません。現象を実際に確認し、約 5 秒（+4 秒）スリープするということと、毎回同じ秒数であることも確認しました。さらに sleep 10 とすると 10 倍の 50 秒になるのか、それとも 10+4 で 14 秒スリープするのか確認しました。結局 14 秒スリープしており、これが解決のきっかけとなりました。

少し違う例ですが、「コンソールメッセージが画面に表示されない」という質問をよく受けます。このような場合は echo 7 › /proc/sys/kernel/printk と実行してみてください。このファイルで設定されているコンソールレベルよりも低いレベルのメッセージは表示されませんので、気をつけてください。

できるだけ範囲を狭める

3 つの TP（テストプログラム）を実行してバグが発生するのであれば、1 つずつ実行してみます。1 つに絞ることができればデバッグが簡単になりますし、3 つを同時に実行しないと発生しないのであれば、また違う観点でデバッグができます。オプションにより範囲を特定する例は「**アプリケーションのストール（無限ループ編）**」**[HACK #32]** を参照してください。また関係するパラメータがあれば調整しましょう。範囲を限定したり、パラメータを調整することで、再現の時間も短縮できるはずです。パラメータにより再現の確率を上げる例は「**カーネルパニック（レースコンディション編）**」**[HACK #35]** を参照してください。

ファイルシステムのジャーナルに関するバグであれば、mount -o commit=1 でコミットの時間をデフォルトの 5 秒から 1 秒に変更すれば再現しやすくなるかもしれません。また e1000 ドライバのバッファに関するバグであれば ethtool -G ethX rx 64 tx 64 で送受信のバッファサイズを小さくするなど、バグに応じていろいろパラメータを変更してみましょう。

カーネルコンフィグ、カーネルブートパラメータによる問題の切り分け

カーネルのバグであればブートパラメータやカーネルコンフィグを変更して切り分けをします。SMP環境で発生するバグか確認するにはブートパラメータにnosmpと設定すればUPにできます。e1000ドライバに関するバグであればNAPIを無効にすると切り分けることができるかもしれません。

バージョンによる問題の切り分け

オープンソースは常に修正、更新がされています。現在システムで使用されているものより大幅にバージョンが上がっていることもあります。上位のバージョンで発生しないのであれば、あとは差分を調査するだけです。バージョンにより切り分ける例は**「アプリケーションのストール（無限ループ編）」[HACK #32]**、**「カーネルのストール（無限ループ編）」[HACK #36]**を参照してください。逆にバージョンを上げたことでバグが発生した場合は下位のバージョンでどうだったのか再確認します。

他のアプローチも確認

1つの情報だけで判断せずに、他の情報も合わせて確認しましょう。ネットワークであればifconfigコマンドの表示だけではなく、ipコマンド、routeコマンドや/proc/netの情報なども合わせて確認しましょう。コマンドの表示にバグがあることもあります。

事実を元に判断する

現象を見ると、「これが原因の可能性が高い」と思えるときがありますが、根拠がないまま外見だけでそれと決めつけずに、まずはしっかりと確認します。そうしないと本当の原因を見逃してしまいます。解決すると、「まさかこれが原因だったなんて」ということがよくあります。

4. 問題の原因が不明であるときの心得

問題の調査をしていると原因がわからず、行き詰まることもあります。また説明がつかないこともあります。このようなときの心得を説明します。

ハードウェアを疑う

ソフトウェアのデバッグを進めて、どうしても説明のできない現象であったり、条件を変えていないのに毎回違う動作をしたりする場合は、ハードウェアを疑ってみましょう。最初からハードウェアを疑って決めつけると直るものも直りませんが、あまりに不自然な動作をする場合はハードウェアの故障が原因のときもあります。ハードウェアが故障と言っても、電源が入らないようなわかりやすいものからソフトウェアのバグに見えるのもあります。

以前に、突然パーティションが見えなくなるという現象がありました。常にソフトウェアを更新していたため、最初はソフトウェアに問題があると思っていました。しかしまれにパーティションが見えることもありました。そのころ、近くに電子部品が落ちていたことがあり、そのときは何か全然わかりませんでしたが、もしかしてと思いディスクの裏側を見るとその電子部品が取れていました。結局違うディスクに取り替えると全く問題はありませんでした。
これは私ではなく、他の方が経験したものですが、バグのように見えてずっとソフトウェアのデバッグをしていましたが、結局ハードウェアを接続する端子の 1 本が折れていたことがあったそうです。

EDAC（Error Detection And Correction）（bluesmoke）

Linux にはパリティエラーを検出 / 通知する機能として EDAC があります（ストックカーネルにマージされる前は bluesmoke という名称でした）。EDAC はメモリの ECC パリティエラーと PCI バスのパリティエラーを検出します。

メモリの ECC パリティエラーが発生すると、ハードウェアは MCH（Memory Controller Hub）のレジスタにエラーの詳細を示し、NMI 割り込みを上げます。EDAC はこの NMI 割り込みを利用してエラーの検出を行います。EDAC は MCH のレジスタを確認し、1 ビットの訂正可能なエラーであればウォーニングを、2 ビット以上の訂正不可能なエラーであればカーネル内で故意にパニックさせることができます。PCI バスのパリティエラーはポーリングでレジスタを確認し、エラーがあればパニックなどの動作をさせることができます。

sysfs ファイルシステムで設定やエラーの統計情報を取得できます。エラーになったメモリの DIMM 番号もわかります。

詳細はカーネルソースの Documentation/drivers/edac.txt、または Documentation/edac.txt を参照してください。

過去にあった同じようなバグの修正を見つける

原因がわからず、調査のネタがなくなった場合は、同じようなバグが過去にあったのか調べましょう。git や Bugzilla でキーワードにより検索すると、似たような問題で、ヒントになるものがあるかもしれません。

再現しない、原因がつかめない場合

情報が足りず再現しなかったり、そのため原因が解明できない場合もあります。時間切れでこれ以上解析できないということもあると思います。そういう場合はデバッグ情報が出力されるような仕掛けをプログラムに入れておきます。次回同じ現象が発生した場合に

はその情報を元に再現、または原因がわかるようにします。

バグ発生への備え

突然バグが発見された場合、直前に何をしていたかわからないと行き詰まってしまいます。そのためいつバグが発見されてもいいようにしておきます。自動化されたTPであればログを出力するようにして、深夜に実行していたとしてもあとで確認できるようにします。メモリやネットワーク、I/O、CPU使用率など定期的にログの採取をしておくとさらにデバッグしやすくなります。sar、top、free、/proc/meminfo、/proc/slabinfoなど状況によってログを取りましょう。

同僚に説明する

どうしても、原因がつかめない場合、同僚に現象を説明してみます。現象を説明するために、自分なりにまとめる必要がでてきます。そのプロセスの中で、質問をする前に自力で解決することも少なくありません。また、同僚と話すことによって、思いもかけない解決の糸口を得られたり、ヒントを貰ったりすることもあります。ただし、同僚も忙しいですから、なんでもかんでも質問をするのではなく、どうしても解決できない問題に限って相談することにしましょう。

コミュニティに質問する

Linuxのようなオープンソースソフトウェアには、コミュニティがあります。不明な点があれば開発元のメーリングリストに質問してみましょう。世界中でそのソースコードを一番知っている開発者に直接聞けますので一番確かな情報が得られるはずです。有効に活用しましょう。提案やバグフィックスのためなら喜ばれるはずです。コミュニティによる修正は「CPU負荷が高くなる不具合」[HACK #42]で紹介しています。

まとめ

デバッグ前のこころがまえとして注意すべき点や、スムーズに解析を行うための心得を紹介しました。基本的なことではありますが、忙しいときにはおろそかになりがちです。とても大切なことですのでしっかりと身につけておきましょう。

参考文献

- EDAC Project
  http://bluesmoke.sourceforge.net/

—— Naohiro Ooiwa

2章 デバッグ前に知っておくべきこと Hack #4-14

この章では、デバッグの基本として、デバッガ（GDB）の使い方、Intel アーキテクチャの基本、スタックの基礎知識、関数コール時の引数の渡され方、アセンブリ言語の勉強方法などを記しています。

HACK #4 プロセスのコアダンプを採取する

ユーザランドプロセスのコアダンプを取る方法を説明します。

コアダンプを採取することの一番の利点は、問題が発生した時の状態を保存できることです。問題が発生したプログラムの実行ファイルとコアダンプがあれば、その時のプロセスの状態を知ることができます。これは非常に便利な時があります。例えばバグの再現方法がわかっておらず、ごくたまにしか発生しない場合や、特定のマシンでしか発生しない場合などです。そのような場合でも、コアダンプを取ることによって手元に再現環境がなくてもデバッグすることができるからです。

コアダンプを有効化する

多くの Linux ディストリビューションではデフォルトでコアダンプ機能が無効化されています。ulimit コマンドで現在コアダンプ機能が有効化されているかどうか確認できます。

```
$ ulimit -c
0
```

-c オプションはコアファイルのサイズ制限を表示します。上記の例では 0 となっていますので、コアダンプが無効化されている状態です。以下のように ulimit コマンドを実行すると、コアダンプを有効化できます。

```
$ ulimit -c unlimited
```

これはコアファイルのサイズ制限を無制限にするという意味です。無制限にしておけば問題発生時のプロセスのメモリをすべてコアファイルにダンプすることができます。大量のメモリを使用するようなプロセスを扱っている場合、コアファイルの上限サイズを指定したいかもしれません。そのような場合、引数に直接上限サイズを指定します。例えば上限を 1GB とするなら次のように実行します。

```
$ ulimit -c 1073741824
```

有効にしたら、プログラムを実行してみてコアダンプが作成されるか確かめてみましょう。ここで実行したプログラムは、0 番地アクセスをするだけのものです。

```
$ ./a.out
Segmentation fault (core dumped)
```

カレントディレクトリにコアファイルが作成されます。

```
$ file core*
core.7561: ELF 64-bit LSB core file x86-64, version 1 (SYSV), SVR4-style, from './a.out'
```

作成されたコアファイルを使って GDB でデバッグするには次のように GDB を起動します。

```
$ gdb -c core.7561 ./a.out
GNU gdb Fedora (6.8-17.fc9)
...
Core was generated by `./a.out'.
Program terminated with signal 11, Segmentation fault.
[New process 7577]
#0  0x000000000040048c in main () at segfault.c:6
6                       *a = 0x1;
```

segfault.c の 6 行目でシグナル番号 11 を受信しています。gdb の list コマンドで周辺のソースを確認してみます。

```
(gdb) l 5
1       #include <stdio.h>
2
3       int main(void)
4       {
5                       int *a = NULL;
```

```
6                   *a = 0x1;
7                   return 0;
8           }
```

ポインタ a には NULL が入っているので、NULL ポインタ参照でシグナルを受けているのがわかりました。これは非常に単純な例ですが、複雑なプログラムをデバッグする場合でもコアダンプからのデバッグは強力です。プログラムが複雑なほどどこで何をしているときにシグナルを受けたか判断するのが難しくなります。また、再現性が低いとソースコードを追うだけでは原因特定できないケースもあります。このような場合、コアダンプには問題発生時の状態がそのまま保存されているので、原因の特定に役に立ちます。

専用ディレクトリにコアダンプを生成する

大きなシステムを利用している場合、コアファイルを決まった場所に置きたくなります。デフォルトではカレントディレクトリに作成されてしまうので、どこに作成されたのかわかりにくい場合があります。また、コアファイルが大量に作られてしまい、システムのディスク容量を圧迫してしまうこともあります。そのような場合は、コアダンプ専用パーティションを用意して、そこへコアファイルが作成されるように設定すると便利です。sysctl 変数の kernel.core_pattern にダンプ先をフルパスで設定することで変更できます。/etc/sysctl.conf に次のように設定したとします。

```
# cat /etc/sysctl.conf
kernel.core_pattern = /var/core/%t-%e-%p-%c.core
kernel.core_uses_pid = 0
# sysctl -p
```

この状態で先ほどのプログラム a.out を実行すると /var/core/ の下にコアファイルが作成されるようになります。

```
$ ls /var/core/
1223267175-a.out-2820-18446744073709551615.core
```

これは次のようなファイル名になっています。

コアダンプした時刻 - プロセス名 -PID- コアダンプ最大サイズ .core

kernel.core_pattern に指定できる書式指定子を以下の表に示します。

指定子	説明
%%	1つの % 文字
%p	ダンプされたプロセスのプロセス ID（PID）
%u	ダンプされたプロセスの実ユーザ ID（real UID）
%g	ダンプされたプロセスの実グループ ID（real GID）
%s	ダンプを引き起こしたシグナルの番号
%t	ダンプ時刻（1970 年 1 月 1 日 0:00 からの秒数）
%h	ホスト名（uname(2) で返される nodename と同じ）
%e	実行ファイル名
%c	ダンプサイズの上限値（カーネル 2.6.24 から使用可能）

先ほどの例で kernel.core_uses_pid=0 としたのはファイル名に入れる PID の場所を変更したかったからです。これが 1 になっていると、ファイル名末尾に .PID が追加されてしまいます。

ユーザモードヘルパーを使って自動でコアダンプを圧縮する

先ほどの kernel.core_pattern にパイプを記述してユーザモードヘルパーを起動させることもできます。パイプ（|）に続けてコマンドを記述することで利用できます。例えば次のような書式です。

```
# echo "|/usr/local/sbin/core_helper" > /proc/sys/kernel/core_pattern
```

これを応用して自動でコアダンプを圧縮するようにしてみます。

```
# cat /proc/sysctl.conf
kernel.core_pattern = |/usr/local/sbin/core_helper %t %e %p %c
kernel.core_uses_pid = 0
# sysctl -p
```

core_helper の中身は簡単です。

```
$ cat /usr/loca/sbin/core_helper
#!/bin/sh

exec gzip - > /var/core/$1-$2-$3-$4.core.gz
```

この状態でコアダンプさせると /var/core/ の下に圧縮されたコアファイルが作成されるようになります。

```
$ ls /var/core/
1223269655-a.out-2834-18446744073709551615.core.gz
```

システム全体でコアダンプを有効化する

昔は /etc/initscript に ulimit コマンドを実行するスクリプトを書けば良かったのですが、最近のディストリビューションでは /etc/initscript が使えなくなっているものが多いようです。ここでは Fedora9 で確認した手順を紹介します。まずは /etc/profile を編集し、システムにログインするすべてのユーザでコアダンプを有効にします。以下の行でデフォルトは無効化されています。

```
ulimit -S -c 0 > /dev/null 2>&1
```

これを unlimited に変更します。

```
ulimit -S -c unlimited > /dev/null 2>&1
```

次に init スクリプトで起動されるデーモンプロセスについて、コアダンプを有効化します。それには /etc/sysconfig/init ファイルに以下の記述を追加します。

```
DAEMON_COREFILE_LIMIT='unlimited'
```

最後に /etc/sysctl.conf に次の設定を追加します。

```
fs.suid_dumpable=1
```

これは、SUID されたプログラムもコアダンプさせるという設定です。セキュリティのためデフォルトでは無効化されています。システム全体でコアダンプを有効化すると、どのプログラムがどこのディレクトリにコアダンプしたのかわからなくなってしまいます。ですから「専用ディレクトリにコアダンプを生成する」（15 ページ）で紹介した方法で、コアダンプが固定のディレクトリに作成されるようにしましょう。

最後にシステムを再起動すると、システム全体でコアダンプ設定が有効化されます。

コアダンプマスキングを利用して共有メモリをスキップする

大規模なアプリケーションプログラムではプロセスを複数使用し、さらに数ギガバイトに及ぶような、とても大きな共有メモリを使用するものがあります。そのようなアプリケーションのプロセスをコアダンプするとき、すべてのプロセスで共有メモリをダンプしているとディスクを圧迫したり、ダンプによるシステムへの負荷が高くなってしまったり、ダンプに時間がかかってしまうためサービス停止時間が長くなってしまったりというような

デメリットがありました。そこでプロセスごとにコアダンプさせるメモリセグメントを選択する機能がカーネルに実装され、カーネル 2.6.23 以降から利用できるようになっています。またベースカーネルバージョンは異なりますが、RHEL4.7 や RHEL5.2 でも使用できます。共有メモリの内容はそれを共有するプロセス間で同一ですので、全プロセスで同じ内容をダンプする必要はありません。ですので、そのようなアプリケーションであれば、どれかひとつのプロセスで共有メモリをダンプさせるようにし、他プロセスではダンプしないように設定するとよいでしょう。

設定方法は簡単で、/proc/<PID>/coredump_filter を通して行います。coredump_filter はメモリタイプをビットマスクで表しています。

ビットマスク	メモリタイプ
ビット 0	匿名プライベートメモリ
ビット 1	匿名共有メモリ
ビット 2	ファイルを使用するプライベートメモリ
ビット 3	ファイルを使用する共有メモリ
ビット 4	ELF ファイルマッピング（カーネル 2.6.24 以降から使用可能）

筆者の環境では、デフォルトの値は 3 です。すべての匿名メモリセグメントをダンプします。現在の設定は coredump_filter の内容を読むことで確認できます。

```
# cat /proc/<PID>/coredump_filter
00000003
```

すべての共有メモリセグメントをスキップさせるには値を 1 に変更します。

```
# echo 1 > /proc/<PID>/coredump_filter
```

ビット 4 は共有ライブラリや実行ファイルなどの ELF ファイルをマッピングしたメモリセグメントの最初の 1 ページ（x86 では 4KB）をダンプさせるフラグです。どの ELF ファイルがマッピングされていた領域なのかがコアダンプから調べることができるようになります。

まとめ

コアダンプを取るための基本設定と、ダンプ専用ディレクトリやユーザモードヘルパーなど少し変わった使い方を説明しました。また、最近の Linux カーネルで実装されたコアダンプマスキング機能についても説明しました。

参考文献

- Manpage of CORE
  http://www.linux.or.jp/JM/html/LDP_man-pages/man5/core.5.html

—— Toyo Abe

HACK #5 デバッガ（GDB）の基本的な使い方（その 1）

ブレークポイントの設定から実行の継続まで、GDB の基本を説明します。

Linux 環境の定番なデバッガである GDB の基本的な使い方をここでは紹介します。コンパイラの例として gcc を利用することにします。GDB は大変機能が豊富ですが、デバッグのプロセスに沿って解説します。基本は単純です。

その流れは下記のようになります。

(1)　デバッグ対象のプログラムをデバッグオプション付きでコンパイル、ビルドする
(2)　デバッガ（gdb）の起動
(2-1) ブレークポイントの設定
(2-2) スタックフレームの表示
(2-3) 値の表示
(2-4) 実行の継続

準備

gcc の -g オプションによって、デバッグ情報を生成します。

```
$ gcc -Wall -O2 -g ソースファイル
```

Makefile によってビルドしている場合は CFLAGS に -g オプションを与えるのが一般的です。

```
CFLAGS = -Wall -O2 -g
```

configure スクリプトで Makefile を生成している場合は下記のようにします。

```
$ ./configure CFLAGS="-Wall -O2 -g"
```

ビルド方法などは通常 INSTALL ファイル、README ファイルなどに記されているのでそれらを参照してみましょう。

コンパイラにはソースコード上のさまざまなエラーに対してメッセージを出してくれる機能があります。これをウォーニングオプション（Warning Option）と呼びます。これらのメッセージは必ずしもエラーではないのですが、バグを誘発しやすいコーディングを指摘してくれたりします。
コンパイルする際にはすべてのウォーニングないしエラーメッセージが出ないようにコードを綺麗にしておくように心がけましょう。コンパイルエラーはバグの大元です。
-Werror ウォーニングが発生したときに、エラーと同様に扱います。
なおコンパイルエラーが発生するとバイナリは生成されません。

コンパイラ（gcc）の最適化オプションをつけるとソースコードの順番と実際の実行の順番が最適化によって変化する場合があります。そのためデバッガで実行順序を追っているときに、時としてソースコードと違うところを実行していて混乱することがあります。
例えば、インライン化した関数の場合、（関数呼び出しを、その場所で展開して、実際の呼び出しを削除することをインライン化と呼びます）、その関数名でのブレークポイントが設定できなくなります。それはインライン化したために、オブジェクトコードのエントリポイントがなくなって、シンボルテーブルにその関数名が載っていないからです。
最適化によってローカルな変数がレジスタに載っていたりすると、そのローカル変数を表示させることができなくなって、直接レジスタの値を見る必要が出てきます。
このような副作用があるために、デバッグするときは最適化オプションなしでコンパイル、ビルドを勧める人がいますが、これはお勧めできません。
なぜでしょうか。
CとかC++とか手続き型プログラミング言語でプログラムを書くということは、コンパイラという道具だてを利用してコンピュータに対して期待すべき動作を伝えています。コンパイラの最適化オプションの詳細について逐一知る必要はないですが、最適化オプションによってコードの実行の順番がソースコードの順ではない場合があるということくらいは理解しておく必要があります。
理解した上で、実行速度を向上させるために最適化オプションを付けるわけです。わざわざ外す必要はありません。
デバッグのときだけ最適化オプションを外すという場合、最適化オプション付きのバイナリと最適化オプションなしのバイナリの２種類のバイナリを維持管理する必要が出てきます。
管理すべき実体が増えることは管理のコストが増加して、よろしくないです。最適化オプションなしのバイナリで延々デバッグしていたら、実は最適化オプションありのバイナリでは当該バグに遭遇しないとか、そもそも、同じソースからコンパイル、ビルドしたのかをどう管理するのかとか、さまざまなコストが発生します。

２つバイナリを用意すれば間違いなくテストの工数は２倍になるし、管理のコストも増大します。
プログラマは楽をしたがる人種です。なぜ好きこのんで問題を複雑化するのでしょう。テスト、デバッグするバイナリは１つであるべきです。そして出荷するコードが最適化オプション付きのものであれば、当然、最適化オプション付きでテスト、デバッグするというのが正しい姿だと思います。

起動

```
$ gdb 実行ファイル名
```

emacs からの起動の場合は M-x gdb とします。
下記のようなメッセージが表示され、gdb のプロンプトが出ます。

```
Current directory is /home/hyoshiok/work/coreutils/src/
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb)
```

ブレークポイントの設定

ブレークポイントは関数名、行番号などに設定できます。プログラムを実行して、その場所に到着すると自動的に実行を一時停止します。その時点での変数の値の表示、スタックフレームの表示、ブレークポイントの再設定、再実行などなどができます。ブレークポイントコマンド（break）は b と省略できます。

形式：

break ブレークポイント

```
(gdb) b main
Breakpoint 1 at 0x8048e1f: file uname.c, line 184.
```

ブレークポイントとして、関数名、現在のファイル内の行番号、ファイル名を指定して行番号をさらに指定、停止している位置からのオフセット、アドレスなどを指定できます。

形式：

```
break 関数名
break 行番号
break ファイル名 : 行番号
break ファイル名 : 関数名
break +オフセット
break -オフセット
break *アドレス
```

[例]

```
(gdb) b iseq_compile
Breakpoint 2 at 0x81126f6: file compile.c, line 422.
(gdb) b compile.c:516
Breakpoint 3 at 0x8107421: file compile.c, line 516.
(gdb) b +3
Breakpoint 4 at 0x805bd58: file main.c, line 31.
(gdb) b *0x08116fd6
Breakpoint 5 at 0x8116fd6: file iseq.c, line 360.
```

上記の例はそれぞれ、① iseq_compile() 関数、② compile.c の 516 行目、③現時点で停止している位置から 3 行先、④アドレス（0x08116fd6）、にブレークポイントを設定します。

ブレークポイントを指定しないと、次の命令にブレークポイントを設定します。

```
(gdb) b
Breakpoint 6 at 0x805bd44: file main.c, line 28.
```

設定したブレークポイントは info break で確認できます。

```
(gdb) info break
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x081126f6 in iseq_compile at compile.c:422
3       breakpoint     keep y   0x08107421 in iseq_translate_threaded_code at compile.c:516
4       breakpoint     keep y   0x0805bd58 in main at main.c:31
5       breakpoint     keep y   0x08116fd6 in iseq_load at iseq.c:360
6       breakpoint     keep y   0x0805bd44 in main at main.c:28
```

実行

run コマンドで実行を開始します。引数がない場合は、run だけになります。ブレークポイントを設定している場合は、そこで一時停止します。r と省略できます。

形式：

run 引数

```
(gdb) run -a
Starting program: /home/hyoshiok/work/coreutils/src/uname -a

Breakpoint 1, main (argc=2, argv=0xbf9cd714) at uname.c:184
```

main() にブレークポイントを設定し、main() まで実行させるということはよくあります。それと同様の結果をもたらすのが、start コマンドです。

形式：

start

スタックフレームの表示

バックトレース（backtrace）コマンドによってブレークポイントによって一時停止したときのスタックフレームを表示できます。backtrace コマンドは bt と省略できます。また、where と info stack（info s と省略可能です）は backtrace コマンドの別名として利用できます。

形式：

backtrace
bt
　すべてのバックトレースを表示します。

backtrace N
bt N
　最初の N 個のフレームだけバックトレースを表示します。

backtrace -N
bt -N
　最後の N 個のフレームだけバックトレースを表示します。

backtrace full
bt full
backtrace full N
bt full N

backtrace full -N

bt full -N

バックトレースだけではなくローカル変数も表示します。Nは前述したとおり、最初（ないし最後）のN個のフレームの表示になります。

[例]

```
Breakpoint 2, vm_exec_core (th=0x0, initial=0) at vm_exec.c:86
(gdb) bt
#0  vm_exec_core (th=0x0, initial=0) at vm_exec.c:86
#1  0x08107421 in iseq_translate_threaded_code (iseq=0x977dcf0) at compile.c:510
#2  0x08107ac5 in iseq_setup (iseq=0x977dcf0, anchor=0xbfd5f01c) at compile.c:963
#3  0x081127c7 in iseq_compile (self=158469340, node=0x0) at compile.c:501
#4  0x081175f2 in rb_iseq_new_with_bopt_and_opt (node=0x0, name=158469360, filename=158469360, parent=0,
type=3, bopt=0, option=0x81a6ac0) at iseq.c:329
#5  0x081179e5 in rb_iseq_new (node=0x0, name=158469360, filename=158469360, parent=0, type=3) at iseq.
c:306
#6  0x08127148 in Init_VM () at vm.c:1864
#7  0x0806a595 in rb_call_inits () at inits.c:55
#8  0x0805e4d5 in ruby_init () at eval.c:65
#9  0x0805bd77 in main (argc=4, argv=0xbfd5f264) at main.c:34
(gdb)
```

[例：最初の３つのフレームを表示]

```
(gdb) bt 3
#0  vm_exec_core (th=0x0, initial=0) at vm_exec.c:86
#1  0x08107421 in iseq_translate_threaded_code (iseq=0x977dcf0) at compile.c:510
#2  0x08107ac5 in iseq_setup (iseq=0x977dcf0, anchor=0xbfd5f01c) at compile.c:963
(More stack frames follow...)
```

[例：外側から３個のスタックフレームと、それのローカル変数の表示]

```
(gdb) bt full -3
#7  0x0806a595 in rb_call_inits () at inits.c:55
No locals.
#8  0x0805e4d5 in ruby_init () at eval.c:65
        _th = (rb_thread_t * const) 0x9711758
        _tag = {buf = {{__jmpbuf = {-1076497952, 135679824, 134593648, -1076498024, 516014545,
-1092008770}, __mask_was_saved = 0, __saved_mask = {__val = {134537212, 3086358120, 3218469200,
3086296411, 3086358560, 3086027816, 1, 1, 0, 134561536, 188, 3218469192, 3085631476, 3218469208,
3085102259, 0, 3218469224, 3085102424, 3, 8388608, 8388608, 4294967295, 0, 134593648, 3218469272,
```

```
135462041, 8388608, 0, 5, 0, 8388608, 0}}}}, tag = 0, retval = 4294967295, prev = 0x0}
        state = 0
        initialized = 1
#9  0x0805bd77 in main (argc=4, argv=0xbfd5f264) at main.c:34
        variable_in_this_stack_frame = 135679849
```

スタックフレームを表示することによって、どこでプログラムが一時停止しているか（ブレークポイント）ということとプログラムの呼び出し経路がわかります。

変数の表示

print コマンドで変数の表示をします。print は p と省略できます。

形式：

print 変数

```
(gdb) p argv
$1 = (char **) 0xbf9cd714
(gdb) p *argv
$2 = 0xbf9cf6a5 "/home/hyoshiok/work/coreutils/src/uname"
(gdb) p argv[0]
$3 = 0xbf9cf6a5 "/home/hyoshiok/work/coreutils/src/uname"
(gdb) p argv[1]
$4 = 0xbf9cf6cd "-a"
(gdb)
```

この例では argv[] を表示しています。argv[0] には実行ファイル名（"/home/hyoshiok/work/coreutils/src/uname"）が、argv[1] には最初のオプション（"-a"）が入っていることがわかります。

レジスタの表示

info registers でレジスタを表示します。info reg と省略できます。

```
(gdb) info reg
eax            0x61    97
ecx            0x0     0
edx            0xb7f140f8          -1208925960
ebx            0xbf9cd714          -1080240364
esp            0xbf9cd4a0          0xbf9cd4a0
ebp            0xbf9cd678          0xbf9cd678
```

```
esi            0x0    0
edi            0x2    2
eip            0x8048ebd          0x8048ebd <main+173>
eflags         0x200213           [ CF AF IF ID ]
cs             0x73   115
ss             0x7b   123
ds             0x7b   123
es             0x7b   123
fs             0x0    0
gs             0x33   51
```

各レジスタについては、レジスタ名に $ を付けることで表示できます。

```
(gdb) p $eax
$8 = 97
```

表示するときのフォーマットとして下記が利用できます。

形式：

p/ フォーマット 変数

フォーマット	説明
x	16 進数で表示
d	10 進数で表示
u	符号なしの 10 進数で表示
o	8 進数で表示
t	2 進数で表示。t は two からくる。
a	アドレス
c	文字（ASCII）として表示
f	浮動小数点
s	文字列として表示
i	（メモリを表示する x コマンドのみ利用できる）機械命令を表示

```
(gdb) p/c $eax
$7 = 97 'a'
```

10 進数の 97 は ASCII 文字で 'a' となります。

プログラムポインタは $pc でも $eip でも表示することができます。Intel IA-32 アーキテクチャではプログラムポインタ名は eip だからですね。

```
(gdb) p $pc
$9 = (void (*)()) 0x8048ebd <main+173>
(gdb) p $eip
$10 = (void (*)()) 0x8048ebd <main+173>
```

メモリの中身を表示するには x コマンドを使います。x は eXamining からきている名前です。

形式：

x/ フォーマット アドレス

```
(gdb) x $pc
0x8048ebd <main+173>:        0x0f6ef883
(gdb) x/i $pc
0x8048ebd <main+173>:        cmp    $0x6e,%eax
```

ここで x/i は機械命令として表示するということです。

一般的には x コマンドは x/NFU ADDR という形式です。ここで ADDR は表示したいアドレスです。N は何回繰り返すか、F は先に示した表示のフォーマット（x、d、u、o、t、a、c、f、s、i）、U は下記の単位です。

単位	説明
b	バイト
h	ハーフバイト（2 バイト）
w	ワード（4 バイト）デフォルト
g	ジャイアントバイト（8 バイト）

下記は pc が示す番地から 10 命令（i）表示します。

```
(gdb) x/10i $pc
0x8048ebd <main+173>: cmp    $0x6e,%eax
0x8048ec0 <main+176>: je     0x8049048 <main+568>
0x8048ec6 <main+182>: jg     0x8048f62 <main+338>
0x8048ecc <main+188>: cmp    $0x61,%eax
0x8048ecf <main+191>: nop
0x8048ed0 <main+192>: je     0x8049055 <main+581>
0x8048ed6 <main+198>: jg     0x8048f90 <main+384>
0x8048edc <main+204>: cmp    $0xffffff7d,%eax
```

```
0x8048ee1 <main+209>: je     0x8048fe8 <main+472>
0x8048ee7 <main+215>: cmp    $0xfffffff7e,%eax
```

逆アセンブルするコマンドもあります。disassemble です。disas と省略できます。

形式：

① disassemble
② disassemble プログラムカウンタ
③ disassemble 開始アドレス 終了アドレス

①の形式は、現在の関数全体を逆アセンブルします。②の形式は、プログラムカウンタの値を含む関数全体を逆アセンブルします。③の形式は、開始アドレスから終了アドレス未満のアドレスまで逆アセンブルします。

```
(gdb) disassem $pc $pc+50
Dump of assembler code from 0x8048ebd to 0x8048eef:
0x08048ebd <main+173>: cmp    $0x6e,%eax
0x08048ec0 <main+176>: je     0x8049048 <main+568>
0x08048ec6 <main+182>: jg     0x8048f62 <main+338>
0x08048ecc <main+188>: cmp    $0x61,%eax
0x08048ecf <main+191>: nop
0x08048ed0 <main+192>: je     0x8049055 <main+581>
0x08048ed6 <main+198>: jg     0x8048f90 <main+384>
0x08048edc <main+204>: cmp    $0xfffffff7d,%eax
0x08048ee1 <main+209>: je     0x8048fe8 <main+472>
0x08048ee7 <main+215>: cmp    $0xfffffff7e,%eax
0x08048eec <main+220>: lea    0x0(%esi,%eiz,1),%esi
End of assembler dump.
```

プログラムの実行を任意の場所で停止し、任意の変数、アドレスなどを上記のようにして自由に表示することができます。期待する値であるか確認することによって、バグの存在を確認できます。

ステップ実行

ソースコードに沿って、一行一行実行することをステップ実行と言います。

ソースコード一行ごとの実行は next（n と略す）コマンドで行えます。実行するものが関数などの場合、その関数の中も実行したい場合があります。そのときは、step（s と略す）コマンドで行います。

例えば、下記の例で print_element (name.sysname) で実行が停止していたとします。step コマンドでは、print_element () 関数内に入りこんで実行しますが、next コマンドの場合は、print_element () 関数を実行後、次の行（if (toprint & PRINT_NODENAME)）で実行が停止します。

```
if (toprint & PRINT_KERNEL_NAME)
  print_element (name.sysname);
if (toprint & PRINT_NODENAME)
```

next コマンドも step コマンドもソースコードの一行ごとに実行を一時停止します。一命令（アセンブリ命令）ごとの実行をしたい場合は、それぞれ nexti コマンドないし stepi コマンドを利用します。

nexti コマンドは関数の中に入って実行しませんが、stepi は関数の中に入って実行します。

実行の再開

デバッグ中のプログラムの実行を再開する場合、continue（c と略す）コマンドを利用します。ブレークポイントに到達すれば再び停止します。到達しなければ実行を終了するかなどします。

形式：

```
continue
continue 回数
```

回数を指定すると回数分ブレークポイントを無視します。例えば continue 5 とすれば 5 回停止せず、6 回目のブレークポイントに達した時点で停止します。

デバッグ対象のプログラムは通常、

① 実行が正常に終了する
② 何らかの原因で異常終了（コアダンプ、アクセス違反などなど）する
③ 実行が停止しない（無限ループなど）
④ 実行がハング（ストール、デッドロックなど）する

のいずれかの場合になります。

正常終了以外は、再度実行を最初から実行し原因を究明する（デバッグ）必要があります。

ウォッチポイント

大規模なソフトウェアや、ポインタを多用しているようなプログラムの場合、変数がどこで変更されているか容易にわからない場合があります。変数の実行時の変更場所を簡単に見つける方法としてwatch（ウォッチポイント）コマンドがあります。

形式：

watch ‹式›

‹式›が変更された時、実行を一時停止します。

ここで‹式›というのは定数や変数などです。

形式：

awatch ‹式›

‹式›が参照、変更された時、実行を一時停止します。

形式：

rwatch ‹式›

‹式›が参照された時、実行を一時停止します。

[例]

```
(gdb) awatch short_output
Hardware access (read/write) watchpoint 3: short_output
(gdb) c
Continuing.
Hardware access (read/write) watchpoint 3: short_output

Old value = false
New value = true
main (argc=1, argv=0xbfbf8924) at who.c:783
```

変数（short_output）の値が変化したところで実行が一時停止します。

ウォッチポイントの設定をすると実行速度が低下する場合がありますので注意が必要です。

ブレークポイント、ウォッチポイントの削除

delete（dと略す）コマンドで削除します。

形式：

delete ‹番号›

‹番号›で示されるブレークポイントないしウォッチポイントを削除します。

[例]

```
(gdb) info b
Num     Type           Disp Enb Address    What
2       watchpoint     keep y              assumptions
3       acc watchpoint keep y              short_output
          breakpoint already hit 1 time
(gdb) delete 2   ←2番目のウォッチポイントを削除
(gdb) info b
Num     Type           Disp Enb Address    What
3       acc watchpoint keep y              short_output
          breakpoint already hit 1 time
```

その他のブレークポイント

ハードウェアブレークポイント（hbreak）。ROM 領域にプログラムがあるなど当該メモリを変更できない場合に利用します。アーキテクチャによっては利用できない場合があります。

一時ブレークポイント（tbreak）、一時ハードウェアブレークポイント（thbreak）。ブレークポイント（ハードウェアブレークポイント）と同様にブレークポイントに達すると実行を一時停止しますが、そのブレークポイントを解除する点が異なります。一回だけ停止したい場合などは便利です。

一時ウォッチポイントというのは残念ながらありません。

変数の値の変更

形式：

set variable ‹変数›=‹式›

[例]

```
(gdb) p options
$7 = 1
(gdb) set variable options = 0
(gdb) print options
$8 = 0
```

変数（options）の値を 0 に変更してみました。

実行時に自由に変数の値を変えられるので、ソースコードを変更する前に、いろいろの値で確認することができます。

コアファイルの生成

generate-core-file コマンドによって、デバッグしているプロセスのコアファイルを生成することができます。

[例]

```
(gdb) generate-core-file
Saved corefile core.13163
```

コアファイルとデバッグ対象があれば後に、そのコアを生成した時点での実行履歴（レジスタやメモリの値）を確認することができます。

また、コマンドラインから直接コアファイルを生成する gcore というコマンドがあります。

```
$ gcore `pidof emacs`
```

起動中のプログラムを終了しないでコアファイルが取得できますので、原因究明を別のマシンで独立に行うなど、客先で発生した問題を分析するときに利用できます。

まとめ

Linux 環境の定番なデバッガである GDB の基本的な使い方をここでは紹介しました。デバッガ利用の準備から、起動、ブレークポイントの設定、スタックフレームの表示、値の表示、実行の継続など、デバッグプロセスの基本について記しました。

参考文献

- GDB：The GNU Project Debugger
  http://sources.redhat.com/gdb/
  http://sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html

—— Hiro Yoshioka

HACK #6 デバッガ（GDB）の基本的な使い方（その 2）

GDB のちょっと便利な使い方を紹介します。

Linux 環境の定番デバッガである GDB の基本的な使い方に続いてちょっと便利な使い方をここでは紹介します。

プロセスへのアタッチ（attach）

デーモンプロセスのようにすでに起動しているプロセスをデバッグしたい場合や、プログラムが無限ループに入ってしまって、端末に制御が戻らない等のデバッグをしたい場合があります。そのような時は attach コマンドを利用します。

形式：

```
attach pid
```

プロセス ID が pid のプロセスにアタッチします。

プロセス ID を調べるのは ps コマンドを利用します。ここでは sleep コマンドをデバッグする例を示しています。

```
$ ps aux|grep sleep
hyoshiok 17315  0.0  0.3   8984  5840 pts/4    Ss+  13:33   0:00 /usr/bin/gdb --annotate=3 sleep
hyoshiok 17606  0.0  0.0   2792   620 pts/2    T+   13:41   0:00 ./sleep 100
hyoshiok 17895  0.0  0.0   3044   808 pts/1    S+   13:50   0:00 grep sleep
```

左から 2 つ目の欄の数字がプロセス ID（pid）です。この例では 17606 が当該 pid です。gdb からアタッチするには下記のようにすればできます。

```
(gdb) attach 17606
Attaching to program: /home/hyoshiok/work/coreutils-6.10/build-tree/coreutils-6.10/src/sleep, process
17606
`system-supplied DSO at 0xb801a000' has disappeared; keeping its symbols.
Loaded symbols for /lib/tls/i686/cmov/libc.so.6
Loaded symbols for /lib/ld-linux.so.2
0xb803d430 in __kernel_vsyscall ()
(gdb) bt
#0  0xb803d430 in __kernel_vsyscall ()
#1  0x410bbdc0 in __nanosleep_nocancel () from /lib/tls/i686/cmov/libc.so.6
#2  0x0804a1ca in xnanosleep (seconds=100) at xnanosleep.c:112
```

```
#3  0x08048fd1 in main (argc=2, argv=Cannot access memory at address 0x4) at sleep.c:147
(gdb)
```

bt コマンドで、バックトレース（スタックフレーム）を表示すれば、どのように呼び出されてプログラムが待ちに入っているか理解できます。sleep コマンドの例では xnanosleep() から呼び出した __nanosleep_nocancel () が呼んでいるシステムコールで待っているというのがバックトレースから読み取れます。

ソースコードで確認してみると、確かに xnanosleep() から nanosleep() が呼び出されることがわかります。

```
xnanosleep (double seconds)
{
…
      errno = 0;
      if (nanosleep (&ts_sleep, NULL) == 0)
       break;
```

この例はバグを含んでいるわけではないですが、バックトレースを確認することによって無限ループや何らかの原因で待ちが発生している場合のデバッグをすることができます。

待ちの原因としては、入出力、システムコールなどを呼び出したことによるブロック、ロックの取得の待ちなどが考えられます。

無限ループは、ある条件が真になるまで延々と繰り返す、スピンロックのようなものから単にロジック上のバグ（絶対真にならないような条件を指定したバグ）などさまざまなものが考えられます。

アタッチした後は、通常の gdb のコマンドが利用できますので、print コマンドで変数の表示をしたり、ブレークポイントを設定したりできます。

またプログラムの実行を再開したい場合は通常の continue コマンド（c と略す）を利用することができます。

動作の確認を終了し、gdb から切り離したい場合は detach コマンドを利用します。detach するとデバッグしていたプロセスは gdb の制御下から解放されます。なおプロセスが detach されるとそのプロセスは実行を継続します。

プロセスの情報は info proc コマンドで表示できます。

```
(gdb) info proc
process 17606
cmdline = './sleep'
cwd = '/home/hyoshiok/work/coreutils-6.10/build-tree/coreutils-6.10/src'
exe = '/home/hyoshiok/work/coreutils-6.10/build-tree/coreutils-6.10/src/sleep'
```

条件付きブレークポイント

ある条件の時のみ停止するという条件付きブレークポイントというのがあります。

形式：

break ブレークポイント if 条件

これは、与えられた条件を評価し、真の場合に実行を停止します。

[例]

```
(gdb) b iseq_compile if node==0
Breakpoint 1 at 0x81126f6: file compile.c, line 422.
(gdb) run -e ' p 1'
Starting program: /home/hyoshiok/work/ruby_trunk/ruby/ruby -e ' p 1'
[Thread debugging using libthread_db enabled]
[New Thread 0xb80df6b0 (LWP 10586)]
[Switching to Thread 0xb80df6b0 (LWP 10586)]

Breakpoint 1, iseq_compile (self=166726860, node=0x0) at compile.c:422
```

形式：

condition ブレークポイント番号

condition ブレークポイント番号 条件

ブレークポイント番号で指定したブレークポイントに条件を追加、削除します。最初の構文は、ブレークポイント番号に対応するブレークポイントの条件を削除し、2 番目の構文は、ブレークポイントに条件を追加します。

実行の繰り返し

形式：

ignore ブレークポイント番号 回数

ブレークポイント番号で示されたブレークポイント、ウォッチポイント、キャッチポイントを回数で指定した回数分、無視します。

continue コマンドにも ignore コマンド同様、回数を指定できます。これは指定した回数回ブレークポイントで停止しないことと同義です。

形式：

```
continue 回数
step 回数
stepi 回数
next 回数
nexti 回数
```

それぞれ continue 回数、step 回数、stepi 回数、next 回数、nexti 回数の指定した回数分コマンドを繰り返します。

形式：

```
finish
until
until アドレス
```

finish コマンドは、現在実行している関数が終了するまで実行します。until コマンドは、現在実行している関数等が終了するまで実行します。ループなどの実行では、ループを終了した後まで実行します。ループ内から抜け出すために利用します。

ブレークポイントの削除と無効化

定義したブレークポイントを削除する場合は clear コマンドを使います。定義を残したまま、一時的にブレークポイントを無効化する場合は disable コマンドを使います。無効化したブレークポイントを有効化するのには enable コマンドを使います。

形式：

```
clear
clear 関数名
clear 行番号
clear ファイル名 : 行番号
clear ファイル名 : 関数名
delete [breakpoints] ブレークポイント番号
```

形式：

```
disable [breakpoints]
disable [breakpoints] ブレークポイント番号
```

disable display ディスプレイ番号
disable mem メモリ領域

ブレークポイント番号が指定されていなければすべてのブレークポイントを無効化します。ブレークポイントを指定していれば、そのブレークポイントを無効化します。3 番目の構文は、display コマンドで定義した自動表示の設定を無効化します。4 番目の構文は mem コマンドで定義したメモリ領域を無効化します。

breakpoints というキーワードは省略できます。

形式：

enable [breakpoints]
enable [breakpoints] ブレークポイント番号
enable [breakpoints] once ブレークポイント番号
enable [breakpoints] delete ブレークポイント番号
enable display ディスプレイ番号
enable mem メモリ領域

ブレークポイント等を有効化します。once という構文は指定したブレークポイントを一度だけ有効化します。すなわち、そのブレークポイントでプログラムの実行を停止したときに、そのブレークポイントを無効化します。delete という構文は、実行を停止したときに、そのブレークポイントを削除するところが once の動作と異なります。

ブレークポイントコマンド

ブレークポイントコマンド（commands）はブレークポイントで停止した時に自動的に実行するコマンドを定義できます。

形式：

```
commands ブレークポイント番号
  コマンド
  …
end
```

ブレークポイント番号で指定したブレークポイントで停止したとき、自動的にコマンドを実行します。下記の例では、ブレークポイントで停止したとき、p *iseq（iseq をプリント）します。

```
(gdb) b 425
Breakpoint 2 at 0x811271a: file compile.c, line 425.
(gdb) command 2
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
>p *iseq
>end
(gdb) c
Continuing.
[New Thread 0xb80eab90 (LWP 10836)]

Breakpoint 1, iseq_compile (self=166714140, node=0x0) at compile.c:422
(gdb) c
Continuing.

Breakpoint 2, iseq_compile (self=166714140, node=0x0) at compile.c:425
$4 = {type = 3, name = 166714160, filename = 166714160, iseq = 0x0, iseq_encoded = 0x0, iseq_size = 0,
mark_ary = 166714120, coverage = 0, insn_info_table = 0x0, insn_info_size = 0, local_table = 0x0, local_
table_size = 0, local_size = 0, argc = 0, arg_simple = 0, arg_rest = -1, arg_block = -1, arg_opts = 0,
arg_post_len = 0, arg_post_start = 0, arg_size = 0, arg_opt_table = 0x0, stack_max = 0, catch_table =
0x0, catch_table_size = 0, parent_iseq = 0x0, local_iseq = 0x9f662d0, self = 166714140, orig = 0, cref_
stack = 0x9efdacc, klass = 0, defined_method_id = 0, compile_data = 0x9f59170}
(gdb) info b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x081126f6 in iseq_compile at compile.c:422
          stop only if node==0
          breakpoint already hit 2 times
2       breakpoint     keep y   0x0811271a in iseq_compile at compile.c:425
          breakpoint already hit 1 time
        p *iseq
```

またコマンドの最初の1行目がsilentコマンドの場合、ブレークポイントで停止した時のメッセージを表示しません。独自のメッセージを出力する時などに便利です。

前述の条件付きブレークポイントと組み合わせればブレークポイントで停止した時に複雑な表示などをすることが可能です。

```
break foo if x>0
  commands
```

```
    silent
    printf "x is %d\n", x
    cont
  end
```

よく使うコマンドとその省略形（エイリアス）

コマンド名は下記のように省略できます。他のコマンド名と重複がない限り省略することもできます。コマンドラインモードで利用している場合はタブを入力することによって、自動的に gdb が補完してくれます。

表 2-1　コマンドと省略形

コマンド	省略形	説明
よく使うコマンド		
backtrace	bt、where	バックトレースの表示
break		ブレークポイントの設定
continue	c、cont	実行の再開
delete	d	ブレークポイントの削除
finish		関数を終了するまで実行
info breakpoints		ブレークポイント情報の表示
next	n	次の 1 行まで実行
print	p	式の表示
run	r	プログラムの実行
step	s	次の一行まで実行、関数内にも入る
x		メモリ内容表示
until	u	指定行まで実行
その他のコマンド		
directory	dir	ディレクトリの挿入
disable	dis	ブレークポイントの無効化
down	do	呼び出しているフレームの選択表示
edit	e	ファイル、関数の編集
frame	f	フレームの選択表示
forward-search	fo	前方検索
generate-core-file	gcore	コアファイルの生成
help	h	コマンド一覧の表示
info	i	情報の表示
list	l	関数または行の表示
nexti	ni	次の 1 行まで実行（アセンブリ単位）

表 2-1 コマンドと省略形（続き）

コマンド	省略形	説明
その他のコマンド		
print-object	po	オブジェクトの情報を表示
sharedlibrary	share	共有ライブラリのシンボルのロード
stepi	si	1 命令の実行

info コマンドはデバッグ対象に対するさまざまな情報を表示します。一方で show コマンドは gdb 内部の機能、変数、オプションなどの情報を表示します。

まとめ

gdb のちょっと便利な使用方法について記しました。

参考文献

- GDB：The GNU Project Debugger
  http://sources.redhat.com/gdb/
  http://sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html

—— Hiro Yoshioka

HACK #7 デバッガ（GDB）の基本的な使い方（その 3）

ヒストリ、初期化ファイル、コマンド定義などを説明します。

Linux 環境の定番デバッガである GDB のちょっと便利な使い方を引き続き紹介します。

値ヒストリ

print コマンドで表示した値を値ヒストリとして内部で記録しています。その値を他の式で利用することができます。

```
(gdb) p argc
$1 = (int *) 0xbf926e00
(gdb) p *argc
$2 = 1
```

最後の値は $ で参照することができます。

```
(gdb) p $
$3 = 1
```

show value コマンドで、ヒストリ内の最後の 10 個の値を表示します。

```
(gdb) show value
$1 = (int *) 0xbf926e00
$2 = 1
$3 = 1
```

変数	説明
$	値ヒストリの最後の値
$n	値ヒストリの n 番目の値
$$	値ヒストリの最後の値の 1 つ前の値
$$n	値ヒストリの最後のから数えて n 番目の値
$_	x コマンドによって最後に調査したアドレス
$__	x コマンドによって最後に調査したアドレスの値
$_exitcode	デバッグしているプログラムの終了コード
$bpnum	最後に設定したブレークポイント番号

変数

自由に変数を定義できます。変数名は $ で始まる英数字です。

```
(gdb) set $i=0
(gdb) p $i
$1 = 0
```

コマンドヒストリ

コマンドヒストリをファイルに保存できます。コマンドヒストリを保存しておけば、デバッグのセッションを跨がったコマンドの再利用（矢印キーを利用して過去のコマンドを遡る）などができて便利です。コマンドヒストリファイルのデフォルトは ./.gdb_history です。

```
(gdb) show history
expansion:  History expansion on command input is off.
filename:  The filename in which to record the command history is "/home/hyoshiok/work/dbg/hyoshiok/
chapter1/.gdb_history".
save:  Saving of the history record on exit is on.
size:  The size of the command history is 256.
```

形式：

set history expansion
show history expansion

csh スタイルの！文字を使用します。

形式：

set history filename ファイル名
show history filename

コマンドヒストリをファイル名に保存します。環境変数 GDBHISTFILE によりデフォルトのファイル名は変更できます。

形式：

set history save
show history save

コマンドヒストリのファイルへの保存、復元を可能にします。

形式：

set history size 数
show history size

コマンドヒストリに保存するコマンド数を設定します。デフォルトは 256。

初期化ファイル（.gdbinit）

初期化ファイルは Linux 環境では .gdbinit というファイル名です。gdb の起動に先だって、.gdbinit ファイルがあれば、それをコマンドファイルとして実行します。初期化ファイルやコマンドファイルの実行の順番は下記のとおりです。

① $HOME/.gdbinit
② コマンドラインオプションの実行
③ ./.gdbinit
④ -x オプションで与えられているコマンドファイル

初期化ファイルの構文はコマンドファイルの構文と同じで、gdb のコマンドを書き記し

ます。

コマンド定義

define コマンドで、ユーザ定義のコマンドを作成できます。また document コマンドで作成したコマンドの説明を記述できます。help コマンド名で、定義したコマンドの説明を参照できます。

形式：

```
define コマンド名
  コマンド
  …
end
```

形式：

```
document コマンド名
  説明
end
```

形式：

```
help コマンド名
```

下記の例は、li というコマンドを定義しています。それは現在の $pc が示すアドレスから 10 命令を表示します。また document コマンドによって、li コマンドの説明（list machine instruction）を定義しています。それは help li で参照できます。

```
define li
  x/10i $pc
end
document li
  list machine instruction
end
```

[実行例]

```
(gdb) start
Breakpoint 1 at 0x805bd04: file main.c, line 28.
Starting program: /home/hyoshiok/work/ruby_trunk/ruby/ruby
```

```
[Thread debugging using libthread_db enabled]
[New Thread 0xb800e6b0 (LWP 8116)]
[Switching to Thread 0xb800e6b0 (LWP 8116)]
main (argc=1, argv=0xbfe23384) at main.c:28
28          setlocale(LC_CTYPE, "");
(gdb) li
0x805bd04 <main+20>:  movl   $0x8179826,0x4(%esp)
0x805bd0c <main+28>:  movl   $0x0,(%esp)
0x805bd13 <main+35>:  call   0x805b608 <setlocale@plt>
0x805bd18 <main+40>:  lea    0x4(%ebx),%eax
0x805bd1b <main+43>:  mov    %eax,0x4(%esp)
0x805bd1f <main+47>:  mov    %ebx,(%esp)
0x805bd22 <main+50>:  call   0x80d4390 <ruby_sysinit>
0x805bd27 <main+55>:  lea    -0xc(%ebp),%eax
0x805bd2a <main+58>:  mov    %eax,(%esp)
0x805bd2d <main+61>:  call   0x812f890 <ruby_init_stack>
(gdb) help li
list machine instruction
```

初期化ファイルだけではなく、各種設定などをファイルに登録して、デバッガ実行時に読み込んで実行することができます。

形式：

source ファイル名

gdb のコマンドファイルに libm.so で定義されている関数を利用して、簡単な関数電卓を作ったブログがありました。実行例は下記です（引用文献①）。

```
(gdb) start
Breakpoint 1 at 0x805bd04: file main.c, line 28.
Starting program: /home/hyoshiok/work/ruby_trunk/ruby/ruby
[Thread debugging using libthread_db enabled]
[New Thread 0xb7f486b0 (LWP 9718)]
[Switching to Thread 0xb7f486b0 (LWP 9718)]
main (argc=1, argv=0xbfa5cfb4) at main.c:28
28          setlocale(LC_CTYPE, "");
(gdb) source gdbcalc
(gdb) p $log10(10000.0)
$1 = 4
```

```
(gdb) p $log2(1024.0)
$2 = 10
```

gdbcalc ファイルは下記のようになっています。

```
#
# 下記のブログからの引用
# http://www.keshi.org/blog/2006/03/gdb_hacks_gdbcalc.html
#
set $e = 2.7182818284590452354
set $pi = 3.14159265358979323846
set $fabs = (double (*)(double)) fabs
set $sqrt = (double (*)(double)) sqrt
set $cbrt = (double (*)(double)) cbrt
set $exp = (double (*)(double)) exp
set $exp2 = (double (*)(double)) exp2
set $exp10 = (double (*)(double)) exp10
set $log = (double (*)(double)) log
set $log2 = (double (*)(double)) log2
set $log10 = (double (*)(double)) log10
set $pow = (double (*)(double, double)) pow
set $sin = (double (*)(double)) sin
set $cos = (double (*)(double)) cos
set $tan = (double (*)(double)) tan
set $asin = (double (*)(double)) asin
set $acos = (double (*)(double)) acos
set $atan = (double (*)(double)) atan
set $atan2 = (double (*)(double, double)) atan
set $sinh = (double (*)(double)) sinh
set $cosh = (double (*)(double)) cosh
set $tanh = (double (*)(double)) tanh
set $asinh = (double (*)(double)) asinh
set $acosh = (double (*)(double)) acosh
set $atanh = (double (*)(double)) atanh
```

まとめ

GDB の便利な使い方を紹介しました。

引用文献

① gdb hacks - gdbcalc スクリプト
ほげめも 深追いと佳境の日々
http://www.keshi.org/blog/2006/03/gdb_hacks_gdbcalc.html

参考文献

- GDB：The GNU Project Debugger
  http://sources.redhat.com/gdb/
  http://sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html

—— Hiro Yoshioka

HACK #8 Intel アーキテクチャの基本

CPU アーキテクチャの基本をおさらいします。

デバッグに必要な基礎知識として、CPU アーキテクチャについて簡単に触れます。

バイトオーダ

インテル系の CPU の場合、ビットやバイトの順番は下記のようになります。

エンディアンというのは多バイトのデータをどのようにメモリに配置するかの方法を指します。
例えば、0x12345678 というデータを**図 2-2** のように下位データを下位メモリから配置する方法をリトルエンディアンと呼び Intel アーキテクチャが採用している方法です。逆に上位データを下位メモリから配置する方法をビッグエンディアンと呼び、SPARC や MIPS アーキテクチャが採用しています。

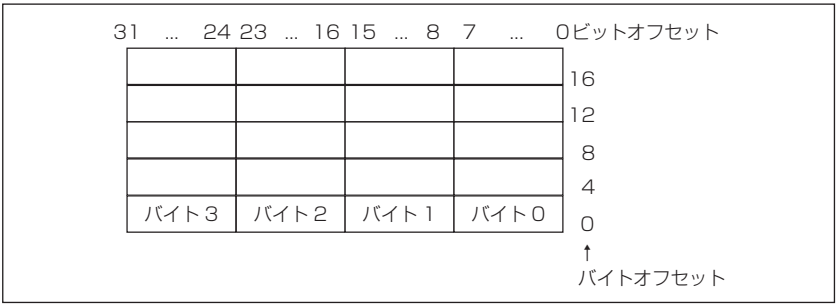

図 2-1 インテル系 CPU の構造

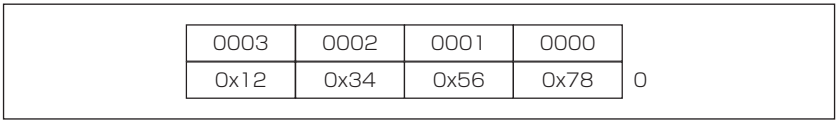

図 2-2　リトルエンディアンの例

32 ビット環境におけるレジスタ

汎用レジスタには**図 2-3** で示す通り 8 種類、EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP があり、論理演算、算術演算、アドレス計算、メモリポインタなどで利用されます。

ESP レジスタはスタックポインタを保持するために利用されます。

命令によっては特定のレジスタを利用します。例えば、文字列命令は ECX、ESI、EDI レジスタをオペランドとして利用します。汎用レジスタの主な用途については**表 2-2** を参照してください。

表 2-2　主なレジスタの用途

レジスタ	用途
EAX	オペランドの演算、結果
EBX	DS セグメントのデータへのポインタ
ECX	文字列やループのカウンタ
EDX	I/O ポインタ
ESI	DS レジスタによって示されるセグメントにあるデータへのポインタ。あるいは、文字列操作のコピー元（source）
EDI	ES レジスタによって示されるセグメントにあるデータへのポインタ。あるいは、文字列操作の行き先（destination）
ESP	スタックポインタ（SS セグメント）
EBP	スタック上のデータへのポインタ（SS セグメント）

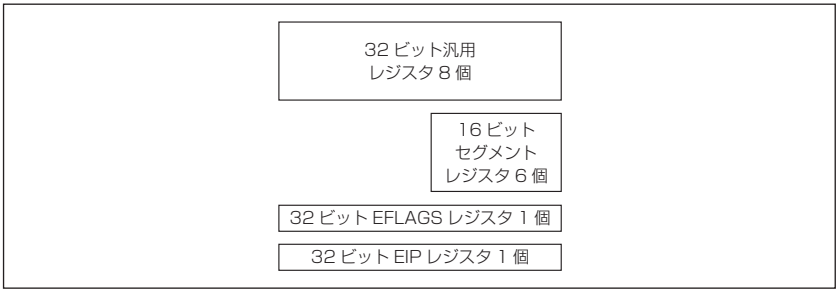

図 2-3　32 ビット環境における基本プログラム実行レジスタ

表 2-3 主なセグメントレジスタの用途

レジスタ	用途
CS	コードセグメント
DS	データセグメント
SS	スタックセグメント
ES	データセグメント
FS	データセグメント
GS	データセグメント

プログラムコードはコードセグメントに置き、データはデータセグメントに置きます。またプログラムのスタックはスタックセグメントに置きます。

ただ汎用レジスタの用途に関しては、上記のような用途に固定されないで汎用に利用される場合もあり目安程度と言えます。

EFLAGS レジスタは、ステータスフラグ、コントロールフラグ、システムフラグなどから成ります。

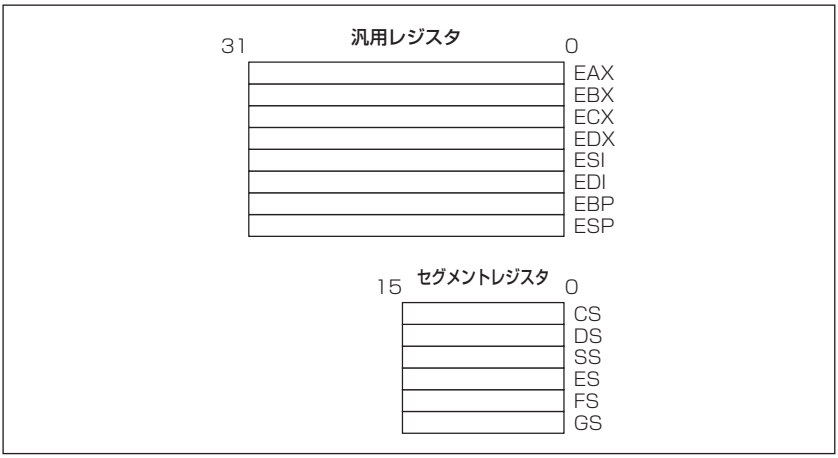

図 2-4 汎用システム

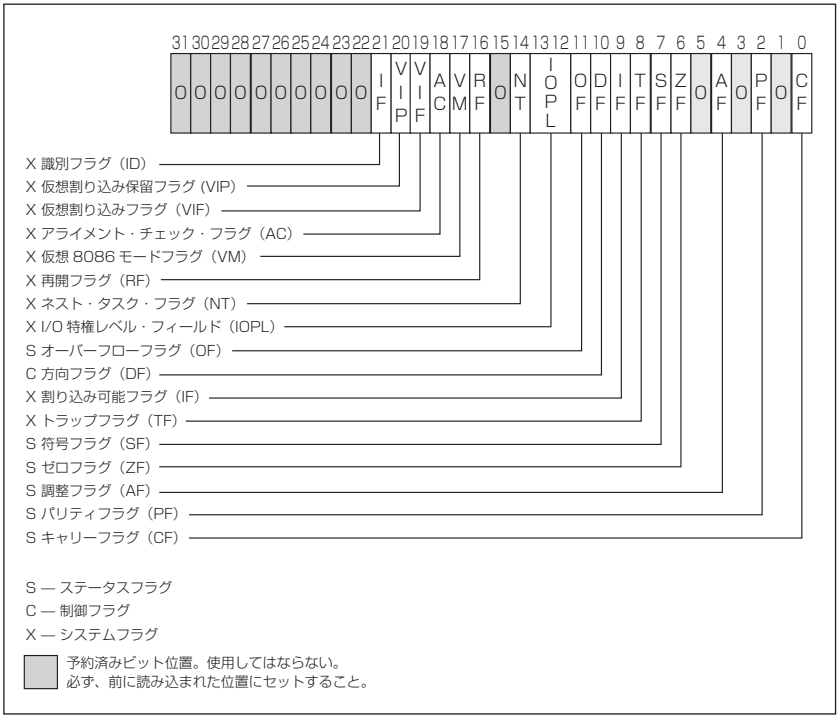

図 2-5　EFLAGS レジスタ

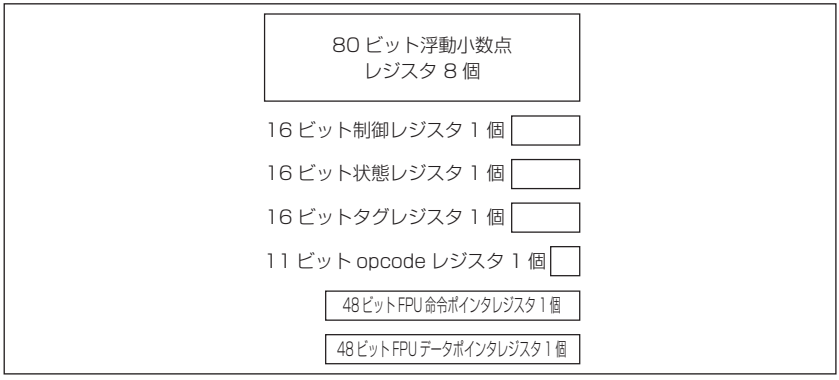

図 2-6　FPU レジスタ

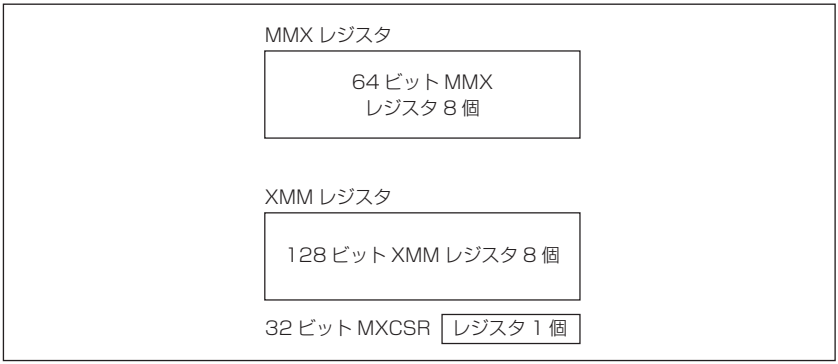

図 2-7 MMX レジスタと XMM レジスタ

EIP（Instruction Pointer）レジスタは、32 ビットの命令ポインタです。

その他のレジスタとして、コントロールレジスタ（CR0 から CR4）、GDTR、IDTR、TR、LDTR、デバッグレジスタ（DR0/DR1/DR2/DR3/DR6/DR7）、メモリタイプレンジレジスタ MTRR、マシン依存レジスタ MSR、マシンチェックレジスタ、パフォーマンスモニタリングカウンタなどがあります。

64 ビット環境におけるレジスタ

アドレス空間 2^{64} バイトまでサポートします。CPUID 命令によって、実行しているプロセッサのサポートしている物理アドレスを確認することができます。

64 ビットモード汎用レジスタは、32 ビットオペランドの場合、EAX/EBX/ECX/EDX/EDI/ESI/EBP/ESP/R8D ～ R15D が利用できます。64 ビットオペランドの場合、RAX/RBX/RCX/RDX/RDI/RSI/RBP/RSP/R8 ～ R15 が利用できます。R8D ～ R15D/R8 ～ R15 が 8 個の新しい汎用レジスタです。RIP レジスタは、64 ビットの命令ポインタ

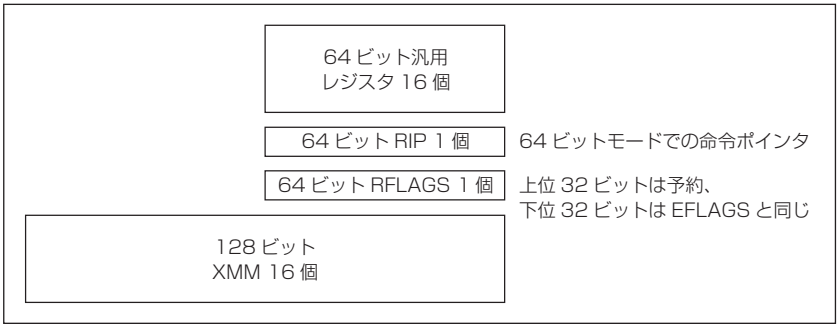

図 2-8 64 ビット環境のレジスタ

です。

スタックポインタは 64 ビットに拡張でき、コントロールレジスタは 64 ビットに拡張できます。新規に CR8 が追加されました。そしてデバッグレジスタは 64 ビットに拡張でき、GDTR、IDTR は 10 バイト拡張、LDTR、TR は 64 ビットに拡張できます。

アドレス

CPUがメモリバスでアドレス指定するメモリは、物理アドレスと呼ばれています。8ビットのバイト列として構成されます。32 ビットモードでは、最大 64GB（2^{36}）です。64 ビットモードの最大物理アドレスは、現在の Intel の実装では 2^{40} バイト、AMD の実装では 2^{48} となっています。

フラットモデル（**図 2-9** 参照）では、メモリはリニアアドレス空間という単一のフラットな連続したアドレス空間のように見えます。Linux はこのメモリモデルを採用しています。

セグメント化メモリモデルは、メモリは、セグメントと呼ばれる独立したアドレス空間のグループのように見えます。セグメント内のアドレスを指定するには、セグメントセレクタとオフセットで構成する論理アドレスによって行います。セグメントセレクタでアクセスの対象となるセグメントを識別し、オフセットで、そのセグメントのアドレス空間にあるメモリを識別します。32 ビットモードでは、最大 16383 個のセグメントを指定できます。各セグメントのサイズは最大 2^{32} バイトまでです。

64 ビットモードではフラットモデルが採用されていて、64 ビットのリニアアドレスを利用できます。セグメント化メモリモデルは利用できません。

データ型

基本データ型としてバイト（8 ビット）、ワード（16 ビット）、ダブルワード（32 ビット）、クワッドワード（64 ビット）、ダブルクワッドワード（128 ビット）があります。

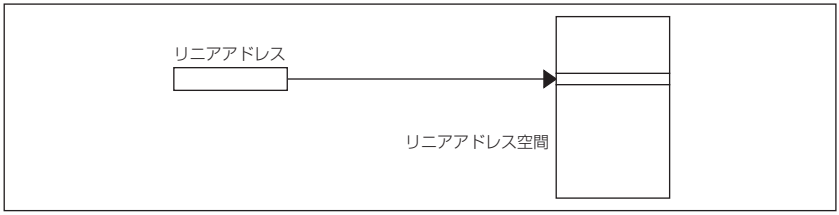

図 2-9　フラットモデル

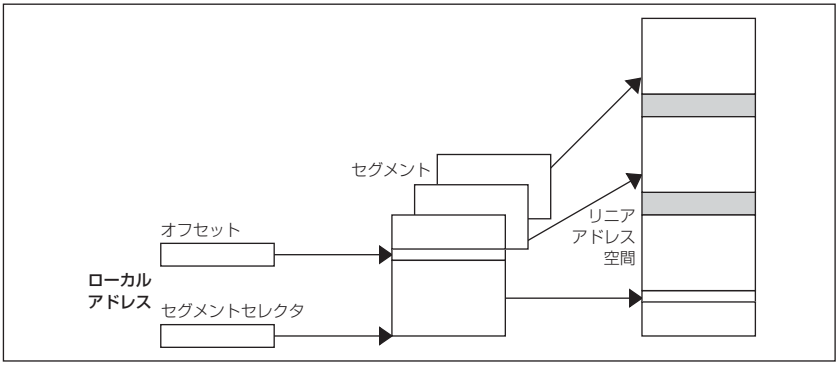

図 2-10　セグメント化モデル

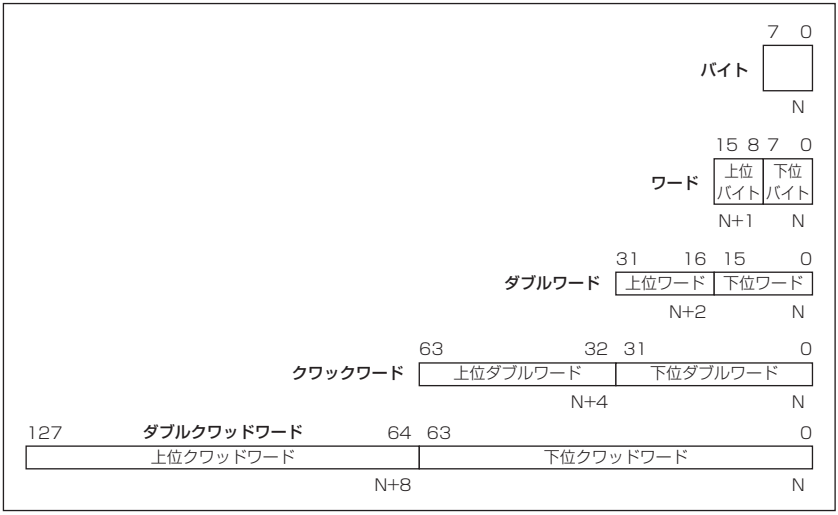

図 2-11　基本データ型

整数データ型

2 種類の整数（符号なし整数と符号付き整数）をサポートしています。符号なし整数では 0 ～正の最大数の範囲で、選択したオペランドサイズ（バイト、ワード、ダブルワード、クワッドワード）でエンコードできます。符号付き整数は、正と負の両方の整数値を表現できます。2 の補数の 2 進数です。

浮動小数点データ型

単精度浮動小数点、倍精度浮動小数点、拡張倍精度浮動小数点の 3 つをサポート

しています。これらのデータ型のデータフォーマットはIEEE754で定義されているものに対応しています。
単精度浮動小数点（32ビット）の精度は24ビット、倍精度浮動小数点（64ビット）の精度は53ビット、拡張倍精度浮動小数点（80ビット）の精度は64ビットです。

その他、ポインタデータ型、ビットフィールドデータ型、ストリングデータ型、パックドSIMDデータ型、BCDおよびパックドBCD整数データ型があります。それぞれの詳細については、Intelのマニュアルを参照してください。

スタック

スタックに関しては **[HACK #9]** を参照してください。

まとめ

Intelアーキテクチャの基本について記しました。

参考文献

- Intel 64 and IA-32 Architectures Software Developer's Manual (in five volumes)
  http://developer.intel.com/products/processor/manuals/index.htm

—— Hiro Yoshioka

HACK #9 デバッグに必要なスタックの基礎知識

デバッグには欠かせないスタックの基本について説明します。

スタックとはプログラムがデータを格納するためのメモリ領域のひとつであり、後から入れたデータを先に取り出すというLIFO（Last In First Out）のデータ構造であるのが特徴です。スタックにデータを積むことをPUSH、スタックからデータを取り出すことをPOPと言います。動的に確保される自動変数を格納する際にはスタックが使用されます。また関数コールをする際には、呼び出す関数への引数渡しに使用されたり、戻り番地や戻り値を格納する用途にも使用されます。

本Hackでは、以下のサンプルプログラムを使います。これはコマンド引数で渡された数字を最終値として、0から最終値までの正数の総和を求めるプログラムです。

```
$ cat sum.c
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
```

```
#define MAX             (1UL << 20)

typedef unsigned long long u64;
typedef unsigned int u32;

u32 max_addend = MAX;

u64 sum_till_MAX(u32 n)
{
        u64 sum;
        n++;
        sum = n;

        if (n < max_addend)
                sum += sum_till_MAX(n);
        return sum;
}

int main(int argc, char** argv)
{
        u64 sum = 0;

        if ((argc == 2) && isdigit(*(argv[1])))
                max_addend = strtoul(argv[1], NULL, 0);
        if (max_addend > MAX || max_addend == 0) {
                fprintf(stderr, "Invalid number is specified\n");
                return 1;
        }

        sum = sum_till_MAX(0);
        printf("sum(0..%lu) = %llu\n", max_addend, sum);
        return 0;
}
```

0 から 10 までの総和を求める場合、次のように実行します。

```
$ gcc -o sum -g sum.c
$ ./sum 10
sum(0..10) = 55
```

関数コールとスタックの関係

関数コール前後でスタックがどのように変化するのかを説明します。図 2-12 の（a）が関数コール前、(b) が sum_till_MAX() 関数コール後、(c) はさらに sum_till_MAX() 関数コール後のスタックの状態を表しています。

関数に渡す引数、コール元への戻り番地、上位フレームのフレームポインタ、そして関数内で使用する自動変数という順にスタックに積まれます。さらに関数の処理によっては、レジスタ値の一時保存領域としてスタックが消費されます。これらの情報は関数ごとに独立して作成され、スタックフレームと呼びます。この時、スタックフレームのベースを表すフレームポインタ（FP）に適切な値が設定されます。またスタックポインタ（SP）は常にスタックの先頭を指しています。

x86_64 では自動変数や作業領域がスタックポインタを超える場合があります。スタックポインタが指すアドレスから、さらに 128 バイト先までの領域をレッドゾーンと言い、自動変数や作業領域として使ってよいことになっています。これは AMD64 の ABI 仕様で定義されています。

では対応するアセンブラコードと合わせて動作を確認しましょう。

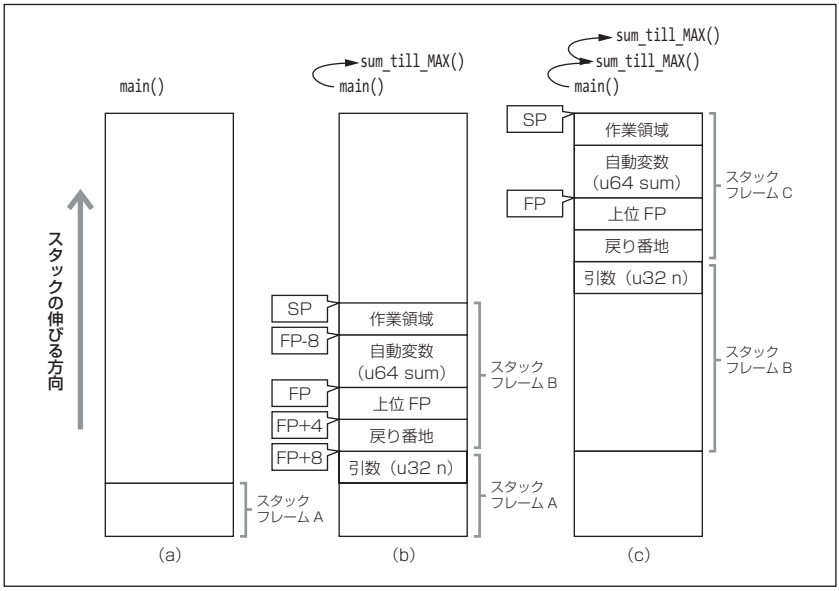

図 2-12　関数コール前後のスタックの状態

```
(gdb) disas main
…
0x8048544 <main+175>:   push   $0x0 ──────────────────────①
0x8048546 <main+177>:   call   0x8048458 <sum_till_MAX> ──②
0x0804854b <main+182>:  add    $0x8,%esp
…
```

関数コールする際にはまず、関数に渡す引数をスタックに積みます①。次の sum_till_MAX() の call 命令によって、自動的に戻り番地（0x0804854b）がスタックに積まれます②。

ではコールされる関数 sum_till_MAX() を見ていきます。

```
(gdb) disas sum_till_MAX
Dump of assembler code for function sum_till_MAX:
0x08048458 <sum_till_MAX+0>:    push   %ebp ──────────────③
0x08048459 <sum_till_MAX+1>:    mov    %esp,%ebp ─────────④
0x0804845b <sum_till_MAX+3>:    sub    $0x10,%esp ────────⑤
0x0804845e <sum_till_MAX+6>:    incl   0x8(%ebp) ─────────⑥
0x08048461 <sum_till_MAX+9>:    mov    0x8(%ebp),%eax
0x08048464 <sum_till_MAX+12>:   mov    $0x0,%edx
0x08048469 <sum_till_MAX+17>:   mov    %eax,-0x8(%ebp) ───⑦
0x0804846c <sum_till_MAX+20>:   mov    %edx,-0x4(%ebp)
0x0804846f <sum_till_MAX+23>:   mov    0x804978c,%eax
0x08048474 <sum_till_MAX+28>:   cmp    %eax,0x8(%ebp)
0x08048477 <sum_till_MAX+31>:   jae    0x804848d <sum_till_MAX+53>
0x08048479 <sum_till_MAX+33>:   sub    $0xc,%esp
0x0804847c <sum_till_MAX+36>:   pushl  0x8(%ebp)
0x0804847f <sum_till_MAX+39>:   call   0x8048458 <sum_till_MAX>
0x08048484 <sum_till_MAX+44>:   add    $0x10,%esp
0x08048487 <sum_till_MAX+47>:   add    %eax,-0x8(%ebp)
0x0804848a <sum_till_MAX+50>:   adc    %edx,-0x4(%ebp)
0x0804848d <sum_till_MAX+53>:   mov    -0x8(%ebp),%eax
0x08048490 <sum_till_MAX+56>:   mov    -0x4(%ebp),%edx
0x08048493 <sum_till_MAX+59>:   leave ──────────────────⑧
0x08048494 <sum_till_MAX+60>:   ret ────────────────────⑨
End of assembler dump.
```

上位フレームのフレームポインタをスタック上に退避し③、新しいスタックフレームのフレームポインタを設定します④。そして自動変数格納用の領域をスタック上に確保します⑤。ここまでで図 2-12 に示したスタックフレームの作成が完了です。

⑥からは sum_till_MAX() 関数の処理になっています。0x8(%ebp) はフレームポインタから +8

バイトの場所を指しているので、図 2-12 を見ると関数に渡された引数（u32 n）を参照しています。つまり、n++; の箇所はアセンブラではこのようにエンコードされるわけです。⑦ではフレームポインタから -8 バイトの場所ですので、自動変数（u64 sum）を表しています。ただし、変数 sum は 64 ビット長で宣言されているため、⑦では sum の下位 32 ビットだけを扱っています。%eax には引数 n の値が格納されているので、⑦は sum = n; に該当していることがわかります。⑧の leave 命令は、スタックフレームを削除する命令です。③と④の全く逆の処理を実行し、現在のフレームを破棄します。⑨はサブルーチン（関数）からのリターンです。スタックに保存された戻り番地をプログラムカウンタレジスタに POP し、処理を呼び出し元へ戻します。

デバッガのバックトレース

GDB などのデバッガのバックトレース機能は、スタックに保存された情報を検索することで実現されています。

以下では、2 度目の sum_till_MAX() の中で中断させています。ちょうど図 2-12 の (c) と同じ状況です。

```
(gdb) bt
#0  sum_till_MAX (n=2) at sum.c:18
#1  0x08048484 in sum_till_MAX (n=1) at sum.c:19
#2  0x0804854b in main (argc=1, argv=0xbfd89b34) at sum.c:34
```

では自分の手で GDB のバックトレースと同じことを行ってみます。スタックに保存された情報と、図 2-12 のスタックイメージを照らし合わせながら見ていきます。まずは現在の実行位置と現在の FP の値を取得します。現在の実行位置はプログラムカウンタ（PC）で取得でき、x86 プロセッサであれば PC は eip レジスタです。FP は ebp レジスタです。

```
(gdb) i r eip ebp
eip            0x804846f          0x804846f <sum_till_MAX+23>
ebp            0xbfd89a28         0xbfd89a28
```

次にスタックをダンプします。具体的には次のように、スタックの先頭を表す SP から適当なサイズだけダンプさせます。

```
(gdb) x/40w $sp
```

実際にスタックをダンプさせた結果を図 2-13 に示します。図中の説明に対応する部分にコメントと印を入れてあるので、対応を確認してみてください。

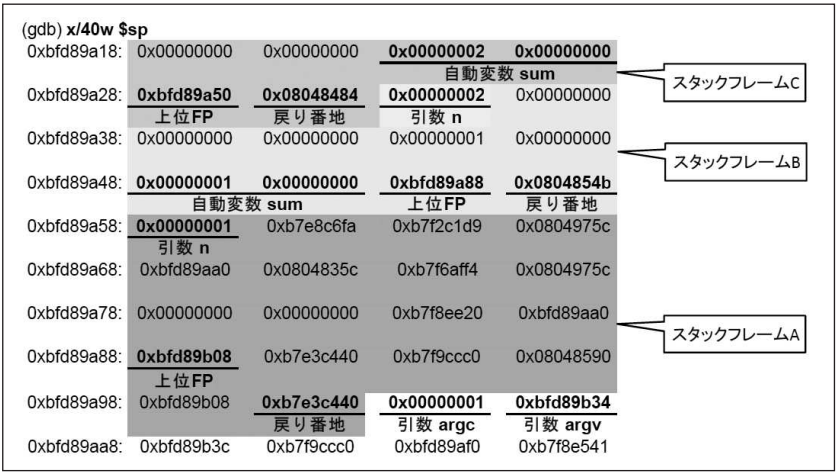

図 2-13 スタックのダンプ結果

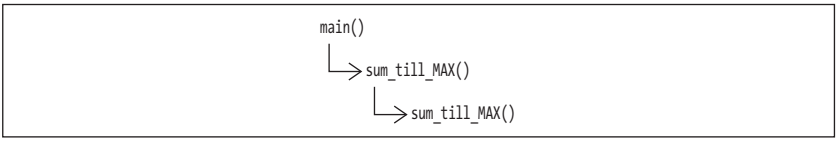

図 2-14 スタック情報から得られたコールトレース

このようにスタックに残された戻り番地の情報から、GDB のバックトレース結果と同じコールトレースが得られました（図 2-14）。

スタック上のデータがデバッガにとって非常に重要な情報だということがわかったと思います。万が一、スタック上のデータが壊れてしまっている場合は、デバッガを使ったコールトレースを行うことはできません。スタック破壊については**「バックトレースが正しく表示されない」[HACK #27]**、**「配列の不正アクセスによるメモリ内容の破壊」[HACK #28]** を参照してください。

コンパイル時、gcc に -fomit-frame-pointer オプションを指定するとフレームポインタを使用しないバイナリが生成されます。この場合、**図 2-12** のスタックイメージにある FP や上位 FP の情報はスタックに記録されません。しかし、そのような場合でも GDB は正しくフレームを理解することができます。これは GDB が、デバッグ情報に記録されたスタック使用量を基にフレーム位置を割り出すからです。

GDB でスタックフレームを操る

GDB にはスタックフレームを操作するコマンドがあります。ここではコマンドの利用例を紹介します。GDB でプロセスを以下の状態で停止させています。

```
(gdb) bt
#0  sum_till_MAX (n=4) at sum.c:18
#1  0x08048484 in sum_till_MAX (n=3) at sum.c:19
#2  0x08048484 in sum_till_MAX (n=2) at sum.c:19
#3  0x08048484 in sum_till_MAX (n=1) at sum.c:19
#4  0x0804854b in main (argc=1, argv=0xbfb92454) at sum.c:34
```

frame コマンドで現在選択しているフレームを確認できます。

```
(gdb) frame
#0  sum_till_MAX (n=4) at sum.c:18
18                  if (n < max_addend)
```

現在選択されているのはフレーム #0 です。このフレーム内で自動変数 sum を確認すると値に 4 が入っています。

```
(gdb) p sum
$1 = 4
```

次に 1 つ上のフレームであるフレーム #1 を選択し、同様に自動変数 sum を確認します。

```
(gdb) frame 1
#1  0x08048484 in sum_till_MAX (n=3) at sum.c:19
19                              sum += sum_till_MAX(n);
(gdb) p sum
$1 = 3
```

フレーム #1 では自動変数 sum の値は 3 です。フレーム #0 とフレーム #1 において、同じ変数名 sum でアクセスしていますが、GDB は選択したフレーム上での値を返してくれるわけです。またこの他にも、フレームの選択には up コマンドと down コマンドが使えます。up コマンドは 1 つ上のフレーム、down コマンドは 1 つ下のフレームを選択することができます。

```
(gdb) up
#2  0x08048484 in sum_till_MAX (n=2) at sum.c:19
```

```
19                              sum += sum_till_MAX(n);
(gdb) p sum
$2 = 2
```

infoコマンドのオプションにframeを指定すると、より詳細なスタックフレーム情報が得られます。このコマンドでは、引数にフレーム番号を指定できます。

```
(gdb) i frame 1
Stack frame at 0xbfd8e218:
 eip = 0x8048484 in sum_till_MAX (sum.c:19); saved eip 0x8048484
 called by frame at 0xbfd8e240, caller of frame at 0xbfd8e1f0
 source language c.
 Arglist at 0xbfd8e210, args: n=3
 Locals at 0xbfd8e210, Previous frame's sp is 0xbfd8e218
 Saved registers:
  ebp at 0xbfd8e210, eip at 0xbfd8e214
```

スタックサイズの制限

実は本Hackで使用しているサンプルプログラムは、引数なしで実行するとセグメンテーションフォルトを起こすようになっています。実行してみましょう。

```
$ ./sum
Segmentation fault
```

スタックオーバーフローが発生しています。GDBからサンプルプログラムを実行し、何を実行した時にセグメンテーションフォルトが発生したのかを確認してみましょう。プログラムの実行位置を調べるにはプログラムカウンタ（PC）の値を確認します。

```
$ gdb ./sum
...
(gdb) r
Starting program: /home/toyo/work/test/sum

Program received signal SIGSEGV, Segmentation fault.
0x0804847c in sum_till_MAX (n=209442) at sum.c:19
19                              sum += sum_till_MAX(n);
(gdb) x/i $pc
0x804847c <sum_till_MAX+36>:    pushl  0x8(%ebp)
```

sum_till_MAX() の引数 n をスタックの先頭に PUSH する命令です。では現在のスタックポインタ（SP）の位置を確認します。

```
(gdb) p $sp
$1 = (void *) 0xbf06dffc
```

このプロセスのメモリマップを調べます。GDB がアタッチしているプロセスのメモリマップを調べるには、次のコマンドを実行します。このコマンドを実行すると、GDB はデバッグ対象プロセスに対応する /proc/<PID>/maps の情報を表示します。

```
(gdb) i proc mapping
process 11545
cmdline = '/home/toyo/work/test/sum'
cwd = '/home/toyo/work/test'
exe = '/home/toyo/work/test/sum'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000          0        /home/toyo/work/test/sum
         0x8049000  0x804a000     0x1000          0        /home/toyo/work/test/sum
        0xb7e56000 0xb7e57000     0x1000 0xb7e56000
        0xb7e57000 0xb7f9a000   0x143000          0        /lib/libc-2.7.so
        0xb7f9a000 0xb7f9b000     0x1000   0x143000        /lib/libc-2.7.so
        0xb7f9b000 0xb7f9d000     0x2000   0x144000        /lib/libc-2.7.so
        0xb7f9d000 0xb7fa0000     0x3000 0xb7f9d000
        0xb7fae000 0xb7fb0000     0x2000 0xb7fae000
        0xb7fb0000 0xb7fb1000     0x1000 0xb7fb0000           [vdso]
        0xb7fb1000 0xb7fcd000    0x1c000          0        /lib/ld-2.7.so
        0xb7fcd000 0xb7fcf000     0x2000    0x1b000        /lib/ld-2.7.so
        0xbf06e000 0xbf86e000   0x800000 0xbf800000           [stack]
```

最終行の [stack] に注目してください。これはスタック領域を示していますが、スタック領域のトップが 0xbf06e000 となっています。先ほど調べたスタックポインタの値は 0xbf06dffc でしたので、このスタック領域の範囲を超えています。スタックとして使用可能な範囲を超えてアクセスしようとした、つまりスタックオーバーフローです。

このコマンドは GDB から /proc/<PID>/maps を開いています。つまりコアダンプ解析の場合は使用できません。コアダンプ解析の場合には、次のコマンドによって同様の情報が得られます。

```
(gdb) info files
あるいは
(gdb) info target
```

このサンプルプログラムはデフォルトでは100万回以上も sum_till_MAX() が再帰的にコールされるようになっています。これまで説明してきたように、関数コールの度にスタックフレームが生成され、それに伴いスタックが消費されていきます。これがプロセスに許可されるスタック量をオーバーしてしまうため、スタックオーバーフローが発生するという仕組みです。

筆者の環境ではプロセスに許可されるスタック量は 8MB でした。

```
$ ulimit -s
8192
```

これを 10 倍に増やし、再度サンプルプログラムを実行してみます。するとセグメンテーションフォルトを起こさず、正常終了するようになりました。

```
$ ulimit -Ss 81920
$ ./sum
sum(0..1048576) = 549756338176
```

0 から 1048576 までの正数の和ですから、次の計算と等価です。

```
(1 + 1048576) * (1048576 / 2)
= 1048577 * 524288
= 549756338176
```

これで正しい結果が得られたことが証明できました。

まとめ

基本的なスタックの仕組みと、デバッガのバックトレースがどのようにスタック情報を利用しているのかを説明しました。また、GDB で利用できるスタックフレームを操る便利なコマンドを紹介しました。スタックオーバーフローについても例を挙げて説明しました。本 Hack ではプロセスに許可されるスタック量を説明しましたが、スレッドごとに設

定されるスタック量制限もあります。マルチスレッドプログラムの場合、各スレッドが使用するスタックの総和が、プロセスに許可されるスタック量を超えないようにしなければなりません。同時に、スレッドごとのスタック量制限にも注意する必要があります。アプリケーションを設計する際には、スタックの使用量にも注意をはらいましょう。

参考文献

- AMD64 Application Binary Interface
  http://www.x86-64.org/documentation/abi.pdf

—— Toyo Abe

HACK #10 関数コール時の引数の渡され方（x86_64 編）

x86_64 アーキテクチャにおいて、どのように引数が呼び出し先の関数へ渡されるかを説明します。

関数の引数とデバッグ

プログラムが異常終了する等、期待と異なる動作をすることは、よくある不具合のひとつです。エラーメッセージが表示されていれば、その表示を行うソースコードの位置は、文字列検索を行うことで、比較的容易に特定されます。しかし、不具合の真の原因は、エラーメッセージを表示する箇所より、ずっと前に発生していることもあります。例えば、ある関数で誤った値を算出し、それを引数として、別の関数を呼び出す場合等です。このような場合、プログラムのどの箇所が不正であるかを突き止めることが、不具合解決への糸口となります。しかしながら、実際には、そのような箇所の見当がつかないことも多いでしょう。そのような時には、不具合に関連しそうな関数の引数を検査することで、どの関数に問題があるのかを絞り込むことができます。本 Hack では、以下のプログラムを例に、GDB を使って関数の引数を調べる方法を説明します。

引数の呼び出し先関数への渡され方はアーキテクチャや言語、コンパイラによって異なります。本 Hack では、x86_64 アーキテクチャ上で C 言語を用いる場合について解説します。続く **[HACK #11]** では、i386 アーキテクチャで C 言語を用いた場合、**[HACK #12]** では、C++ 言語を用いた場合の引数の渡され方を説明します。いずれの Hack でもコンパイラには GCC（G++）を使用します。

```
#include <stdio.h>
#include <stdlib.h>

int v1 = 1;
float v2 = 0.01;
```

```
void func(int a, long b, short c, char d, long long e, float f, double g, int *h, float *i, char*j)
{
    printf("a: %d, b: %ld, c: %d, d: %c, e: %lld\n"
           "f: %.3e, g: %.3e\nh: %p, i: %p, j: %p\n", a, b, c, d, e, f, g, h, i, j);
}

int main(void)
{
   func(100, 35000L, 5, 'A', 123456789LL, 3.14, 2.99792458e8, &v1, &v2, "string");

   return EXIT_SUCCESS;
}
```

このプログラムの実行結果は、筆者の実行環境では以下のようになりました。

```
a: 100, b: 35000, c: 5, d: A, e: 123456789
f: 3.140e+00, g: 2.998e+08
h: 0x600990, i: 0x600994, j: 0x4006a3
```

GDB で調べる

もっとも簡単に調べる方法は、GDB を使うことです。先ほどのサンプルプログラムを -g オプションを付けてビルドした後、GDB で呼び出し先の関数 func() の先頭でブレークさせると、以下のように引数が表示されます。

```
(gdb) b func
Breakpoint 1 at 0x4004a0: file func_call.c, line 10.
(gdb) run
...
Breakpoint 1, func (a=100, b=35000, c=5, d=65 'A', e=123456789, f=3.1400001, g=299792458, h=0x600990,
i=0x600994, j=0x4006a3 "string") at func_call.c:10
10          printf("a: %d, b: %ld, c: %d, d: %c, e: %lld\n"
```

一方、-g オプションがビルド時に指定されていない場合、すなわち、デバッグ情報が利用できない場合、下記のように停止アドレスが表示されるだけで、引数の値は表示されません。このような場合に、引数を取得する方法を以下に説明します。

```
(gdb) b func
Breakpoint 1 at 0x40047c
(gdb) run
```

```
...
Breakpoint 1, 0x0000000000040047c in func ()
```

x86_64 の呼び出し

x86_64 では、整数型やポインタ型の引数は、左から rdi、rsi、rdx、rcx、r8、r9 に、浮動小数点型の引数は、xmm0、xmm1、... に格納されます。引数の数がこれらのレジスタより多い場合、残りの引数はスタックに格納されます。したがって、GDB で調査したい関数の先頭でブレークさせた直後に、レジスタやスタックを調べると、引数の値を取得できます。

なお、先ほどの例では、ブレークポイントを関数名 func で指定しましたが、これより後では、関数名の前に *（アスタリスク）を付けます。というのは、* を付けない場合、ブレークポイントは、アセンブリ言語レベルでの関数の先頭ではなく、ソースコードに対応する少し後のアドレスになります。多くの場合、関数の先頭では、以下のようにスタックの操作が行われます。引数は、スタックにも格納されることがあるため、* なしで関数名を指定した break コマンドを、引数を調査するために使用することはできません。

```
(gdb) disas func
Dump of assembler code for function func:
0x0000000000400478 <func+0>:    push   %rbp
0x0000000000400479 <func+1>:    mov    %rsp,%rbp
0x000000000040047c <func+4>:    sub    $0x50,%rsp
...
```

では、さっそく引数を調べていきましょう。

```
(gdb) b *func
Breakpoint 1 at 0x400478
(gdb) run
...
Breakpoint 2, 0x0000000000400478 in func ()
```

この状態で、レジスタを確認します。最初の 5 つの引数、a、b、c、d、e は、それぞれ rdi、rsi、rdx、rcx、r8 にそれぞれ格納されていることがわかります。

```
(gdb) i r
rax            0x7fff93d32328   140735673475880
rbx            0x37d8019bc0     239847185344
rcx            0x41     65      ———————————— 引数 d
```

```
rdx            0x5      5                    ―――――――――――― 引数c
rsi            0x88b8   35000 ―――――――――――――――――――― 引数b
rdi            0x64     100  ―――――――――――――――――――― 引数a
rbp            0x7fff93d32330   0x7fff93d32330
rsp            0x7fff93d322f8   0x7fff93d322f8
r8             0x75bcd15        123456789    ―――――――― 引数e
r9             0x7fff93d3232c   140735673475884 ――――― 引数h
r10            0x0      0
r11            0x37d821d7b0     239849297840
r12            0x0      0
r13            0x7fff93d32410   140735673476112
r14            0x0      0
r15            0x0      0
rip            0x400478 0x400478 <func>
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

また、浮動小数点型の第6、第7引数f、gは、それぞれxmm0、xmm1に格納されます。これらの値は、次のように取得できます。

```
(gdb) p $xmm0.v4_float[0]
$4 = 3.1400001
(gdb) p $xmm1.v2_double[0]
$5 = 299792458
```

xmm0/xmm1のサフィックス（v4_float、v2_double）は、GDBがこれらのレジスタを、次の共用体のように扱うために、付加されています。

```
union {
  float    v4_float[4];
  double   v2_double[2];
  int8_t   v16_int8[16];
  int16_t  v8_int16[8];
  int32_t  v4_int32[8];
  int64_t  v2_int64[8];
```

```
    int128_t unit128;
  } xmm0;
```

これは、xmm0 や xmm1 は、実際には 128 ビットの長さを持ち、それよりも小さいサイズの変数を同時に複数保持できる構造になっているためです。

ポインタ型である第 8 引数の h は、整数型と同様に扱われ、r9 に格納されています。第 9、第 10 引数もポインタ型であるため、引数の数が少ないときには、レジスタに格納されるのですが、本 Hack のサンプルの場合、もう格納すべきレジスタがありません。そのため、これらはスタックに格納されて渡されます。

以下のようにスタックを調べます。いま注目している引数は 2 つであるのに、3 つ表示させた理由は、スタックの先頭には関数のリターンアドレスが積まれているためです。また、g（giant word）で表示させるのは、x86_64 アーキテクチャでは、整数やポインタは、このサイズで扱われるためです。

```
(gdb) x/3g $rsp
0x7fff4c79fd78: 0x0000000000400558      0x0000000000600994
0x7fff4c79fd88: 0x00000000004006a3
```

これらを図 2-15 に示します。

スタックの先頭（アドレス 0x7fff4c78fd78）の値は、リターンアドレスなので無視すると、その次の 2 つの値が、それぞれ第 9、第 10 引数の i、j に対応していることがわかります。また、i や j は、ポインタなので、そのポインタが指す値や文字列を確認した場合もあるでしょう。その場合は、下記のようにします。

```
(gdb) printf "%.2f\n", *(float*)0x0000000000600994
0.01
```

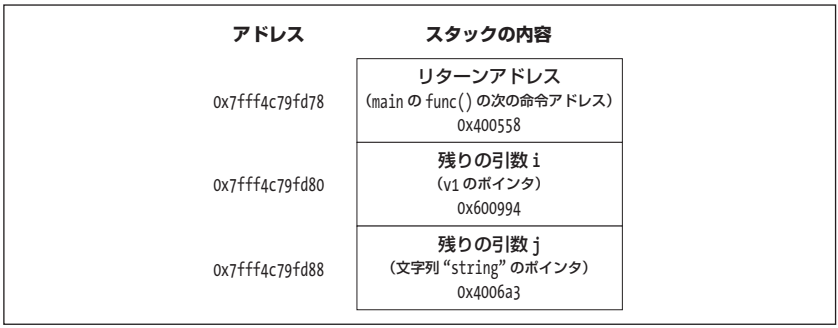

図 2-15　スタックの内容

```
(gdb) p (char*)0x0000000004006a3
$28 = 0x4006a3 "string"
```

ここで、表示アドレスを16進数で入力するのが、スマートでないと思う方もいるでしょう(もちろん、マウスでコピーするだけなので簡単と思う人もいるでしょうが)。入力する文字数はあまり変わりませんが、以下のようにしても同じ結果を得られます。

```
(gdb) printf "%.2f\n", *(float*)(*(unsigned long*)($rsp+0x8))
0.01
(gdb) p (char*)(*(unsigned long*)($rsp+0x10))
$29 = 0x4006a3 "string"
```

参考文献

- AMD64 Application Binary Interface
  http://www.x86-64.org/documentation/abi.pdf

まとめ

関数に渡される引数の調べ方を説明しました。x86_64では基本的にレジスタを使用し、レジスタの数が足りない場合、引数をスタックに格納します。

—— Kazuhiro Yamato

HACK #11 関数コール時の引数の渡され方(i386編)

i386アーキテクチャにおいて、どのように引数が呼び出し先の関数へ渡されるかを説明します。

i386の呼び出し

ここでは、**[HACK #10]**で説明したサンプルプログラムを再び使用します。筆者の環境でビルドして実行させた結果は、以下のようになっています。当然ですが、ポインタの値h、i、jだけが、**[HACK #10]**での結果と異なっています。

```
a: 100, b: 35000, c: 5, d: A, e: 123456789
f: 3.140e+00, g: 2.998e+08
h: 0x80496d0, i: 0x80496d4, j: 0x80485bb
```

i386では、原則、引数は、すべてスタックに積まれます。そのため、関数の先頭でブレークさせた後、以下のようにスタックの内容を調べることで、引数を取得できます。第1引数の値を取得する際に、espに+4をしているのは、i386アーキテクチャでは、スタックの先頭に、リターンアドレスが積まれており、かつ、整数やポインタのサイズが4バイト

であるためです。

```
(gdb) p *(int*)($esp+4)
$4 = 100
(gdb) p *(long*)($esp+8)
$6 = 35000
(gdb) p *(short*)($esp+12)
$7 = 5
(gdb) p *(char*)($esp+16)
$8 = 65 'A'
(gdb) p *(long long*)($esp+20)
$9 = 123456789
```

次の引数 f についても同様に調べることができますが、その引数の格納アドレスとして、前の引数が格納されていたアドレスよりも 8 バイト大きい値を指定します。なぜなら、i386 アーキテクチャでは、long long 型と double 型は、8 バイト長であるためです。

```
(gdb) printf "%.2e\n", *(float*)($esp+28)
3.14e+00
(gdb) printf "%.3e\n", *(double*)($esp+32)
2.998e+08
```

次の引数を表示させる際、直前の値が double 型なので、スタックを 8 バイト使用していることに注意します。

```
(gdb) p/x *(int*)($esp+40)
$15 = 0x80496d0
(gdb) p/x *(int*)($esp+44)
$16 = 0x80496d4
(gdb) p/x *(int*)($esp+48)
$17 = 0x80485bb
```

i386 でのレジスタ呼び出し

i386 でも、x86_64 のように、一部の引数をレジスタに格納して、呼び出すことができます。一般にこのような呼び出し方をファーストコールと呼びます。具体的にどのレジスタが使用されるかは、処理系に依存します。

GCC では、__attribute__((regparm(3))) を付加して、関数を宣言することで、このような呼び出しを行うことができます。これにより、最初の 3 つの引数が eax、edx、ecx を使って渡されるようになります。

Linux カーネルでは、FASTCALL や asmregparm というマクロが、この機能を利用しています。

以下の説明では、**[HACK #10]** でのプログラムの func() を次のように変更したものを使用します。

```
__attribute__((regparm(3)))
void func(int a, long b, short c, char d, long long e, float f, double g, int *h, float *i, char*j)
{
…
```

では、func() の先頭で、ブレークさせ、レジスタの値を確認します。第 1、2、3 引数は、それぞれ、eax、edx、ecx に格納されていることがわかります。

```
(gdb) b *func
Breakpoint 1 at 0x8048374
(gdb) run
…
(gdb) i r
eax            0x64     100          ———————————— 引数 a
ecx            0x5      5            ———————————— 引数 c
edx            0x88b8   35000        ———————————— 引数 b
ebx            0x8c5ff4 9199604
esp            0xbfdddf2c       0xbfdddf2c
ebp            0xbfdddf58       0xbfdddf58
esi            0xbfdddff4       -1075978252
edi            0xbfdddf80       -1075978368
eip            0x8048374        0x8048374
eflags         0x200286 2097798
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

第 4 引数以降は、通常の場合と同様に、スタックに格納されています。

```
(gdb) p *(char*)($esp+4)
$1 = 65 'A'
(gdb) p *(long long*)($esp+8)
$2 = 123456789
…
```

もし、第 1 引数や第 2 引数に long long 型（64 ビット型）の変数を指定した場合、eax と edx などの組を使って、レジスタで引数が渡されます。ただし、第 2 引数に long long を指定した場合、レジスタ渡しになるのは、第 1 引数のサイズが 32 ビットの場合に限られます。

まとめ

i386 では基本的にすべての引数をスタックに格納します。ただし、GCC の拡張機能である __attribute__((regparm())) を用いれば、一部の引数をレジスタで渡すこともできます。

—— Kazuhiro Yamato

HACK #12 関数コール時の引数の渡され方（C++ 編）

C++ で記載されたプログラムにおいて、どのように引数が呼び出し先の関数へ渡されるかを説明します。

C++ 言語の呼び出し

次の C++ のソースコードの動作を考えてみましょう。

```
#include <cstdio>

class foo {
    int a;
    int b;
public:
    void func(int x, int y);
};

void foo::func(int x, int y)
{
    a = x;
    b = y + 2;
}
```

```
int main(void)
{
    foo f1, f2;
    printf("f1: %p, f2: %p\n", &f1, &f2); ―――――⓪
    f1.func(5, 1); ―――――――――――――――――①
    f2.func(-4, 2); ―――――――――――――――――②

    return 0;
}
```

class foo のメンバ a と b は、foo のインスタンス（この例では、f1、f2）ごとに異なる値を持つため、一般的には、インスタンスの数だけ、それらのための領域が必要になります。一方で、メソッド func の実体はひとつだけあれば十分です。実際、foo::func() は、ビルドされた後の ELF ファイル中では、C 言語で記述された関数と同じように扱われます。しかし、例えば、f1.func(5, 1) が実行された時、func() の中では、明示的に、f1 の a と b に対してアクセスとすることを記述していないにも関わらず、自動的に、f1 の a と b に対して計算結果を代入します。この仕組みは、実際には、func(int x, int y) に、アクセスすべきインスタンスの情報が渡されるために可能になっています。すなわち、func() が下記の C 言語の関数のように振る舞っているとも言えます。そのため、メソッドのコール時には、プロトタイプ宣言の引数 +1 個の引数が渡されます。

```
void func(class foo *this, int x, int y)
{
    this->a = x;
    this->b = y + 2;
}
```

コンパイル後の ELF ファイル中では、C++ の関数も C の関数も、アセンブリ言語で記述された関数でさえも区別はされません。しかし、C++ の関数は、ビルド時にマングルとよばれる関数名の変更処理が行われます。例えば上記ソースの foo::func は、ELF ファイル中では、下記のようなシンボルです。なお、マングルによって、どんなシンボルになるかは処理系により異なります。

```
# nm foo | grep foo
0000000000400508 T _ZN3foo4funcEii
```

また、マングルされたシンボル（マングルドシンボル）からソース上での表記を復元する場合（デマングルを実行する場合）、c++filt コマンド（または、nm の -c オプション）を使います。

```
# nm foo | grep foo | c++filt
0000000000400508 T foo::func(int, int)
# nm -C foo | grep foo
0000000000400508 T foo::func(int, int)
```

x86_64 での引数の確認

引数の値を確認するため、上記のソースコードをビルドして、GDB で実行させます。ただし、先ほど述べたように、オブジェクトのインスタンスのアドレスが、第 1 引数として渡されます。そのため、まず、f1 と f2 のアドレスを知る必要があります。これらはソースコード中の⓪で表示されるようにしてあります。また、デバッグオプションが付加されていない場合、ブレークポイントの指定には foo::func のようなソース上の表記ではなく、下記のようにマングルドシンボルを使用します。

```
(gdb) b *_ZN3foo4funcEii          ———————— foo::func() のマングルドシンボル
Breakpoint 1 at 0x400558
(gdb) run
…
f1: 0x7fffa34c3ab0, f2: 0x7fffa34c3aa0   ——— f1 と f2 のインスタンスのアドレス
…
```

いま、プログラムは、ソースコードの①から呼ばれた foo::func() の先頭で停止しています。

```
(gdb) i r
…
rdx            0x1      1
rsi            0x5      5
rdi            0x7fffa34c3ab0   140735933070000
…
```

x86_64 における C 言語の関数では、rdi、rsi、rdx の順に引数が渡されますが（**[HACK #10]** 参照）、ソースコード上の第 1 引数と第 2 引数は、rsi と rdx を使って渡されていることがわかります。つまり、rdi には、f1 のアドレスが渡されています。実行を続けてみます。次に foo::func() がコールされた時、rdi にはソースコードの②に対応するオブジェクトのインスタンスのアドレスと引数が渡されていることがわかります。

```
(gdb) c
…
(gdb) i r
…
rdx            0x2      2
```

```
rsi            0xfffffffc       4294967292
rdi            0x7fffa34c3aa0   140735933069984
…
```

i386での引数の確認

i386では、**[HACK #11]**で説明したように、引数は基本的にスタックを経由して渡されます。先ほどと同じように引数を調べてみます。インスタンスのアドレスがスタックで渡され、その次にソースコード上での引数が渡されていることがわかります。

```
(gdb) b *_ZN3foo4funcEii
Breakpoint 1 at 0x8048454
(gdb) run
…
f1: 0xbf9874fc, f2: 0xbf9874f4  ―― f1とf2のインスタンスのアドレス
…
(gdb) x/3 $esp+4
0xbf9874e0:     0xbf9874fc      0x00000005      0x00000001
(gdb) c
…
(gdb) x/3 $esp+4
0xbf9874e0:     0xbf9874f4      0xfffffffc      0x00000002
```

まとめ

C++で記載されたプログラムにおいて、どのように引数が呼び出し先の関数へ渡されるかを説明しました。引数の他にオブジェクトのポインタも渡される点がC言語で関数コールと異なっています。

―― Kazuhiro Yamato

HACK #13 アセンブリ言語の勉強法

難解のように思われるアセンブリ言語ですが、自作のテストプログラム（TP）を逆アセンブルすることで簡単に理解できます。

GCCでソースコードをコンパイルするとマシン語（命令コード）のバイナリが作られます。コアダンプやカーネルダンプを解析するにはgdb、crashなどのデバッガにより逆アセンブルされたアセンブリ言語を見ることになります。最終的にはC言語などのソースコードでどこに不具合があるのか見つけなければなりません。

アセンブリ言語は勉強しようと思ってもなかなか手が出ないところです。本Hackではintelのアーキテクチャマニュアルを読み解くのではなく、自分でTPを作成し、感覚的

に学ぶ方法を紹介します。

逆アセンブラの出力を見る

アセンブリ言語を理解するためにはCPUのレジスタとマシン語を理解する必要があります。レジスタについては「Intel アーキテクチャの基本」[HACK #8]を参照してください。

本Hackでは簡単なC言語プログラムがどのようなアセンブリ言語になるのか見てみます。以下のassemble.cをコンパイルします。環境はFedora8 32ビットでgccのバージョンは4.1.2、objdumpバージョンは2.17.50.0.18-1になります。

```
$ cat assemble.c
#include <stdio.h>

int global;
int func_op(void) { return 0; }

void func(void)
{
    unsigned long long val64 = 0;

    val64 = 0xffffeeeeddddcccc; ――――⑦
    global = 0x5555; ――――――――――――⑧
}
#define MAX_WORD 16

int main(void)
{
    unsigned int i = 0;
    char words[MAX_WORD]="Hello World";
    char word;

    int (*func_pointer)(void) = &func_op;

    i = 0xabcd; ――――――――――――――①

    if (i != 0x1234) ――――――――――②
        i = 0; ――――――――――――――③

    while (i == 0) ――――――――――――④
        i++; ――――――――――――――⑤
```

```
    func(); ──────────────────⑥
    i = func_pointer(); ──────────⑨

    for (i=0; i<MAX_WORD-1 ; i++) ──⑩
        word = words[i]; ──────────⑪

    return 0; ──────────────────⑫
}
```

アセンブリ言語を理解しやすいようにTPを少し工夫しています。①、⑦、⑧は"i = 0"のような10進の数は入れず、アセンブリ言語を見るときにわかりやすい16進の数を入れています。またmain()ではif文②、while文④、関数コール⑥など代表的なものを書いています。

それではコンパイルしてobjdumpで逆アセンブルします。アセンブリ言語を見やすくするためgccの最適化オプションを無効（-O0：ハイフン オー ゼロ）にします。またマシン語を出力させないためにobjdumpで--no-show-raw-insnオプションを指定します。

```
$ gcc -Wall -O0 assemble.c -o assemble
$ objdump -d --no-show-raw-insn assemble
...
0804839e <func>:
 804839e:   push   %ebp
 804839f:   mov    %esp,%ebp
 80483a1:   sub    $0x10,%esp
 80483a4:   movl   $0x0,-0x8(%ebp)
 80483ab:   movl   $0x0,-0x4(%ebp)
 80483b2:   movl   $0xddddcccc,-0x8(%ebp) ──❼
 80483b9:   movl   $0xffffeeee,-0x4(%ebp)
 80483c0:   movl   $0x5555,0x80496c4 ──────❽
 80483ca:   leave
 80483cb:   ret

080483cc <main>:
 80483cc:   lea    0x4(%esp),%ecx
 80483d0:   and    $0xfffffff0,%esp
 80483d3:   pushl  -0x4(%ecx)
 80483d6:   push   %ebp
 80483d7:   mov    %esp,%ebp ──────────❶
```

```
80483d9:   push   %ecx
80483da:   sub    $0x24,%esp
…
804840a:   movl   $0xabcd,-0x10(%ebp) ―――――❶
8048411:   cmpl   $0x1234,-0x10(%ebp) ―――――❷-1 ┐
8048418:   je     8048427 <main+0x5b> ―――――❷-2 ├ if()
804841a:   movl   $0x0,-0x10(%ebp) ―――――❸      │
8048421:   jmp    8048427 <main+0x5b> ―――――❷-3 ┘
8048423:   addl   $0x1,-0x10(%ebp) ―――――❺      ┐
8048427:   cmpl   $0x0,-0x10(%ebp) ―――――❹-1 ├ while()
804842b:   je     8048423 <main+0x57> ―――――❹-2 ┘
804842d:   call   804839e <func> ―――――❻
8048432:   mov    -0x8(%ebp),%eax
8048435:   call   *%eax ―――――❾
8048437:   mov    %eax,-0x10(%ebp)
804843a:   movl   $0x0,-0x10(%ebp) ―――――❿-1 ┐
8048441:   jmp    8048452 <main+0x86>          │
8048443:   mov    -0x10(%ebp),%eax ―――――⓫-1 │
8048446:   movzbl -0x20(%ebp,%eax,1),%eax ―――⓫-2 ├ for()
804844b:   mov    %al,-0x9(%ebp) ―――――⓫-3     │
804844e:   addl   $0x1,-0x10(%ebp) ―――――❿-2 │
8048452:   cmpl   $0xe,-0x10(%ebp) ―――――❿-3 ┘
8048456:   jbe    8048443 <main+0x77>
8048458:   mov    $0x0,%eax ―――――⓬
804845d:   add    $0x24,%esp
8048460:   pop    %ecx
8048461:   pop    %ebp
8048462:   lea    -0x4(%ecx),%esp
8048465:   ret
…
```

アセンブリ言語の最初の数行では push 命令などでスタックフレームを作成しています。これらについての詳細は**「デバッグに必要なスタックの基礎知識」[HACK #9]** を参照してください。

変数に値を設定する：movl 命令

ソースコードの丸数字とアセンブリ言語の黒丸数字は対応しています。❶では movl 命令により 0xabcd を -0x10(%ebp) に値を代入しています。0xabcd を扱っていることから❶は①の変数初期化に対応していることがわかります。-0x10(%ebp) は ebp レジスタの中にあるア

ドレスから 0x10 引いたアドレス値を示します。仮に ebp レジスタにあるアドレス値が 0x801000 であれば、変数 i のアドレスは 0x801000 - 0x10 で 0x800ff0 になります。

逆アセンブラによっては -0x10 が 0xfffffff0 と表示されることもあります。

❶の「mov %esp,%ebp」だけ見ると、%esp の値を %ebp に入れるのか、%ebp から %esp なのかわかりません。しかしこのような TP で確認すると、最初にアセンブリ命令（movl）があり、その次の値（0xabcd）をレジスタ（-0x10(%ebp)）に入れるということがわかり、movl 命令は C 言語の = だということがわかります。このあとも変数 i は継続して使われるため、あとの -0x10(%ebp) も変数 i になります。

if 文での変数比較：cmpl 命令

❷ -1 は cmpl 命令で、0x1234 と -0x10(%ebp)（変数 i）を比較（compare）しています。次の❷ -2 では「i == 0x1234」であれば「i = 0」（❸）は実行されないので、je 命令（jump equal）で❹ -1 にジャンプします。cmpl 命令は変数 i と 0x1234 を比較し、真であれば CPU の ZF レジスタが 1 になります。je 命令は ZF フラグが 1（つまり i==0x1234）であればジャンプします。

while 文のアセンブリ言語

❹ -1 の cmpl 命令は while 文の条件式を判断します。i == 0 であれば❹ -2 の je 命令でジャンプし、addl 命令で i++（❺）を実行します。i != 0 であれば、❹ -2 でジャンプせずに❻に移ります。

関数の呼び出し：call 命令

call 命令はその関数にジャンプしたあと、また戻ってきます。❻は見てのとおり <func> とあり、func() 関数に処理が移ります。

❼は2回 movl 命令を繰り返して32ビットの値を入れています。32ビット OS 上で「unsigned long long」と 64 ビットの変数を定義するとアセンブリ言語ではこのようになります。

❽ではグローバル変数に値を入れています。今までローカルに定義した変数は -0x10(%ebp) のように ebp レジスタとオフセット値で表され、スタック内のアドレスに書き込んでいました。グローバル変数になると "0x80496c4" のようにアドレスで表示されます。

func() 関数の処理が終わると main() の❻に戻ります。

関数ポインタの呼び出し

関数ポインタ func_op() を呼ぶと❾のようになります。eax に関数のポインタがあるわけですが、*%eax と * が付きます。

配列の操作：movzbl 命令

⑩の for 文と⑪を順番に見ていきます。まず for 文の i=0 は❿ -1 になります。次に❿ -3 へ無条件ジャンプします。❿ -3 では変数 i と MAX_WORD-1 を比較します。for 文を続ける場合は⓫ -1 にジャンプです。⓫ -1 では変数 i を eax レジスタに入れます。さて⓫ -2 ですが、これは words（-0x20%ebp）の i（%eax）番目の 1 バイトを eax レジスタに入れる意味です。つまり word = words[i]; です。⓫ -3 の %al は eax レジスタの下位 8 ビットになります。char 型の変数で 8 ビットなためです。

int（4 バイト）の配列であれば以下のようになります。

```
8048427:   mov    -0x50(%ebp,%eax,4),%eax
804842b:   mov    %eax,-0xc(%ebp)
```

戻り値の設定

⓬は return で返す値に 0 を入れています。return で返す値が int など 4 バイト以下の場合は eax レジスタに代入されると決まっています。それまで eax レジスタは汎用的に使用されます。

まとめ

本 Hack のように TP を使うとアセンブリ言語の理解を助けることができます。-O0 で学習すると -O2 で最適化されたアセンブリ言語も理解しやすくなります。またコアダンプやカーネルダンプを解析するのに必要な知識の大半を身につけることができます。

参考文献

- IA-32 インテル ® アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル、中巻 A: 命令セット・リファレンス A-M
  http://download.intel.com/jp/developer/jpdoc/IA32_Arh_Dev_Man_Vol2A_i.pdf

- IA-32 インテル ® アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル、中巻 B: 命令セット・リファレンス N-Z
  http://download.intel.com/jp/developer/jpdoc/IA32_Arh_Dev_Man_Vol2B_

i.pdf

—— Naohiro Ooiwa

HACK #14 アセンブリ言語からソースコードの対応を調べる

crash コマンドの逆アセンブラが、ソースコードのどこを示しているか特定します。

ユーザアプリケーションの調査でも、カーネルの調査でも、コアダンプやカーネルダンプを解析するためは、アセンブリ言語からソースコードを特定する必要があります。

本 Hack では Linux カーネル 2.6.19 の journal_commit_transaction() を例にそのノウハウを紹介します。

crash で逆アセンブル

まずは journal_commit_transaction() を逆アセンブルします。

```
# crash /boot/vmlinux-2.6.19
...
crash> dis journal_commit_transaction
0xf88580e0 <journal_commit_transaction>:        push   %ebp
0xf88580e1 <journal_commit_transaction+1>:      mov    %eax,%ebp
...
0xf88585bc <journal_commit_transaction+1244>:   call   0xf88580a0 <journal_do_submit_data> ―――❷
0xf88585c1 <journal_commit_transaction+1249>:   mov    %ebx,%eax
0xf88585c3 <journal_commit_transaction+1251>:   call   0xc120e1b9 <_spin_lock> ―――――――――❸
0xf88585c8 <journal_commit_transaction+1256>:   movl   $0x0,0x24(%esp)
0xf88585d0 <journal_commit_transaction+1264>:   jmp    0xf885869c ―――――――――――――――❹
0xf88585d5 <journal_commit_transaction+1269>:   mov    0x24(%eax),%esi ―――――――――❶、❺-1
```

❶の行がソースコードのどこにあたるのかを特定してみます。NULL ポインタアクセスやメモリ破壊などの解析では、mov 命令がソースコードのどこに相当するかを調べることはよくあります。

前後の情報からソースコードを限定する

❶は journal_commit_transaction() の 1269 バイト目にあります。そのため、journal_commit_transaction+1 から順番に見ると、とても大変です。まずは前後の情報を見てみます。❷を見ると journal_do_submit_data() がコールされています。ただしソースコードでは journal_commit_transaction() から直接 journal_do_submit_data() を呼んでいません。最適化のため、明示的に __inline__ 宣言されていない static 関数も展開されています。ソースコードでは journal_do_

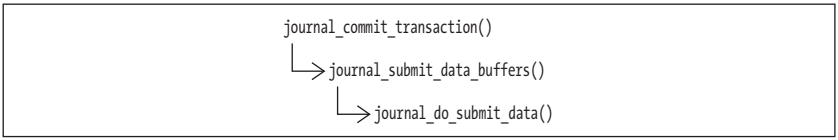

図 2-16　journal_do_submit_data() のコールシーケンス

submit_data() は図 2-16 のシーケンスで呼ばれます。

journal_submit_data_buffers() から journal_do_submit_data() が 3 カ所で呼ばれます。❷がこの 3 箇所のどこかを特定すれば、すぐにわかりそうです。

❸で spin_lock() を呼んでいます。journal_do_submit_data() のあとに spin_lock() を呼ぶのは 1 カ所で、journal_submit_data_buffers() の最後に呼ばれるところです。

[fs/jbd/commit.c]

```
static void journal_submit_data_buffers(journal_t *journal,
                                transaction_t *commit_transaction)
{
…
write_out_data:
    cond_resched();
    spin_lock(&journal->j_list_lock);

    while (commit_transaction->t_sync_datalist) {
    …
        if (buffer_dirty(bh)) {
            if (test_set_buffer_locked(bh)) {
            …
                journal_do_submit_data(wbuf, bufs);  /* 1つ目 */
                bufs = 0;
                lock_buffer(bh);                  /* これがあるので違う */
                spin_lock(&journal->j_list_lock);
            }
            locked = 1;
        }
       …
        if (locked && test_clear_buffer_dirty(bh)) {
        …
              journal_do_submit_data(wbuf, bufs);   /* 2つ目 */
               bufs = 0;
```

```
                goto write_out_data;            /* gotoしたあとはcond_resched()を */
            }                                   /* 呼ぶので違う */
        }
        …
        }
        spin_unlock(&journal->j_list_lock);
        journal_do_submit_data(wbuf, bufs);     /* このあと(journal_submit_data_buffers()を
}                                                  抜けたあと)どうなるか確認 */ ――――②
```

下記のソースコードを見てください。journal_submit_data_buffers() から戻るとすぐに spin_lock() を呼んでいます。そのため②が❷と一致していると考えて大丈夫です。

次に❹を見ると、以下の❹ -1 にジャンプします。無条件ジャンプです。

```
0xf8858697 <journal_commit_transaction+1463>:   call   0xc1023496 <cond_resched_lock>
0xf885869c <journal_commit_transaction+1468>:   mov    0x14(%esp),%edx ――――❹ -1
0xf88586a0 <journal_commit_transaction+1472>:   mov    0x18(%edx),%eax ――――❹ -2
0xf88586a3 <journal_commit_transaction+1475>:   test   %eax,%eax
0xf88586a5 <journal_commit_transaction+1477>:   jne    0xf88585d5 ――――❺
0xf88586ab <journal_commit_transaction+1483>:   mov    $0x1,%al
```

これはソースコードと照らし合わせるとわかります。**「アセンブリ言語の勉強法」[HACK #13]** の while 文と同じです。それでは以下のソースコードを見てみます。黒丸数字と丸数字の中の数字は対応しています。

```
void journal_commit_transaction(journal_t *journal)
{
    transaction_t *commit_transaction;
…
    err = 0;
    journal_submit_data_buffers(journal, commit_transaction);
                                            /* すぐにspin_lock()が呼ばれている */
    /*
     * Wait for all previously submitted IO to complete.
     */
    spin_lock(&journal->j_list_lock); ――――――――――――――③
    while (commit_transaction->t_locked_list) { ――――――――④、④ -1、④ -2
        struct buffer_head *bh;

        jh = commit_transaction->t_locked_list->b_tprev; ――⑤ -1
        bh = jh2bh(jh);
```

```
…
    }
…
```

❹で無条件にジャンプし❹ -1 へたどりつきます。ここでは何かの変数を edp に入れています。これはソースコードと一致しないので、詳細は見ません。次に❹ -2 を見ます。while() 文であれば 0x18(%edx) が commit_transaction->t_locked_list なはずです。

レジスタのオフセットと構造体のメンバを確認する

0x18(%edx) が commit_transaction->t_locked_list であることを確認します。ソースコードで確認せずに crash の struct コマンドを使います。

```
crash> struct -o transaction_t
struct: invalid data structure reference: transaction_t
crash>
```

この構造体は見られないようです。これは jbd モジュールのシンボルが解決できていないためです。そこで mod コマンドを実行します。

```
crash> mod
 MODULE   NAME            SIZE  OBJECT FILE
…
f8863100  jbd            58152  (not loaded)  [CONFIG_KALLSYMS]
…
crash>
```

シンボルが解決できず (not loaded) となっています。このようなときには mod コマンドでモジュールをロードします。

```
crash> mod -s jbd
 MODULE   NAME            SIZE  OBJECT FILE
f8863100  jbd            58152  /lib/modules/2.6.19/kernel/fs/jbd/jbd.ko
crash>
```

モジュールをロードしたので、もう一度構造体を見てみます。

```
crash> struct transaction_t
No struct type named transaction_t.
struct transaction_s {
    journal_t *t_journal;
```

```
    tid_t t_tid;
    enum {T_RUNNING, T_LOCKED, T_RUNDOWN, T_FLUSH, T_COMMIT, T_FINISHED} t_state;
    long unsigned int t_log_start;
    int t_nr_buffers;
    struct journal_head *t_reserved_list;
    struct journal_head *t_locked_list;
…
```

シンボルは解決され、構造体を確認できました。しかしこれではソースコードを見ているのと同じです。また No struct type…と出ていますが、これは typedef で定義されており、実際の構造体名が transaction_s であるためです。

[include/linux/journal-head.h]

```
typedef struct transaction_s    transaction_t;  /* Compound transaction type */
```

アセンブリ言語と対比させて見たいので -o オプションを付けて構造体メンバのオフセットを表示させます。

```
crash> struct -o transaction_s
struct transaction_s {
   [0] journal_t *t_journal;
   [4] tid_t t_tid;
   [8] enum {T_RUNNING, T_LOCKED, T_RUNDOWN, T_FLUSH, T_COMMIT, T_FINISHED} t_state;
  [12] long unsigned int t_log_start;
  [16] int t_nr_buffers;
  [20] struct journal_head *t_reserved_list;
  [24] struct journal_head *t_locked_list;
…
crash> eval 24
hexadecimal: 18       /* 16進数での値 */
    decimal: 24
      octal: 30
     binary: 00000000000000000000000000011000
crash>
```

t_locked_list のオフセットが 24 で 16 進数だと 18 なので、❹ -2 の 0x18(%edx) が commit_transaction->t_locked_list だと確認できました。上の例ではオフセットが 10 進数なため eval コマンドで変換しましたが、crash には表示を 16 進数にする hex コマンドがあります。また上で t_locked_list メンバだとわかっているので、struct コマンドでメンバを指定します。

```
crash> hex
output radix: 16 (hex)
crash> struct -o transaction_s.t_locked_list
struct transaction_s {
  [0x18] struct journal_head *t_locked_list;
}
crash>
```

これでとても見やすくなります。このように構造体のオフセットからアセンブリ言語のオフセットを比較するとソースコードとの確認が取れます。

❹ -2 のあとは test 命令で %eax レジスタ同士の AND を取ります。その結果が 0 であれば ZF フラグが 1 に設定されます。eax レジスタ（commit_transaction->t_locked_list）が NULL であれば ZF フラグが 1 になります。その次の❺では jne 命令で ZF フラグが 0 の場合にジャンプします。つまり commit_transaction->t_locked_list ！= NULL の場合は❺ -1 にジャンプし while 文の中に入ります。

❺ -1 は while 文の最初の命令になります。

そのためソースコードの⑤ -1 を見ます。commit_transaction->t_locked_list->b_tprev ですが、commit_transaction->t_locked_list はさっきの while 文の条件で使用しており eax レジスタに入っています。0x24(%eax) はさらにオフセットが付いているので commit_transaction->t_locked_list->b_tprev なはずです。確認します。

```
crash> struct -o journal_head
struct journal_head {
   [0x0] struct buffer_head *b_bh;
…
  [0x24] struct journal_head *b_tprev;
  [0x28] transaction_t *b_cp_transaction;
…
crash>
```

これでアセンブリ言語の❶は、ソースコードの jh = commit_transaction->t_locked_list->b_tprev; であることがわかりました。

ソースコードのファイル名と行数を確認する

crash の dis コマンドには -l オプションがあり、対応するソースコードのファイル名と行数を出力することができます。

```
crash> dis -l journal_commit_transaction
/root/linux-2.6.19/fs/jbd/commit.c: 281
0xf88580e0 <journal_commit_transaction>:        push   %ebp
0xf88580e1 <journal_commit_transaction+1>:      mov    %eax,%ebp
0xf88580e3 <journal_commit_transaction+3>:      push   %edi
0xf88580e4 <journal_commit_transaction+4>:      push   %esi
0xf88580e5 <journal_commit_transaction+5>:      push   %ebx
0xf88580e6 <journal_commit_transaction+6>:      sub    $0x64,%esp
/root/linux-2.6.19/fs/jbd/commit.c: 284
0xf88580e9 <journal_commit_transaction+9>:      mov    0x114(%eax),%eax
0xf88580ef <journal_commit_transaction+15>:     mov    %eax,0x1c(%esp)
/root/linux-2.6.19/fs/jbd/commit.c: 309
0xf88580f3 <journal_commit_transaction+19>:     testb  $0x8,0x0(%ebp)
0xf88580f7 <journal_commit_transaction+23>:     je     0xf8858105 <journal_commit_transaction+37>
. . .
0xf884f5d0 <journal_commit_transaction+1264>:   jmp    0xf884f69c <journal_commit_transaction+1468>
/root/linux-2.6.19/fs/jbd/commit.c: 437
0xf884f5d5 <journal_commit_transaction+1269>:   mov    0x24(%eax),%esi
include/linux/jbd.h: 324
0xf884f5d8 <journal_commit_transaction+1272>:   mov    (%esi),%ebx
```

Linux カーネル 2.6.19 のソースコードは以下になります。左の数字は行数です。

```
434     while (commit_transaction->t_locked_list) {
435         struct buffer_head *bh;
436
437         jh = commit_transaction->t_locked_list->b_tprev;
438         bh = jh2bh(jh);
439         get_bh(bh);
440         if (buffer_locked(bh)) {
```

このオプションはカーネルの CONFIG_DEBUG_INFO を有効にする必要があります。ただし完全に一致しないため、アセンブリ言語を理解してから目安として使うと便利です。

まとめ

本 Hack ではアセンブリ言語からカーネルソースコードを特定するノウハウとそのときに使用する crash の便利なコマンドを紹介しました。

参考文献

- IA-32 インテル ® アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル、中巻 A: 命令セット・リファレンス A-M
  http://download.intel.com/jp/developer/jpdoc/IA32_Arh_Dev_Man_Vol2A_i.pdf
- IA-32 インテル ® アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル、中巻 B: 命令セット・リファレンス N-Z
  http://download.intel.com/jp/developer/jpdoc/IA32_Arh_Dev_Man_Vol2B_i.pdf

—— Naohiro Ooiwa

3章 カーネルデバッグの準備
Hack #15-25

本章ではLinuxカーネルのデバッグ方法の基本を記しています。Oopsメッセージの読み方、シリアルコンソールの使い方、ネットワーク経由でのカーネルメッセージの取得、SysRqキー、各種ダンプの取得方法、crashコマンドの使い方、IPMIおよびNMI watchdogでのクラッシュダンプの取得、カーネル特有のアセンブリ言語などなど、カーネルデバッグの基本について記しています。

HACK #15 Oopsメッセージの読み方

カーネル内で致命的な問題が発生したときに出力されるカーネルメッセージOopsの読み方を説明します。

Oopsメッセージ

Oopsメッセージは、カーネル内で致命的な問題が発生した時に出力されるカーネルメッセージです。下記は、もっとも典型的なOopsメッセージのひとつである、ハンドルできないページフォルトがx86_64アーキテクチャで発生した時の例です。このOopsメッセージには、大きくエラーの概要、ロードされていたモジュール、レジスタ情報、スタックトレースが含まれています。表示の詳細は、アーキテクチャやカーネルバージョンにより少しずつ異なりますが、おおまかな表示の内容は同じです。

```
Unable to handle kernel NULL pointer dereference at 0000000000000000 RIP:
 [<ffffffff881d0002>] :demo:init_oopsdemo+0x2/0xe
PGD 18b4c067 PUD 18b52067 PMD 0
Oops: 0002 [1] SMP
last sysfs file: /block/dm-1/range
CPU 1
Modules linked in: demo nfs lockd fscache nfs_acl sunrpc ipv6 dm_multipath video sbs backlight i2c_ec
i2c_core button battery asus_acpi acpi_memhotplug ac lp floppy sg serio_raw parport_pc parport pcspkr
e1000 shpchp ide_cd cdrom dm_snapshot dm_zero dm_mirror dm_mod ata_piix libata mptspi mptscsih mptbase
```

```
scsi_transport_spi sd_mod scsi_mod ext3 jbd ehci_hcd ohci_hcd uhci_hcd
Pid: 2473, comm: insmod Not tainted 2.6.18-53.5 #1
RIP: 0010:[<ffffffff881d0002>]  [<ffffffff881d0002>] :demo:init_oopsdemo+0x2/0xe
RSP: 0000:ffff81001701be60  EFLAGS: 00010246
RAX: 0000000000000000 RBX: ffffffff883e1400 RCX: 0000000000000000
RDX: ffffffff883e1400 RSI: 0000000000000296 RDI: ffffffff802ea344
RBP: ffff81001cb98800 R08: ffff81001f669820 R09: 0000000000000020
R10: 0000000000000001 R11: 0000000000000000 R12: ffff81001cb98c00
R13: ffffffff883e1400 R14: ffff81001cb98bb8 R15: ffffc20000155f70
FS:  00002aaaaaad8210(0000) GS:ffff81001fc40840(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 000000008005003b
CR2: 0000000000000000 CR3: 0000000016ade000 CR4: 00000000000006e0
Process insmod (pid: 2473, threadinfo ffff81001701a000, task ffff81001f643820)
Stack:  ffffffff800a397c 000000000000001a 0000000000000000 000000001701be78
 0000000000000000 0000000000000000 0000000000000000 0000000000000000
 0000000000000000 0000000000000000 ffffc200001605a8 ffff81001c852ea0
Call Trace:
 [<ffffffff800a397c>] sys_init_module+0x16aa/0x185f
 [<ffffffff8000c1b0>] _atomic_dec_and_lock+0x39/0x57
 [<ffffffff8005c116>] system_call+0x7e/0x83


Code: c7 04 25 00 00 00 00 17 08 76 19 c3 ff ff 05 00 00 00 ff ff
RIP  [<ffffffff881d0002>] :demo:init_oopsdemo+0x2/0xe
 RSP <ffff81001701be60>
```

Oopsメッセージの最初の行には、Unable to handle kernel NULL pointer dereference at 0000000000000000 †とエラー内容が表示されます。その後ろのRIP: [<ffffffff881d0002>]は、エラーが発生したアドレス、すなわちRIPレジスタの値です。:demo:init_oopsdemo+0x2/0xeは、そのアドレスが、demoモジュールのinit_oopsdemo()関数の2バイト目であることを意味します。最後の/0xeは、init_oopsdemo()関数のサイズです。なお、KALLSYMSカーネルオプションがオフの時には、単に論理アドレスが表示されるだけで、モジュール名や関数名などは表示されません。その次の行PGD 18b4c067 PUD 18b52067 PMD 0は、参照しようとしたアドレス（今回は0）のページテーブル情報です。

Oops:に続く数値は、エラーコードで、その次の[]内の数値は、ページ関係のOopsメッセージが表示された回数です。その後ろには、カーネルコンフィグに応じてSMPとPREEMPTが表示されます。このメッセージを取得したカーネルは、SMPがオンで、カーネルプリエンプションはオフでしたので、SMPだけが表示されています。last sysfs file:は、最後にオープンされたsysfsのファイル名です。

† 著者訳：0番地を参照するNULLポインタを扱えない。

次の行の CPU の後ろの数字は、エラーが発生した論理 CPU の番号であり、続く Modules linked in: 以降には、ロードされているモジュールの一覧が表示されます。その次の行には、エラーが発生した時にその CPU で実行されていたプロセスのプロセス ID（2473）、プロセス名（insmod）、カーネルの汚染要因（Not tainted）、バージョン（2.6.18-53.5）が表示されます。カーネルの汚染要因には、プロプライエタリなドライバのロード（P）、モジュールの強制ロード（F）、強制的なモジュールのアンロード（R）、マシンチェック例外の発生（M）、バッドページの検出（B）があります。ひとつでも該当する事項があれば、Tainted: PF R B のような表示がされます。該当する事項がなければ、上記の例のように Not tainted と表示されます。

RIP: から CR2: までの 9 行は、エラーが発生した時のレジスタの値です。その次の行には、エラーが発生した時に実行されていたプロセスのコマンド名、プロセス ID、thread_info 構造体および task_struct 構造体のアドレスが表示されます。

Stack: から Code: までは、エラーがカーネルモードで発生した時のみ表示されます。Stack: は、スタックの先頭部分の値です。ここで表示されるサイズは、kstack カーネルオプションで指定された値になります。指定がない場合デフォルト値の 12 になります。Code: には、エラーが発生した時の RIP が示すアドレスから 20 バイト分のコードが表示されます。

最後の RIP と RSP は、エラーが発生した時の RIP と RSP の値です。この RIP と RSP のようにいくつかの内容は、重複して表示されていますが、その値はどれも当然同一です。

Oops の表示テスト

上記で例に挙げた Oops メッセージは、次のコードにより発生させました。このコードを demo.ko というモジュール名でビルドして、ロードすると、Oops が発生します。

```
static __init int init_oopsdemo(void)
{
    *((int*)0x00) = 0x19760817;
    return 0;
}
module_init(init_oopsdemo);

static __exit void cleanup_oopsdemo(void)
{
}
module_exit(cleanup_oopsdemo);
MODULE_LICENSE("GPL");
```

このコードは、モジュールの初期化関数 init_oopsdemo() の冒頭で、論理アドレス 0 に対して、書き込みを行います。アドレス 0 に対応する実ページは存在しないので、もちろん例外処理がエラーとなり、Oops メッセージが表示されます。init_oopsdemo() のアセンブリコードは次のようになっています。

```
0:   31 c0                   xor    %eax,%eax
2:   c7 04 25 00 00 00 00    movl   $0x19760817,0x0
9:   17 08 76 19
d:   c3                      retq
```

エラーの原因は、アドレス 2 の movl 命令ですので、Oops メッセージの demo:init_oopsdemo+0x2/0xe が正しいことがわかります。

まず何を見る？

Oops メッセージは、デバッグのために表示されています。crash 等のツールを使用しなくても、あるいは、ダンプが取得できないような装置や状況でも、Oops メッセージを見るだけで、エラーの原因がわかることがあります。それでは、まず、どこを見るべきでしょうか？それは、先頭のメッセージです。今回の例ですと、原因がそのまま表示されています。次は、エラーが発生したアドレスです。例では、先頭から 2 行目のメッセージにも含まれていますし、そのようなメッセージがなければ、RIP（i386 では EIP）を見ます。一般的に原因を解析するためには、アセンブリコードのその部分から見ていきます。今回の例では、RIP（EIP）は、demo モジュールの init_oopsdemo() のアドレス 2 番地を指しています。この部分を見ると、明らかに不正なアドレス 0 番地に対する書き込みを行っていることがわかります。

また、EFLAGS の 9 ビット目の IF（割り込み許可フラグ）の値も、しばしば役に立ちます。というのは、ユーザ空間のプログラムでは、割り込みの許可と禁止の操作がほとんど行われませんが、反対にカーネルではそのような操作が頻繁に行われます。そのため、割り込みが発生してはならない場所で、割り込み処理が行われた等に起因するバグは、コールシーケンスと共にこのフラグを確認すれば、比較的簡単に発見することができます。

まとめ

カーネル内で致命的な問題が発生したときに出力されるカーネルメッセージ Oops の読み方を説明しました。

—— Kazuhiro Yamato

HACK #16 minicom でシリアルコンソール接続を行う

minicom の設定方法と SysRq キーを使うための break 信号について説明します。

Linux ではシリアルコンソール接続をするのに minicom コマンドがあります。シリアルコンソールと minicom を使うとリモートからコンソール画面を確認できます。また、ネットワークにつながっていないマシンであってもリモートから操作できます。コンソールに表示されるメッセージを確認したい時、通常のディスプレイ画面ではすべてを表示しきれずにメッセージが途切れてしまうことがあります。そのような場合には minicom のログ採取機能がとても役立ちます。

シリアルコンソールと minicom の組み合わせはカーネルのデバッグをするときによく使われます。ここでは一般的な使い方とデバッグに便利な機能を紹介します。

準備

対象のマシン（ターゲットマシン）と minicom を実行するマシン（ホストマシン）とをシリアルクロスケーブルで接続します。ターゲットマシンではカーネルパラメータを設定します。以下は例になります。

```
console=ttyS0,115200n8 console=tty0
```

シリアルポート ttyS0 にボーレート 115200 と設定しています。n はパリティ、8 はビット数になります。シリアルコンソールだけでなく通常のディスプレイも接続している場合は上記のように console=tty0 も設定しておくと、両方に同時出力されるようになります。

GRUB などのブートローダの操作をシリアルコンソールから行えるようにするためには、BIOS のセットアップ画面でシリアルコンソールの設定を必要とする場合があります。

次に getty の設定がされていることを確認します。以下はターゲットマシンに RedHat 系のディストリビューションを使用している場合の例です。

```
# vi /etc/inittab
…
co:2345:respawn:/sbin/agetty ttyS0 115200 vt100-nav  /* この行を追記 */
1:2345:respawn:/sbin/mingetty tty1
…
```

さらに、シリアルコンソールから root でログインできるようにするため、以下の設定が必要になります。

```
# vi /etc/securetty
…
ttyS0  /* 追記 */
…
```

minicom を使う

ホストマシンにログインをして minicom コマンドを実行します。

```
# minicom
```

図 3-1 のような画面に切り替わります。

ヘルプ画面は Ctrl と A を同時に入力（Ctrl-A）したあとに Z キーを入力します。以降、このようなキー操作を Ctrl - A Z と表記します。図 3-2 のような画面に切り替わります。

Ctrl - A O でシリアルポートとボーレートを合わせます。カーネルパラメータに合わせて ttyS0、115200 に設定します。ボーレートが一致するとログインプロンプトが表示されます。ログインして、通常の操作も可能です。Ctrl - A L でシリアルコンソール画面のログを採取することができます。

デバッグでよく使用するのは break 信号です。Ctrl - A F（図 3-3）で送信します。

図 3-1　minicom 画面

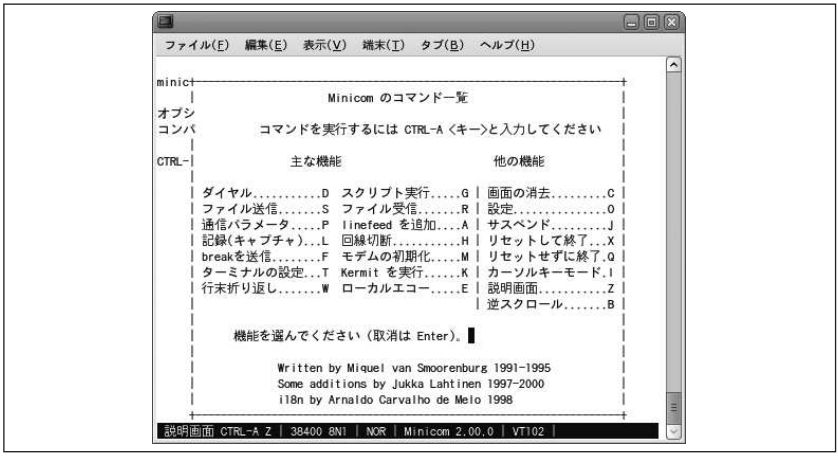

図 3-2　minicom ヘルプ画面

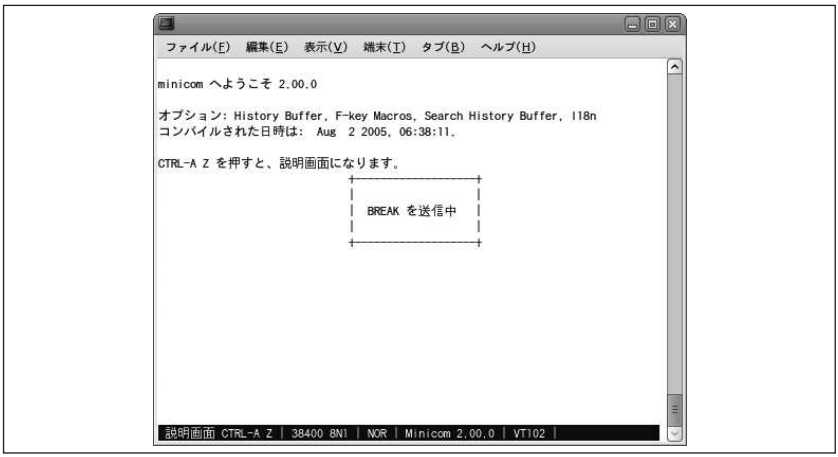

図 3-3　minicom break 信号送信画面

break 信号はキーボードから Alt キーと SysRq キーを同時に入力するのと同じ効果があります。

break 信号を送ったあと 5 秒以内に次のコマンドキーを入力します。Enter を入力すると、以下のようなヘルプメッセージが表示されます。

```
SysRq : HELP : loglevel0-8 reBoot Crash tErm kIll saK showMem powerOff showPc unRaw Sync showTasks
Unmount showCpus
```

まとめ

minicom設定と使用方法を紹介しました。minicomによりSysRqキー(break信号)をリモートから使用できます。さらにSysRqコマンドのログ採取も可能です。カーネルのデバッグでは大変便利です。SysRqキーについては「**SysRqキーによるデバッグ方法**」**[HACK #18]**を参照してください。

参考文献

- Linuxカーネルソース同梱のドキュメント
  Documentation/kernel-parameters.txt
  Documentation/serial-console.txt
  Documentation/sysrq.txt

- Remote Serial Console HOWTO
  http://www.linux.or.jp/JF/JFdocs/Remote-Serial-Console-HOWTO/index.html

—— Naohiro Ooiwa

HACK #17 ネットワーク経由でカーネルメッセージを取得する

カーネルメッセージをnetconsoleモジュールを使用し、ネットワーク経由でリモートホストに転送する方法について説明します。

カーネルパニックの調査の際に参考となるOopsメッセージはカーネルメッセージとして出力されます。本Hackでは、再起動等で失われてしまう、パニック時のOopsメッセージ等を取得するために、ネットワークを使用してカーネルメッセージを転送する方法を説明します。

netconsole機能は単体でも役に立ちますが、kdump、diskdump等のdumpの設定と合わせてサーバに設定しておくと、障害時の原因究明により役立ちます。

Oopsメッセージとは

Oopsメッセージとは Linuxカーネル内で、致命的な問題を検出した場合に出力するメッセージを指します。Oopsメッセージには、問題の原因調査に必要なCPUのレジスタ情報やシステムコールのトレース情報が含まれています。

通常このメッセージはコンソール画面には出力されますが、ログには保存されません。また、コンソールの1画面に収まらずにスクロールアウトした場合には、その部分の情報が欠けてしまい、原因調査が困難になります。

カーネルパニックの原因を調査するためには、パニック時にコンソールに出力される

メッセージをすべて採取する必要があり、そのためには事前に準備しておく必要があります。

Oops メッセージの詳細は「**Oops メッセージの読み方**」**[HACK #15]** を参照してください。

netconsole とは

netconsole モジュールは、printk メッセージ（コンソールに出力されるメッセージ）をネットワーク（UDP）経由でリモートマシンへ送信します。

netconsole モジュールはクラッシュダンプを取得するものではありません。また、通常のコンソール入出力を行うことはできません。

netconsole のメリット

netconsole はシリアルコンソールと違い、シリアルケーブルおよびシリアルポートの利用が不要です。また、複数のサーバのカーネルメッセージを 1 台のサーバに集めることが容易というメリットがあります。シリアルコンソールが使用できない環境においては、netconsole を使用することでリモートマシンに Oops メッセージを保存できる可能性が高くなります。

ネットワークに関連したカーネルパニックおよび、OS が起動してからネットワークおよび netconsole モジュールが起動するまでのパニックに伴うカーネルメッセージはシリアルコンソール等、他の手段での取得を検討して下さい。シリアルコンソールの詳細は「**minicom でシリアルコンソール接続を行う**」**[HACK #16]** を参照してください。

netconsole 設定方法

netconsole を利用する場合、メッセージ送信（netconsole モジュールを起動している）側の他に、受信するリモートマシンが必要です。

本 Hack の環境は以下のとおりです。

（送信側マシン）
IP アドレス：10.1.1.100

（ログ受信用リモートマシン）
IP アドレス：10.1.1.200

送信側設定

1. netconsole モジュールのロード

netconsole は下記のフォーマットで文字列設定パラメータ「netconsole」を扱います。

```
netconsole=[src-port]@[src-ip]/[<dev>],[tgt-port]@<tgt-ip>/[tgt-macaddr]
        src-port…送信元ポート番号
        src-ip…送信元 IP アドレス
        dev…ネットワークデバイス
        tgt-port…受信側ポート番号
        tgt-ip…受信側 IP アドレス
        tgt-macaddr…受信側 MAC アドレス
```

以下の設定では、ログ受信側リモートマシンの syslog へ送信します。ポート番号は UDP 514（syslog）を指定します。

```
# modprobe netconsole netconsole=6665@10.1.1.100/eth0,514@10.1.0.200/00:0C:29:45:EB:FA
```

2. 確認

netconsole モジュールがロードされたことを確認します。

```
# lsmod | grep netconsole
```

送信元の /var/log/messages には以下のようなメッセージが表示されます。

```
Feb 13 17:39:36 hostname kernel: netconsole: local port 6665
Feb 13 17:39:36 hostname kernel: netconsole: local IP 10.1.1.100
Feb 13 17:39:36 hostname kernel: netconsole: interface eth0
Feb 13 17:39:36 hostname kernel: netconsole: remote port 514
Feb 13 17:39:36 hostname kernel: netconsole: remote IP 10.1.0.200
Feb 13 17:39:36 hostname kernel: netconsole: remote ethernet address 00:0c:29:45:eb:fa
Feb 13 17:39:36 hostname kernel: netconsole: network logging started
```

netconsole モジュールは OS 起動時に自動でロードされません。RedHat 系ディストリビューションの場合、OS 起動時に netconsole をロードするには、/etc/sysconfig/network-scripts/ifcfg-ethX ファイルまたは /etc/rc.local ファイルに手順 1 のコマンドを記述しておきます。

受信側設定

1. syslog の設定変更

   受信側の syslog が、リモートマシンのログを受信できるように、/etc/sysconfig/syslog ファイルの SYSLOGD_OPTIONS に、-r オプションを追加します。

```
[変更前]
SYSLOGD_OPTIONS="-m 0"

[変更後]
SYSLOGD_OPTIONS="-m 0 -r"
```

2. syslogd 再起動

   変更を保存後、syslogd を再起動します。

   RedHat 系ディストリビューションの場合は以下のように設定します。

```
# service syslog restart
```

出力テスト

送信側コンソール画面に出力されたカーネルメッセージが、受信側の syslog へ転送されるか確認します。

本 Hack では SysRq キーを使用して、コンソール画面にカーネルメッセージを出力します。

1. SysRq キー有効化

   送信側マシン上で、SysRq キーを有効にするために、/etc/sysctl.conf を以下のように変更します。

```
[変更前]
kernel.sysrq = 0

[変更後]
kernel.sysrq = 1
```

   設定を有効にするために、以下のコマンドを実行します。

```
# sysctl -p
```

2. **ローカルテストと確認**
   以下のコマンドを実行し、コンソール画面にカーネルメッセージを出力します。

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

3. **リモートでの確認**
   ログ受信用リモートマシンの /var/log/messages に、以下のようなメッセージが出力されたことを確認します（hostname には送信元サーバのホスト名が表示されます）。

```
Feb 13 17:39:36 hostname SysRq :
Feb 13 17:39:36 hostname HELP :
Feb 13 17:39:36 hostname loglevel0-8
Feb 13 17:39:36 hostname reBoot
…
Feb 13 17:39:36 hostname Unmount
Feb 13 17:39:36 hostname shoWcpus
```

まとめ

本 Hack では、パニック時の Oops メッセージ等、カーネルメッセージをネットワークを使用して別サーバに転送する netconsole モジュールを紹介しました。netconsole をサーバに設定しておくと、障害時の原因究明に役立ちます。kdump、diskdump 等の dump 機能と合わせて、サーバに設定しておくとより良いでしょう。

参考

- 「diskdump を使ってカーネルクラッシュダンプを採取する」**[HACK #19]**
- 「Kdump を使ってカーネルクラッシュダンプを採取する」**[HACK #20]**

—— Shunsuke Yoshida

HACK #18 SysRq キーによるデバッグ方法

カーネルデバッグでよく使われる SysRq キーについて使い方と SysRq キーによりどのような情報が得られるか説明します。

SysRq キーはカーネルのデバッグに大変便利です。SysRq キーは割り込みを利用するためログインができないときや、キーを押して入力ができないときにも利用できます。ただしカーネルが割り込み禁止状態のままストールしている場合は使用できません（割り込み禁止でストールしている場合は NMI watchdog（「**NMI watchdog により、フリーズ時にクラッシュダンプを取得する**」**[HACK #23]** 参照）を使用してください）。

また起動中や、リブート直前でダンプが取れないような場面でも利用できます。SysRq キーで得られる情報はデバッグにとても有用です。

設定

SysRq キーを使うためにはカーネルコンフィグ CONFIG_MAGIC_SYSRQ を有効にします。

```
# make menuconfig
Kernel hacking  --->
 [*] Magic SysRq key
```

RedHat 系のディストリビューションではデフォルトで有効になっています。

起動後は sysctl で有効／無効の設定ができますが、ディストリビューションによっては起動時に無効にしています。有効にするには以下のコマンドを実行します。

```
# sysctl -w kernel.sysrq=1
```

または

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

1 を設定するとすべてのコマンドキーが使用可能です。この値はビットマスクになっており数字を組み合わせることで SysRq キーのコマンドを制限することも可能です。各値を**表 3-1** にまとめます。() 内はコマンドキーです。

表 3-1　/proc/sys/kernel/sysrq に設定するビットマスク

値	許可するコマンド
2	コンソールログレベルの制御を許可します（0-9）。
4	キーボードの制御を許可します（kr）。
8	プロセスなどの情報表示を許可します（lptwmc）。
16	Sync コマンドを許可します（s）。
32	リードオンリーでの再マウントを許可します（u）。
64	シグナルの送信を許可します（ei）。
128	リブートを許可します（b）。
256	リアルタイムプロセスの制御を許可します（q）。

Sync（s）と再マウント（u）を許可し、他の操作ができないようにするには以下のように設定します。

```
# echo 48 > /proc/sys/kernel/sysrq
```

この制御はコンソールからの入力を制限します。後述の /proc/sysrq-trigger を経由しての操作では、この /proc/sys/kernel/sysrq で制限することはできません。

またカーネルパラメータで /proc/sys/kernel/sysrq の設定を無視して常に SysRq キーを有効にすることも可能です。

```
boot> linux sysrq_always_enabled
```

このカーネルパラメータは 2.6.20 以降でサポートされています。

SysRq キーの入力方法

キーボードから入力する場合は Alt キーと SysRq キーを同時に押しながらコマンドキーを入力します。シリアルコンソールからは break 信号を送信したあとにコマンドキーを入力します。詳細は「**minicom でシリアルコンソール接続を行う」[HACK #16]** を参照してください。SysRq キーにはいくつかのコマンドがあります。マシンの制御や、情報を出力させることができます。コマンドキーとはそれらの動作を指定するためのキー入力になります。

以下のように proc ファイルシステム /proc/sysrq-trigger にコマンドキーを書き込むことで SysRq キーと同等の動作をさせる方法もあります。

```
# echo [ コマンドキー ] > /proc/sysrq-trigger
```

SysRq コマンドキー

まずはストックカーネルでバージョンごとにサポートされているコマンドを**表 3-2** にまとめました。

○はカーネルで対応していることを表します。コマンド名はコマンドキーの内容を表す名前です。例えばコマンドキー b を見ます。コマンド名は reBoot となっており B が大文字です。コマンド名で大文字のアルファベットがキーになります。

表 3-2　カーネルバージョンによるコマンドキーの対応状況

コマンドキー	コマンド名	2.6.9 〜 2.6.11	2.6.12	2.6.13 〜 2.6.15	2.6.16 〜 2.6.19	2.6.20	2.6.21	2.6.26
0-9	loglevel0-8	○	○	○	○	○	○	○
b	reBoot	○	○	○	○	○	○	○
c	Crashdump 注 1			○	○	○	○	○
d	show-all-locks(D) 注 2				○	○	○	○
e	tErm	○	○	○	○	○	○	○
f	Full		○	○	○	○	○	○
i	kIll	○	○	○	○	○	○	○
k	saK	○	○	○	○	○	○	○
l	aLlcpus 注 3							○
m	showMem	○	○	○	○	○	○	○
n	Nice		○	○	○	○	○	○
p	showPc	○	○	○	○	○	○	○
q	show-all-timers(Q) 注 4						○	○
r	unRaw	○	○	○	○	○	○	○
s	Sync	○	○	○	○	○	○	○
t	showTasks	○	○	○	○	○	○	○
u	Unmount	○	○	○	○	○	○	○
w	shoW-blocked-tasks					○	○	○

注 1：CONFIG_KEXEC を有効にする必要があります。
注 2：2.6.17 までの場合は CONFIG_DEBUG_MUTEXES を、2.6.18 から 2.6.26 では CONFIG_LOCKDEP を有効にする必要があります。
注 3：マルチ CPU 環境で CONFIG_SMP を有効にする必要があります。
注 4：CONFIG_GENERIC_CLOCKEVENTS を有効にする必要があります。

表 3-2 以外のキーを入力すると以下のようなヘルプメッセージが出力されますので、コマンドキーを確認することができます。

```
SysRq : HELP : loglevel0-8 reBoot Crashdump tErm Full kIll saK aLlcpus showMem Nice powerOff showPc
show-all-timers(Q) unRaw Sync showTasks Unmount shoW-blocked-tasks
```

デバッグで使用する主なコマンドについて**表 3-3** にまとめます。

表 3-3 SysRq コマンドキーの内容詳細

コマンドキー	説明
0-9	コンソールログレベルを設定します。これは /proc/sys/kernel/printk を設定するのと同じです。SysRq キーで情報が出力されるときだけこのレベルは自動的に 7 または 8 になります。そのため SysRq キーで情報を表示するときは意識する必要はありません。
b	リブート処理が実行されます。
c	クラッシュダンプ（kdump）を取得します。故意にパニックさせるときに使用します。
d	取得されているすべてのロックを出力します。TASK_RUNNING 状態のプロセスが取得しているロックは表示されません。これはロックがすぐに解放される可能性があるためです。カーネルが全プロセスを参照するのに使用する tasklist_lock が取得中であっても強制的に出力します。
e	init（PID が 1）以外のすべてのプロセスに SIGTERM を送信します。
f	OOM Killer を動作させます（詳しくは「**OOM Killer の動作と仕組み**」**[HACK #56]** 参照）。
i	init（PID が 1）以外のすべてのプロセスに SIGKILL を送信します。
l	システムにある全 CPU のスタックを出力します。プロセスのバックトレースも表示されます。
n	すべてのリアルタイムプロセスを強制的に通常のプロセスにします。これは sched_setscheduler(2) でスケジューリングポリシーに SCHED_NORMAL を指定したのと同じです。
m	メモリ、スワップの状態を出力します。
p	CPU のレジスタと動作しているプロセスの情報を出力します。CPU が複数の場合はキー割り込みを処理した CPU の情報のみ出力されます。
q	動作しているすべてのタイマの情報を出力します。
s	すべてのファイルシステムで sync（メモリ上のバッファをディスクに書き出す）を試みます。内部では pdflush を強制的に動作させます。コマンドキー b の前に実行すると安全にリブートできます。
t	動作しているすべてのプロセスのスタック・バックトレースを出力します。
u	すべてのファイルシステムに対しリードオンリーでの再マウントを試みます。
w	UNINTERRUPTABLE でシグナルを無視したまま待機状態になっているプロセスの情報だけを出力します。

RHEL のコマンドキー w はストックカーネルと違います。RHEL のコマンドキー w（showcpus）はシステムにある全 CPU のスタックを出力します。これはストックカーネルのコマンドキー l（allcpus）と同じになります。RHEL4/5 の対応を**表 3-4** にまとめます。

表 3-4　RHEL4/5 の SysRq キー対応状況

コマンドキー	コマンド名	RHEL4	RHEL5
0-9	loglevel0-8	○	○
b	reBoot	○	○
c	Crashdump	○	○
d	show-all-locks(D)		○
e	tErm	○	○
f	Full		○
i	kIll	○	○
k	SAK	○	○
m	showMem	○	○
n	Nice		○
p	showPc	○	○
r	unraw	○	○
s	Sync	○	○
t	showTasks	○	○
u	Unmount	○	○
w	shoWcpus		○

ストックカーネルの SysRq キー表示例

以下はコマンドキー m（showMem）の出力例です。メモリの使用量などが出力されます。

```
SysRq : Show Memory
Mem-info:
Node 0 DMA per-cpu:
...
Active:128436 inactive:87353 dirty:93 writeback:0 unstable:0
 free:6411 slab:30787 mapped:1294 pagetables:435 bounce:0

Swap cache: add 0, delete 0, find 0/0
Free swap  = 1020116kB
Total swap = 1020116kB
261920 pages of RAM
5716 reserved pages
9262 pages shared
0 pages swap cached
```

以下はコマンドキー t（showTasks）の出力例です。

```
SysRq : Show State
  task                        PC stack   pid father
…
sshd          S ffffffff8048d5e0     0  3121   2908
 ffff81003ec31a28 0000000000000082 0000000000000000 ffff810032c455b8
…
Call Trace:
 [<ffffffff80471eb6>] schedule_timeout+0x1e/0xad
 [<ffffffff8036bc83>] tty_poll+0x5f/0x6d
…
 [<ffffffff802982f4>] sys_select+0xc1/0x183
 [<ffffffff8028c00a>] sys_write+0x45/0x6e
…
```

コマンドキー w（showW-blocked-tasks）でも同様の情報が出力されます。

以下はコマンドキー l（allcpus）の出力例です。一部省略していますが、組み込まれているモジュール、SysRq キーのハンドラが動作した CPU のレジスタ値、スタック、バックトレースが出力されます。

```
SysRq : Show backtrace of all active CPUs
CPU 0:
Modules linked in: ipmi_watchdog ipmi_devintf ipmi_si ipmi_msghandler
…
Pid: 0, comm: swapper Not tainted 2.6.26 #2
RIP: 0010:[<ffffffff80211d85>]  [<ffffffff80211d85>] mwait_idle+0x41/0x44
RSP: 0018:ffffffff8087df60  EFLAGS: 00000246
RAX: 0000000000000000 RBX: 0000000000000000 RCX: 0000000000000000
…
Call Trace:
 [<ffffffff8020ab7b>] ? cpu_idle+0x6d/0x8b

CPU1:
…
Call Trace:
 <IRQ>  [<ffffffff8037c71d>] showacpu+0x0/0x52
 [<ffffffff8037c75d>] showacpu+0x40/0x52
 [<ffffffff8021a118>] smp_call_function_interrupt+0x3b/0x62
 [<ffffffff8020c8d6>] call_function_interrupt+0x66/0x70
 <EOI>  [<ffffffffa0049994>] :ext3:ext3_bmap+0x0/0x78
…
```

コマンドキー p（showPc）も同様の情報が出力されます。

コマンドキー q（show-all-timers）では /proc/timer_list と同じ情報が出力されます。

Collect scheduler debugging info（CONFIG_SCHEDSTATS）が有効の場合、コマンドキー t と w で出力される情報に /proc/sched_debug と同じ情報が追加されます。

まとめ

本 Hack では SysRq キーについて説明しました。バージョンが上がるごとに便利な機能が追加されています。SysRq キーの実装は単純ですので使いたいキーがあれば、バックポートをすると良いかもしれません。カーネルが割り込み許可状態でストールする場合は watchdog を無効にして、ストールを再現させます。その状態でコマンドキー l か p を複数回実行すると、ストールしている箇所がわかります。

—— Naohiro Ooiwa

HACK #19 diskdump を使ってカーネルクラッシュダンプを採取する

RHEL4 などに採用されている diskdump の使い方について説明します。

diskdump は RHEL4 等、RedHat 系の一部のディストリビューションで採用されているカーネルクラッシュダンプ機能です。ここで紹介する機能の一部は、ディストリビューションによっては使用できないものもあるので注意してください。本 Hack では RHEL4.7 にて確認した手順を紹介します。使用するアーキテクチャは x86_64 です。

制限事項を理解する

diskdump はカーネルパニックなど、障害の発生したカーネルでダンプを採取する機能です。したがってダンプ機能自体がうまく動作しないケースが考えられます。そのうちよくあるのが割り込みハンドラや割り込み禁止区間での障害、ある種のスピンロックでデッドロックが発生した際にダンプを取るようなケースです。diskdump ではダンプ中は割り込み禁止にすることによって、これらの障害発生時でもなるべくダンプが失敗しないようにしています。diskdump では割り込み禁止状態でディスク I/O をするために、ディスクドライバが polling I/O に対応している必要があります。したがって diskdump に対応するディスクドライバが限られてしまうという制限があります。RHEL4.7 では以下のドライバが対応しています。

aic7xxx
aic79xx

ipr
megaraid
mptfusion
sym53c8xx
sata_promise
ata_piix
CCISS
megaraid_sas
IDE
qla2xxx
lpfc
stex
ips
ibmvscsi
sata_nv
aacraid

また diskdump は、直接ディスクドライバにアクセスするため、LVM や device mapper の上に作成されたディスクパーティションへの書き込みはできません。

クラッシュダンプを有効化する

diskdump ではダンプ用パーティションの指定が必要です。ダンプ専用パーティションを用意してもよいですし、swap パーティションにダンプさせることも可能です。ただし、システムの実装メモリサイズ以上の大きさのパーティションである必要があります。今回はスワップパーティションではなくダンプ専用パーティションとして /dev/sda3 を使うことにします。設定ファイル /etc/sysconfig/diskdump に次のように記載します。

```
DEVICE=/dev/sda3
```

次に /dev/sda3 をダンプ用パーティションとしてフォーマットします。

```
# service diskdump initialformat
```

diskdump サービスを有効化します。

```
# chkconfig diskdump on
```

```
# service diskdump start
```

diskdump が有効化されたかどうかは service コマンドあるいは /proc/diskdump で確認できます。/proc/diskdump では次のように表示されるはずです。

```
# cat /proc/diskdump
# sample_rate: 8
# block_order: 2
# fallback_on_err: 1
# allow_risky_dumps: 1
# dump_level: 0
# compress: 0
# total_blocks: 98197
#
sda3 14329980 2441880
```

また、ダンプ採取完了後に自動的にリブートするように sysctl 変数 kernel.panic を設定しておきます。これには /etc/sysctl.conf に設定します。

```
kernel.panic=10
```

これでダンプ終了後 10 秒程でリブートがかかるようになります。設定を記述したら sysctl コマンドで設定を有効化します。

```
# sysctl -p
```

これで設定は完了です。クラッシュダンプを採取してみましょう。

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

ダンプファイルはリブート後に /var/crash/127.0.0.1-<日付>/vmcore として保存されます。crash コマンドで内容を確認してみてください。crash コマンドについては「**crash コマンドの使い方**」**[HACK #21]** を参考にしてください。

圧縮と部分ダンプ機能を利用してダンプファイルのサイズを小さくする

diskdump でも Kdump と同様にダンプファイルのサイズを小さくすることができます。Kdump については「**Kdump を使ってカーネルクラッシュダンプを採取する**」**[HACK #20]** を参照してください。

圧縮機能や部分ダンプ機能はdiskdumpモジュールのオプションで指定することができます。圧縮機能を有効にするにはcompressオプションに1を指定します。部分ダンプ機能はdump_levelオプションにダンプレベルを指定します。**表3-5**はダンプレベルごとにスキップするページのタイプを表しています。

表3-5 スキップするページの種類

ダンプレベル	キャッシュページ	キャッシュプライベート	ゼロページ	フリーページ	ユーザページ
0					
1	×	×			
2			×		
3	×	×	×		
4				×	
5	×	×		×	
6			×	×	
7	×	×	×	×	
8					×
9	×	×			×
10			×		×
11	×	×	×		×
12				×	×
13	×	×		×	×
14			×	×	×
15	×	×	×	×	×
17	×				
19	×		×		
21	×			×	
23	×		×	×	
25	×				×
27	×		×		×
29	×			×	×
31	×		×	×	×

ただし、この機能を有効にするには注意が必要です。ページによってはスキップさせるためにカーネル内部のメモリ管理用のリストを検索する必要があります。仮に障害が、そのリストが破壊されてしまったため起こったのだとしたら、diskdumpがリストを検索する中で二重パニックを起こしたり、ストールしてしまうこともあるからです。この機能を

使うのであれば、おすすめはダンプレベル 19 です。19 であればリスト検索を必要としないからです。次のように /etc/modprobe.conf に diskdump モジュールへ渡すオプションを記述することで使用できます。

```
options diskdump dump_level=19 compress=1
```

この設定を有効化するために diskdump サービスを再起動します。

```
# service diskdump restart
```

設定したオプションが正しくロードされているかは /proc/diskdump で確認できます。

```
# cat /proc/diskdump
# sample_rate: 8
# block_order: 2
# fallback_on_err: 1
# allow_risky_dumps: 1
# dump_level: 19
# compress: 1
# total_blocks: 98197
#
sda3 14329980 2441880
```

障害発生時にメールで通知する

diskdump では vmcore ファイルを /var/crash/ 以下に保存した後にユーザ定義のスクリプトを起動させる機能があります。これを利用して、ダンプが採取された時（つまりシステム障害が発生した時）にメールで通知させることもできます。これはサンプルが /usr/share/doc/diskdumputils-<version>/example_scripts/ にあるので試してください。今回利用するのは diskdump-success スクリプトです。これを /var/crash/scripts/ 配下にコピーし、次のように編集します。

```
# cat /var/crash/scripts/diskdump-success
#!/bin/sh

ADDRESS=tabe@miraclelinux.com

mail -s "[diskdump] `hostname` crashed" $ADDRESS <<_EOF
The machine `hostname` crashed.
```

```
Writing crash dump to $1
_EOF

# savecore always returns 0 whatever the result of this script because this is
# called after a dump file is created.

exit 0
```

ダンプ出力先デバイスを冗長化する

冒頭の「制限事項を理解する」（107ページ）でも述べましたが、diskdumpはファイルシステムを介さず、直接ディスクドライバにアクセスします。したがって例えばダンプ用パーティションで使用しているディスクドライバで障害が発生した場合、ダンプ採取に失敗する可能性があります。そこで、diskdumpでは複数のパーティションをダンプ用パーティションとして指定できるようになっています。具体的には/etc/sysconfig/diskdumpで次のように設定します。

```
DEVICE=/dev/sda3:/dev/hda
```

/dev/sda3は今まで使っていたダンプ専用パーティションです。筆者の環境ではmptfusionドライバが動作しています。一方、/dev/hdaはIDEドライバが動作する別ディスクデバイスです。仮にmptfusionドライバで障害が発生してしまった場合、/dev/sda3へのダンプの書き込みが失敗してしまうかもしれません。そのような場合、diskdumpは次に登録された/dev/hdaにダンプするようになります。こちらはIDEドライバなため、問題なくダンプの書き込みができます。

まとめ

Kdumpがメインラインにマージされるまでは、さまざまなダンプ機能が提案されていて、ディストリビューションごとに採用するものが異なっていました。netdumpやLKCD、mkdump、本Hackで紹介したdiskdumpもそのひとつです。本Hackでは、その中でもユーザが多いdiskdumpを取り上げ、カーネルクラッシュダンプの採取方法について説明しました。

参考

ここでは紹介しなかった詳細な設定についてdiskdumputilsのREADMEに記載されています。

/usr/share/doc/diskdumputils-<version>/README

—— Toyo Abe

HACK #20 Kdump を使って カーネルクラッシュダンプを採取する

最近のディストリビューションで採用されている Kdump の使い方について説明します。

Kdump は linux-2.6.13 からメインラインに取り込まれたカーネルクラッシュダンプ機能です。バージョン 2.6.13 以降のカーネルを使用した Linux ディストリビューションであれば使うことができます。本 Hack では RHEL5.1 にて確認した手順を紹介します。使用するアーキテクチャは x86_64 です。

クラッシュダンプを有効化する

カーネルブートパラメータに crashkernel=128M@16M を追加します。RHEL5.1 であれば /etc/grub.conf を次のように編集します。

```
title Red Hat Enterprise Linux Server (2.6.18-53.1.21.el5)
        root (hd0,0)
        kernel /boot/vmlinuz-2.6.18-53.1.21.el5 ro root=LABEL=/1 crashkernel=128M@16M rhgb quiet
        initrd /boot/initrd-2.6.18-53.1.21.el5.img
```

これは物理アドレスの 0x1000000 番地から 128MB のメモリをダンプカーネル用に予約することを意味しています。grub.conf を編集したら、この設定を有効化するため、いったんリブートします。

次に kdump サービスを有効化します。chkconfig コマンド、service コマンドを使います。

```
# chkconfig kdump on
# service kdump start
```

設定に成功したかどうかは service コマンド、あるいは /sys/kernel/kexec_crash_loaded によって確認することができます。

service コマンドでは次のように表示されれば設定が有効化されています。

```
# service kdump status
Kdump is operational
```

kexec_crash_loaded の中身が 1 であれば設定が有効化されています。

```
# cat /sys/kernel/kexec_crash_loaded
1
```

ここまでできたら試しにクラッシュダンプを採取してみましょう。

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

ダンプが成功すると、リブート後に /var/crash/ 配下にディレクトリが作成され、vmcore というファイルができています。これを crash コマンドで確認すればよいのです。crash コマンドについては「**crash コマンドの使い方**」**[HACK #21]** を参考にしてください。

makedumpfile を使ってダンプのファイルサイズを小さくする

これまで説明した設定では実装メモリ量と同じサイズのクラッシュダンプが作成されます。8GB のメモリを積んでいればダンプファイルも 8GB になるということです。しかし、Kdump でも diskdump と同様にダンプイメージを圧縮し、より小さなサイズとすることができます。そのためのユーティリティが kexec-tools に含まれる makedumpfile というコマンドです。これを利用するためには /etc/kdump.conf に core_collector の設定を追加します。

```
ext3 /dev/sda5
core_collector makedumpfile -c
```

最初の「ext3 /dev/sda5」は root ファイルシステムがあるデバイスを指定しています。ダンプファイルはこのパーティションの ./var/crash 配下に出力されます。ダンプ出力先を別パーティションとしたい場合、例えば /dump というディレクトリにマウントしている /dev/sda6 にしたい場合、以下のようにします。

```
ext3 /dev/sda6
path .
```

こうすると、/dump ディレクトリの下に日付のディレクトリが作成され、そこにダンプが出力されるようになります。

-c は圧縮するというオプションです。他にもダンプレベルを設定する -d オプションが便利です。ダンプレベルオプションは、クラッシュダンプに含めないページ（メモリ）タイプを指定します。**表 3-6** はダンプレベルごとにスキップするページのタイプを表しています。

表 3-6　スキップするページの種類

ダンプレベル	ゼロページ	キャッシュページ	キャッシュプライベート	ユーザデータ	フリーページ
0					
1	×				
2		×			
4		×	×		
8				×	
16					×
31	×	×	×	×	×

ダンプレベルは**表 3-6** の数値の和で指定できます。例えばゼロページとフリーページを含めたくない場合ダンプレベルに 1+16=17 を指定します。筆者は次のような設定で使っています。

```
core_collector makedumpfile -c -d 1
```

次にデバッグ情報付きでコンパイルされたカーネルをインストールします。RHEL5.1 では kernel-debuginfo と kernel-debuginfo-common パッケージです。

```
# rpm -ivh kernel-debuginfo-2.6.18-53.1.21.el5.x86_64.rpm \
  kernel-debuginfo-common-2.6.18-53.1.21.el5.x86_64.rpm
```

kdump サービスを再起動してください。

```
# service kdump restart
```

すでに説明した方法でクラッシュダンプを採取してみてください。採取したダンプが圧縮されているかどうかは vmcore ファイルのサイズを見ればわかると思います。圧縮されたダンプファイルは ELF フォーマットではなくなるため gdb を使ったデバッグはできません。crash コマンドを利用してください。

自分でリビルドしたカーネルのダンプを makedumpfile で取る場合は、デバッグ情報付きカーネルを次の場所に配置する必要があります。

```
/usr/lib/debug/lib/modules/`uname -r`/vmlinux
```

Fedora9 などの最近のカーネルを組み込んだディストリビューションでは、デバッグ情報付きのカーネルパッケージが不要になっています。カーネルに vmcoreinfo という機能が追加されたためです。これに合わせて kexec-tools パッケージが変更されています。使っているシステムのカーネルと kexec-tools が共に vmcoreinfo に対応していれば、デバッグ情報付きのカーネルをいちいちインストールする必要はありません。2008 年 9 月 16 日現在の RHEL5.1 の最新版（kexec-tools-1.102pre-21.el5.x86_64.rpm）ではすでに対応されています。

リモートサーバへクラッシュダンプを転送する

/etc/kdump.conf に net の設定を追加します。

```
/* NFS マウントして転送する場合 */
net ‹サーバ名あるいは IP アドレス›:‹ export されたディレクトリ›
/* SSH 経由で転送する場合 */
net ‹ユーザ名›@‹サーバ名あるいは IP アドレス›
```

NFS であれば export されたディレクトリ配下に ./var/crash というディレクトリを作成しておく必要があります。SSH であれば、リモートサーバの /var/crash ディレクトリ配下にダンプを転送します。ですのでここで設定するログインユーザが /var/crash への書き込み権限を持っている必要があります。セキュリティ上、好ましくないのであればダンプ用ディレクトリを別に用意し、次のようにしてダンプ先ディレクトリを変更するとよいでしょう。

```
path /dump
```

SSH では path に指定するディレクトリは相対パスではなくフルパスになります。また、パスワード入力なしでログインできるように公開鍵を登録しておくなどの設定が必要です。Kdump の init スクリプトの propagate オプションを利用すると、この作業をスクリプトが行ってくれます。

```
# service kdump propagate
```

link_delay には NIC をリンクアップしてから転送を開始するまでの間に入れる待ち時間を秒単位で指定します。筆者は念のため link_delay の設定を追加しています。

```
link_delay 10
```

makedumpfile と SSH の組み合わせで使う場合、ダンプファイルの変換が必要になります。リモートサーバに出力したダンプファイルを見ると、ファイル名が vmcore.flat となっています。そのままでは crash コマンドから読み込めないため、次のようにしてファイル形式を変換してください。

```
# makedumpfile -R vmcore < vmcore.flat
```

まとめ

Kdump を使ってカーネルクラッシュダンプを採取する方法について説明しました。Kdump ではパニックが発生した際、ディスクコントローラなどのデバイスの終了処理を行わずに kexec を使ってダンプカーネルを起動します。したがってパニック時のデバイスの状態によってはダンプカーネルの起動に失敗するケースがあります。筆者が経験した例では、ディスク I/O 中にパニックが発生すると、ダンプカーネルがディスクコントローラドライバの初期化に失敗し、ダンプが取れないという現象がありました。その時はドライバの初期化時にディスクコントローラをリセットする処理を追加することで回避しました。

—— Toyo Abe

HACK #21 crash コマンドの使い方

crash の便利なコマンドと使用方法を紹介します。

crash にはさまざまなコマンドがそろっています。適切に使うことで欲しい情報が簡単に得られます。

crash を起動すると、プロンプトが表示され対話形式で操作ができるようになります。本 Hack では crash の便利なコマンドや、デバッグで有用なものを紹介します。

コマンド出力は 2.6.18 カーネルを例にしています。

crash には vmcore のようなクラッシュダンプファイルを見る機能と、ライブシステムを見る機能があります。ライブシステムはクラッシュダンプファイルではなく動作中のカーネルを見る機能で、一部制限があります。これについてはその都度説明します。

crash の起動

crash コマンドでクラッシュダンプファイルを見るには、以下のように実行します。

```
# crash vmlinux vmcore
```

vmlinux はカーネルの非圧縮イメージです。RedHat 系のディストリビューションであれ

ば、kernel-debuginfo の RPM パッケージを展開すると /usr/lib/debug/lib/modules/[バージョン名]/ に vmlinux があります。vmcore はダンプ機能で取得したクラッシュダンプファイルです。

crash コマンドのライブシステムは以下のように実行します。

```
# crash vmlinux
```

vmlinux は動作しているカーネルの vmlinux ファイルを指定します。

ユーティリティ

まずは crash を使用する上でのユーティリティを紹介します。

set コマンド

set コマンドは幅広いコマンドです。プロセスを指定するとそのコンテキストを表示します。指定がないときはカーネルパニックしたときに動作していたプロセスを表示します。

```
crash> set
    PID: 4525
COMMAND: "umount"
   TASK: 101040df7f0  [THREAD_INFO: 1009face000]
    CPU: 1
  STATE: TASK_RUNNING (PANIC)
```

-c オプションで CPU を指定でき、その CPU で動作していたプロセスを表示します。また -p オプションでカーネルパニックしたときに動作していたプロセスを表示します。

set コマンドでは現在のエディタ設定を確認できます。

```
crash> set -v | grep edit
          edit: vi
```

エディタ設定を変更するには以下のように crash 起動時に指定します。

```
# crash -e [vi | emacs] ...
```

このエディタ設定により、crash のコマンド入力画面でのキーバインドに vi スタイルか emacs スタイルが選択できます。emacs に設定すると bash と同じようなキーバインドになります。

crash では大量の情報を出力するときがあります。以下のコマンドでスクロールを無効

にできます。

```
crash> set scroll off
```

または以下のコマンドでも可能です。

```
crash> sf
```

sf はエイリアスで設定されており、set scroll off を省略したものです。デフォルトのエイリアスは alias コマンドで確認できます。

```
crash> alias
ORIGIN   ALIAS    COMMAND
builtin  man      help
builtin  ?        help
builtin  quit     q
builtin  sf       set scroll off
builtin  sn       set scroll on
builtin  hex      set radix 16
builtin  dec      set radix 10
builtin  g        gdb
builtin  px       p -x
builtin  pd       p -d
builtin  for      foreach
builtin  size     *
builtin  dmesg    log
builtin  last     ps -l
```

hex、eval コマンド

hex、eval コマンドは「**アセンブリ言語からソースコードの対応を調べる**」**[HACK #14]** を参照してください。

ascii コマンド

ascii コマンドは 16 進数を文字列に変換します。

```
crash> rd modprobe_path 2
ffffffff813213a0:  6f6d2f6e6962732f 000065626f727064   /sbin/modprobe..
crash> ascii 6f6d2f6e6962732f
6f6d2f6e6962732f: /sbin/mo
```

h コマンド

h コマンドは入力したコマンドの履歴を表示します。

```
crash> h
[1] set
[2] set vi
[3] set scroll off
[4] sf
[5] alias
[6] hex
[7] eval
```

カーネルの情報を参照するコマンド

ここではカーネル内部の情報を参照するコマンドを説明します。

bt コマンド

bt コマンドはプロセスのバックトレースを出力します。使用頻度の高いコマンドです。全プロセスのバックトレースを表示させるには以下のようにすると便利です。

```
crash> foreach bt -tf
```

-t オプションはスタックにテキストシンボルがある場合はすべて表示します。

```
crash> bt 2157
PID: 2157   TASK: ffff81007e095040  CPU: 1   COMMAND: "syslogd"
 #0 [ffff810075a5f938] schedule at ffffffff80061f29
 #1 [ffff810075a5fa20] schedule_timeout at ffffffff800627cd
 #2 [ffff810075a5fa70] do_select at ffffffff8001137f
 #3 [ffff810075a5fc10] journal_stop at ffffffff880327ae
...
crash> bt -t 2157
PID: 2157   TASK: ffff81007e095040  CPU: 1   COMMAND: "syslogd"
             START: thread_return (schedule) at ffffffff80061f29
  [ffff810075a5fa20] schedule_timeout at ffffffff800627cd
  [ffff810075a5fa40] add_wait_queue at ffffffff800477f9
  [ffff810075a5fa70] do_select at ffffffff8001137f
  [ffff810075a5fb08] __pollwait at ffffffff8001e2cf
  [ffff810075a5fb38] default_wake_function at ffffffff80089830
  [ffff810075a5fc10] journal_stop at ffffffff880327ae
...
```

-f オプションはフレーム内のスタックデータをすべて表示します。このオプションは関数への引数を確認するときに便利です。

-l オプションはファイル名と行数を表示します。

```
crash> bt -l 2157
PID: 2157   TASK: ffff81007e095040  CPU: 1   COMMAND: "syslogd"
 #0 [ffff810075a5f938] schedule at ffffffff80061f29
    /usr/src/debug/kernel-2.6.18/linux-2.6.18.x86_64/kernel/sched.c: 1840
 #1 [ffff810075a5fa20] schedule_timeout at ffffffff800627cd
    /usr/src/debug/kernel-2.6.18/linux-2.6.18.x86_64/kernel/timer.c: 1543
 #2 [ffff810075a5fa70] do_select at ffffffff8001137f
    /usr/src/debug/kernel-2.6.18/linux-2.6.18.x86_64/fs/select.c: 288
 #3 [ffff810075a5fc10] journal_stop at ffffffff880327ae
 #4 [ffff810075a5fc80] __generic_file_aio_write_nolock at ffffffff80015e21
...
```

-a オプションではカレントプロセスのみを表示します。しかしライブシステムの場合はカレントプロセスを表示できません。

```
crash> bt -a
bt: -a option not supported on a live system
```

task コマンドでスタックポインタを取得して、それを rd -s で見るとライブシステムでもカレントプロセスのバックトレースに近い情報を得ることができます。

```
crash> task | grep rsp
    rsp0 = 0xffff81004c57a000,
    rsp = 0xffff81004c579938,
    userrsp = 0x7fffd81bcb18,
crash> rd 0xffff81004c579938 -e 0xffff81004c57a000 -s
...
ffff81004c5799d8:  ffff81004c579a18 __wake_up+0x38
ffff81004c5799e8:  ffff81007c578000 ffff81007c578018
ffff81004c5799f8:  ffff81007c578018 0000000000000145
ffff81004c579a08:  ffff81004c579b60 remove_wait_queue+0x1c
ffff81004c579a18:  ffff81004c579b58 0000000000000000
ffff81004c579a28:  0000000000000296 free_poll_entry+0x11
ffff81004c579a38:  ffff81004c579b90 poll_freewait+0x29
ffff81004c579a48:  0000000000000008 0000000000000001
ffff81004c579a58:  ffff81007faf9280 0000000000000001
ffff81004c579a68:  0000000000000001 do_select+0x445
```

```
ffff81004c579a78:  ffff81000237f880 ffff81004c579f50
ffff81004c579a88:  000000017ff50030 ffff81004c579dd8
…
ffff81004c579f68:  00007fffd81bd920 00007fffd81be401
ffff81004c579f78:  00007fffd81be3a0 system_call+0x7e
ffff81004c579f88:  0000000000000246 0000000000000001
ffff81004c579f98:  0000000000000001 000000004c579fa0
…
```

dev コマンド

dev コマンドはキャラクタデバイスの一覧を表示します。-p オプションで PCI データを表示します。これは lspci コマンドと同じ内容になります。また -i オプションで I/O ポートと I/O メモリを表示します。以下のコマンドとほぼ同じ内容になります。

```
# cat /proc/ioports
# cat /proc/iomem
```

dis コマンド

dis コマンドは逆アセンブルをするコマンドです。詳細は「**リアルタイムプロセスのストール**」**[HACK #40]** などを参照してください。

files コマンド

files コマンドはプロセスがオープンしていたファイルを表示します。「**カーネルのストール（セマフォ編）**」**[HACK #39]** で実際に使用しています。詳細はそちらを参照してください。

irq コマンド

irq コマンドはカーネル内部で管理している割り込みの情報を表示します。

kmem コマンド

カーネルのメモリに関する情報を表示します。-s オプションはスラブキャッシュの情報を表示します。/proc/slabinfo と同等の情報になります。

```
crash> kmem -s
CACHE            NAME           OBJSIZE  ALLOCATED  TOTAL  SLABS  SSIZE
ffff81007c5c6300 ip_fib_alias        64         21     59      1     4k
ffff81007b51f2c0 ip_fib_hash         64         18     59      1     4k
ffff81007ac50280 fib6_nodes          64         35     59      1     4k
ffff81007ac51240 ip6_dst_cache      320         29     48      4     4k
…
```

-i オプションはメモリの情報になります。free コマンドと同等です。

```
crash> kmem -i
              PAGES        TOTAL      PERCENTAGE
 TOTAL MEM   514976         2 GB         ----
      FREE   421511       1.6 GB   81% of TOTAL MEM
      USED    93465     365.1 MB   18% of TOTAL MEM
    SHARED        0            0    0% of TOTAL MEM
   BUFFERS     3906      15.3 MB    0% of TOTAL MEM
    CACHED    53504       209 MB   10% of TOTAL MEM
      SLAB     3709      14.5 MB    0% of TOTAL MEM
...
 TOTAL SWAP   512069         2 GB         ----
 SWAP USED         0            0    0% of TOTAL SWAP
 SWAP FREE    512069         2 GB  100% of TOTAL SWAP
...
crash>
```

-p オプションでメモリマップを表示します。アドレスを指定することもできます。[] で囲まれている場合はまだ解放されていないことを示します。

```
crash> kmem ffff81004fd64048
CACHE            NAME                 OBJSIZE  ALLOCATED     TOTAL  SLABS  SSIZE
ffff81007f6c2380 ext3_inode_cache         760      73290     73295  14659     4k
SLAB             MEMORY            TOTAL  ALLOCATED  FREE
ffff81004fd64000  ffff81004fd64048      5          5     0
FREE / [ALLOCATED]
  [ffff81004fd64048]

      PAGE        PHYSICAL      MAPPING       INDEX CNT FLAGS
ffff8100018b7de0 4fd64000      -------       -----   1 48080000000080
```

あるメモリを参照してカーネルパニックなどが発生した場合は、このように確認するとすでに解放されたのかがわかります。

list コマンド

list コマンドは list_head 構造体をたどって順にアドレスを表示します。

```
crash> whatis modules
struct list_head modules;
crash> list modules
ffffffff812cd420
ffffffff88302c08
...
ffffffff88017d88
ffffffff8800 9a88
crash> list modules | wc -l
38
crash>
```

modules リストには 38 のエントリがつながっていることがわかります。

次は構造体の中に list_head 構造体のメンバがある例です。このような場合は -o オプションでオフセットを指定すると同じようにリストをたどります。

```
crash> whatis ip_packet_type
struct packet_type ip_packet_type;
crash> struct -o packet_type
struct packet_type {
   [0x0] __be16 type;
   [0x8] struct net_device *dev;
  [0x10] int (*func)(struct sk_buff *, struct net_device *, struct packet_type *, struct
 net_device *);
         struct sk_buff *(*gso_segment)(struct sk_buff *, int);
  [0x20] int (*gso_send_check)(struct sk_buff *);
  [0x28] void *af_packet_priv;
  [0x30] struct list_head list;           /* list_head がある */
}
SIZE: 0x40
crash>
```

list_head 構造体のオフセットが 0x30 とわかりましたので、-o オプションに指定します。

```
crash> list -o 0x30 ip_packet_type
ffffffff813288a0
ffffffff814188a0
ffffffff814188d0
ffffffff81327730
```

ip_packet_type の list メンバに 4 つのエントリがつながっていることになります。

-s オプションを付けると、そのリストエントリのメンバを同時に表示します。次の例はリストをたどり、さらに各エントリの func メンバを表示しています。

```
crash> list -o 0x30 ip_packet_type -s packet_type.func
ffffffff813288a0
  func = 0xffffffff810353e5 <ip_rcv>,
ffffffff814188a0
  func = 0xffffffff813052f0 <llc_tr_packet_type+48>,
ffffffff814188d0
  func = 0xffffffff813052b0 <llc_packet_type+48>,
ffffffff81327730
  func = 0xffffffff812275f3 <arp_netdev_event>,
```

mod コマンド

mod コマンドはモジュールの情報とシンボル情報やデバッグ情報をロードするコマンドです。詳細は**「アセンブリ言語からソースコードの対応を調べる」[HACK #14]**、**「カーネルパニック（NULL ポインタ参照編）」[HACK #33]** などを参照してください。

net コマンド

net コマンドはネットワークデバイスのリストを表示します。net_device 構造体のアドレスを表示するので、それからさらに詳細を調べることができます。

```
crash> net
   NET_DEVICE     NAME   IP ADDRESS(ES)
ffffffff8030f680  lo     127.0.0.1
ffff81007ea24000  eth2   192.168.0.155
ffff81007f7b8000  eth3   192.168.0.156
ffff81007ebbd000  eth0   172.16.0.153
ffff81007e1ff000  eth1
ffff81007d50a000  sit0
crash> struct net_device ffff81007ea24000
struct net_device {
  name = "eth2\00045090668\000\000",
  name_hlist = {
    next = 0x0,
    pprev = 0xffffffff804bae90
  },
...
```

ps コマンド

ps コマンドはプロセス情報を表示します。

-a オプションはコマンドライン引数と環境変数を表示します。

```
crash> ps -a syslogd
PID: 2157   TASK: ffff81007e095040  CPU: 0   COMMAND: "syslogd"
ARG: syslogd -m 0
ENV: CONSOLE=/dev/console
     SELINUX_INIT=YES
     TERM=linux
     INIT_VERSION=sysvinit-2.86
     PATH=/sbin:/usr/sbin:/bin:/usr/bin
     runlevel=3
     RUNLEVEL=3
     PWD=/
     LANG=en_US.UTF-8
     previous=N
     PREVLEVEL=N
     SHLVL=3
     HOME=/
     _=/sbin/syslogd
```

-t オプションはプロセスの動作時間、開始時間、ユーザ空間、カーネル空間での実行時間を表示します。詳細は「**リアルタイムプロセスのストール**」**[HACK #40]** を参照してください。

rd コマンド

rd コマンドはメモリを直接読み出すコマンドです。使用例は本 Hack の bt、wr コマンドを参照してください。

runq コマンド

runq コマンドはスケジューラのランキューを表示します。

sig コマンド

sig コマンドはプロセスのシグナルハンドラを表示します。またペンディングのシグナル情報を表示します。-l オプションは kill -l と同等で、定義されているシグナル番号を表示します。

struct コマンド

struct コマンドは構造体の定義と実際のアドレスから構造体に合わせてデータを表示します。

```
crash> struct timespec xtime
struct timespec {
  tv_sec = 0x492ae39c,
  tv_nsec = 0x25218473
}
```

他の例は **[HACK #14]** を参照してください。

swap コマンド

swap コマンドはスワップデバイスごとにサイズなどの情報を出力します。swapon -s とほぼ同じ内容になります。

sym コマンド

sym コマンドはシンボル解決をするコマンドです。sym -l は cat System.map と同じです。

sys コマンド

sys コマンドはシステムの情報を表示します。時間や CPU のロードアベレージ、カーネルパニックの原因を示すメッセージなどが表示されます。crash を起動して最初に表示される情報と同じです。

カーネルコンフィグで CONFIG_IKCONFIG が有効の場合は sys config を実行することでカーネルコンフィグの一覧が表示できます。これは zcat /proc/config.gz と内容は同じです。

```
crash> sys config
#
# Automatically generated make config: don't edit
# Linux kernel version: 2.6.18
# Tue Dec 16 22:33:31 2008
#
CONFIG_X86_64=y
CONFIG_64BIT=y
CONFIG_X86=y
CONFIG_LOCKDEP_SUPPORT=y
CONFIG_STACKTRACE_SUPPORT=y
```

```
CONFIG_SEMAPHORE_SLEEPERS=y
CONFIG_MMU=y
…
```

sys -panic で意図的にカーネルパニックさせることができます。動作は echo c › /proc/sysrq-trigger と同じです。

task コマンド

task コマンドは task_struct 構造体を表示するコマンドです。詳細は **[HACK #40]** などを参照してください。

timer コマンド

timer コマンドはタイマキューのエントリを表示します。

whatis コマンド

whatis コマンドはシンボルなどの構造体の定義を表示します。以下はグローバル変数 modules の表示例です。list_head 構造体の変数ということがわかります。

```
crash› whatis modules
struct list_head modules;
```

wr コマンド

wr コマンドはメモリの内容を書き換えるコマンドです。下の例は crash のライブシステムで時間を表す jiffies 変数を書き換えてます。書き換えたあとには起動してからの時間を表す UPTIME が 3 日以上増えています。

```
crash› sys
      KERNEL: /boot/vmlinux-2.6.18
    DUMPFILE: /dev/mem
        CPUS: 2
        DATE: Fri Dec 19 20:26:07 2008
      UPTIME: 00:00:45
…
crash› rd jiffies_64
ffffffff81457200:  00000000fffc6751                Qg……
crash› wr jiffies_64 10ffc7651
crash› rd jiffies_64
```

```
ffffffff81457200:  000000010ffc88c0                    ........
crash> sys
      KERNEL: /boot/vmlinux-2.6.18
    DUMPFILE: /dev/mem
        CPUS: 2
        DATE: Fri Dec 19 20:26:56 2008          /* コマンドは 50 秒後なのに */
      UPTIME: 3 days, 02:35:10                  /* UPTIME は 3 日以上増えている */
...
```

crash の起動オプション

crash 起動時の便利なオプションを紹介します。

-i オプション

-i に crash の入力コマンドを記述したファイルを指定すると、自動で crash に入力ができます。このオプションで crash を使った自動化が可能です。以下は help コマンドを実行して、exit で crash を終了する例になります。

```
[bash]# cat crash_cmd.txt
help
exit
[bash]# crash -s -i crash_cmd.txt
*             files         mod           runq          union
alias         foreach       mount         search        vm
ascii         fuser         net           set           vtop
bt            gdb           p             sig           waitq
btop          help          ps            struct        whatis
dev           irq           pte           swap          wr
dis           kmem          ptob          sym           q
eval          list          ptov          sys
exit          log           rd            task
extend        mach          repeat        timer

crash version: 4.0-7.4   gdb version: 6.1
For help on any command above, enter "help <command>".
For help on input options, enter "help input".
For help on output options, enter "help output".

[bash]#                  /* ヘルプを表示したあと bash に戻る */
```

-s オプション

上のように -s を付けるとサイレントモードになり、crash 起動時の余計な表示がなくなります。

crash の初期化ファイル

.crashrc ファイルにコマンドを書いておくと crash コマンド起動時にそのコマンドを実行します。.crashrc はホームディレクトリ、またはカレントディレクトリに置きます。デフォルトの設定を書いておくと便利です。set コマンドや alias コマンドを書くとよいでしょう。

まとめ

crash のコマンドとオプションを紹介しました。コマンドを使いこなして効率よく解析しましょう。

参考文献

White Paper: Red Hat Crash Utility

http://people.redhat.com/anderson/crash_whitepaper/

—— Naohiro Ooiwa

HACK #22 IPMI watchdog timer により、フリーズ時にクラッシュダンプを取得する

ミドルクラス以上のサーバ機に搭載されている IPMI watchdog timer の設定方法を説明し、フリーズした場合のデバッグに備えられるようにします。

IPMI watchdog timer とは

IPMI watchdog timer は、Intelligent Platform Management Interface という規格に従ったウォッチドックタイマです。IPMI は、Intel 社などコンピュータ関連ベンダ数社により作成された規格で、コンピュータ各部の温度、電圧、ファンなどの状態取得や、電源などを制御するためのインタフェースを規定しています。この規格には、本 Hack で説明する watchdog timer（以降、WDT と表記）も含まれています。IPMI WDT は、Core 2 や Xeon などのメイン CPU とは独立しており、専用のハードウェアを用いて実装されます。そのため、システムがフリーズした際、ほぼ確実にリセット等の処理が実行されます。したがって、ウォッチドックタイマとしては、特別なハードウェアが不要な softdog や Intel などの一部のプロセッサで利用可能な NMI watchdog（**[HACK #23]** 参照）より、高い信頼性を持ちます。ただし、この機能は、すべての PC やサーバに搭載されているわけではありません。一般的には、ミドルクラス以上のサーバ機に搭載されていることが多い

ようです。

このように IPMI WDT は、本来、システムの可用性を高めるための機能ですが、デバッグとも深い関わりがあります。それは、WDT によってリセットが行われたということは、システムがフリーズしたということを意味しているからです。フリーズの原因は、ソフトウェアのバグの場合もあれば、ハードウェア故障に起因する場合もありますが、いずれにせよ、その原因を探るためには、何らかの情報が必要です。Linux では、IPMI WDT の機能を利用して、システムがフリーズしていても、リセットが行われる前に、クラッシュダンプを取得できる仕組みがあります。もちろん、それは、カーネルのダンプ取得部が動作するようなフリーズの場合に限られますが、そのような場合には、非常に有用です。**「カーネルのストール（無限ループ編）」[HACK #36]**「**リアルタイムプロセスのストール」[HACK #40]** では、実際に IPMI WDT により取得されたクラッシュダンプを使って、デバッグを行う実例を紹介します。

Linux における IPMI の使用

Linux で、IPMI を使用するためには、カーネルコンフィグによる IPMI の有効化と、ipmitools、FreeIPMI などの IPMI 用のユーザランドプログラムが必要です。カーネルコンフィグは、RHEL など、大半のディストリビューションでは、デフォルトで有効にされており、カーネルモジュールとして提供されています。もし、使用しているディストリビューションのデフォルトがオフの場合や、モジュールから組み込みに変更する場合、make menuconfig で、Device Drivers -> Character devices -> IPMI top-level message handler と、その奥の階層にある IPMI Watchdog Timer 項目を設定します。

ipmi_watchdog モジュール

ここでは、IPMI を使用するためのカーネル機能が、モジュールとして提供されている場合を説明します。このモジュールは ipmi_watchdog.ko という名称です。このモジュールの主なパラメータを**表 3-7** に、各パラメータの意味を**表 3-8** から**表 3-10** にそれぞれ示します。また、timeout、pretimeout、action、preaction の関係を図に示します。

表 3-7　ipmi_watchdog モジュールの主なパラメータ

パラメータ	説明
timeout	タイムアウトまでの時間（秒）
action	タイムアウト時の動作（reset、none、power_cycle、power_off）
pretimeout	タイムアウト動作実行までの時間とプリタイムアウト動作実行まで時間の差（秒）
preaction	プリタイムアウト時の動作（pre_none、pre_smi、pre_nmi、pre_int）
preop	プリタイムアウト時のドライバの動作（preop_none、preop_panic）

表 3-8 action の各パラメータ指定時の動作

パラメータ	説明
none	何もしません
reset	システムをリセットします
power_cycle	電源をいったんオフにした後、再びオンにします
power_off	電源をオフにします

表 3-9 preaction の各パラメータ指定時の動作

パラメータ	説明
pre_none	何もしません
pre_smi	IPMI ドライバへ情報を通知します
pre_int	IPMI ドライバへ割り込みを使って情報を通知します
pre_nmi	NMI 割り込みを発生させます

表 3-10 preop の各パラメータ指定時の動作

パラメータ	説明
preop_none	何もしません
preop_panic	カーネルパニックを発生させます

ダンプを取得するためには、pretimeout に timeout より小さい値を設定し、preaction に pre_nmi または、pre_int を設定します。例えば、timeout に 90、pretimeout を 30 に設定した場合、最後の watchdog timer の更新から 60 秒間、再度の更新が行われなかった（システムがフリーズした）時点で、preaction に指定された動作が実行されます。pre_nmi を設定した場合、pretimeout で指定した時間以上のフリーズが発生すると、M/B 上の NMI 信号がアサートされます。NMI がアサートされると、通常はカーネルの NMI ハンドラが動

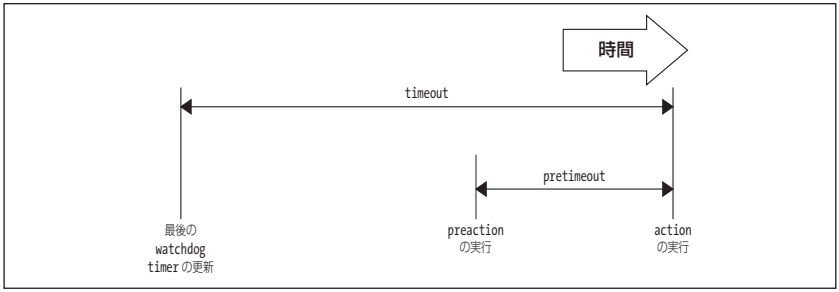

図 3-4 モジュールパラメータと実行されるアクションの関係

作するので、パニックが発生します。kdump（**[HACK #20]** 参照）等が設定されていれば、これによりダンプの取得ができます。また、preaction に pre_int を指定し、preop に preop_panic を指定することでも、ダンプの取得ができます。この場合、プリタイムアウト時間を超過した後、ipmi のドライバにそのことが通知されます。すると、ドライバは、preop に指定された動作を実行します。preop_panic は、パニックを引き起こします。

これらのパラメータは、もちろん、コマンドラインからロードする時に直接指定することもできますし、/etc/modprobe.conf に記述しても構いません。RHEL5 では、/etc/sysconfig/ipmi に次のような記述をし、ipmi サービスをオンにしておくと、起動時に自動的にロードされます。

```
IPMI_WATCHDOG=yes
IPMI_WATCHDOG_OPTIONS="timeout=90 action=reset pretimeout=30 preaction=pre_int preop=preop_panic"
```

なお、パラメータを設定するにあたり留意しておくことが 2 つあります。1 つ目は、もし、カーネルがクラッシュダンプを実行する途中でフリーズした場合、pretimeout の動作は、当然、実行されず、timeout で設定された時間が超過した後、action で指定された動作（H/W 的なリセット等）が実行されることです。2 つ目は、pretimeout が実行されても、その後、フリーズが続くと、timeout 値に設定した時間が経過したところで、action が実行されることです。kdump を使用する場合は、セカンドカーネルが動作しているので、問題ありませんが、preaction でのカーネルパニック後、diskdump など、フリーズしたファーストカーネルを使用してクラッシュダンプを取得する場合、問題が生じることがあります。例えば、搭載メモリ量が多く、timeout までにダンプが完了しない場合、クラッシュダンプの取得途中でも、リセット等の動作が実行されます。

/dev/watchdog インタフェース

ipmi_watchdog.ko をロードすると、/dev/watchdog を通じて、IPMI WDT を制御できるようになります。/dev/watchdog は、Linux の標準的な watchdog の制御機構を提供し、IPMI watchdog 以外にも、softdog やベンダ固有の H/W WDT の制御のためにも使用されます。だたし、このインタフェースは排他的であり、例えば、softdog と IPMI WDT を同時に使うことはできません。

/dev/watchdog を制御するプログラムは複数あり、それらは ipmitools などに含まれています。また、カーネルソースツリーの Documentation/watchdog/src/watchdog-simple.c もその 1 つです。下記は、それをさらに簡略化したソースです。

```
$ cat watchdog-very-simple.c
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(void)
{
        int fd = open("/dev/watchdog", O_WRONLY);
        int ret = 0;
        if (fd == -1) {
                perror("watchdog");
                exit(EXIT_FAILURE);
        }
        while (1) {
                if (write(fd, "\0", 1) != 1) {
                        ret = -1;
                        break;
                }
                sleep(10);
        }
        close(fd);
        return ret;
}
```

クラッシュダンプの出力

ここでは、上記の watchdog-very-simple を用いて、フリーズ時にクラッシュダンプが出力されることを確認します。ただし、実際にフリーズさせることは難しいので、以下のようにします。

```
# modprobe ipmi_si
# modprobe impi_watchdog timeout=90 action=reset pretimeout=30 preaction=pre_int preop=preop_panic
# watchdog-very-simple
Ctrl-Z
#
```

これは、watchdog-very-simple プロセスを停止することで、擬似的なフリーズを発生させます。この操作の後、カーネルモジュールの pre_timeout に設定にしたがって、クラッシュダンプが発生します。

なお、最初の ipmi_si は、M/B 上に搭載されている IPMI 用の H/W と通信するためのモジュールです。IPMI 用の H/W が搭載されていない場合、このモジュールのロードが

失敗します。

参考文献

Linux カーネルの Documentation/IPMI.txt

まとめ

IPMI watchdog timer を利用して、フリーズ時にクラッシュダンプを取得する方法を説明しました。

—— Kazuhiro Yamato

HACK #23 NMI watchdog により、フリーズ時にクラッシュダンプを取得する

x86_64 や i386 アーキテクチャの NMI watchdog 機能を使って、システムがフリーズした場合にクラッシュダンプを取得する方法を説明します。

NMI watchdog とは

NMI は、Non Maskable Interrupt の略で、禁止できない割り込みを意味します。この割り込みは、本来、メモリのパリティエラーなどシステムの致命的エラーを CPU に伝えるために使用されます。しかし、最近の APIC には、定期的にこの割り込みを発生させる機能があります。NMI watchdog は、この機能を利用したウォッチドッグタイマです。NMI watchdog により、Linux カーネルは、システムがフリーズしていることを検知し、パニックの発生や、クラッシュダンプ取得を行うことができます。

カーネルが NMI watchdog を使ってフリーズを検知する仕組みは次のとおりです。通常、タイマ割り込みは、1 秒間にカーネルコンフィグで設定した回数（100 から 1000 回）発生します。しかし、例えば、割り込みを禁止したまま、無限ループやデッドロックに陥ると、タイマ割り込み処理が実行されなくなります。一方、NMI は、そのような状態でも発生し、CPU は NMI ハンドラを実行します。NMI ハンドラは、タイマ割り込みが実行されているかを監視し、一定時間以上（ディストリビューションにより若干異なりますが 5 ～ 30 秒程度）、タイマ割り込みが実行されていないとフリーズとみなします。

NMI watchdog が使用できるかのチェック

最近では、多くのマシンでこの機能が利用できますが、実際に利用できるか否かは次のように調べます。

```
# cat /proc/interrupts
           CPU0       CPU1
…
NMI:  598499293  598499235
…

(2-3秒して)
# cat /proc/interrupts
           CPU0       CPU1
…
NMI:  598502039  598501981
…
```

このとき、NMIが増加していれば、NMI watchdogが使用できます。NMI watchdogが利用できない場合、NMIは致命的なエラーの際に発生する割り込みなので、NMIの数はほぼ0です。

NMI watachdogタイムアウト時にクラッシュダンプを取得する

上記のチェックで、NMI watchdogが使用できるとわかった場合、カーネルオプションに以下を指定します。

```
nmi_watchdog=panic,N   (Nは、1か2)
```

IO-APICを持つ装置では、Nを1（I/O-APIC mode）にし、IO-APICを持たないUP（Uni Processor）の装置ではNを2（local APIC mode）にします。この設定により、NMI watchdogがタイムアウトした時にはカーネルパニックが発生するので、あとはkdump（**[HACK #20]**）やdiskdump（**[HACK #19]**）のクラッシュダンプ取得の設定をします。なお、カーネルのバージョンよって、上記のnmi_watchdogカーネルオプションが設定されていなくても、デフォルトでパニックする場合もあります。また、kdumpに関しては、パニックが発生しなくても、irq実行中であるとか、カレントタスクがアイドルまたはinitであるなどのいくつか条件にあてはまれば、kexecによりセカンドカーネルが起動され、クラッシュダンプが実行されます。いずれにせよ、確実にnmi_watchdogのタイムアウトでダンプを取得するなら、上記の設定をしておきます。

なお、NMI watchdogがタイムアウトした場合のカーネルメッセージの例は**「カーネルのストール（スピンロック編その2）」[HACK #38]**を参照してください。

まとめ

NMI watchdog を利用して、フリーズ時にクラッシュダンプを取得する方法を説明しました。

—— Kazuhiro Yamato

HACK #24 カーネル特有のアセンブリ命令（その 1）

ユーザ空間ではあまり見られないアセンブリ命令を紹介します。

カーネルのダンプを解析する場合は、カーネルのソースコードと crash コマンドの dis などで表示されるアセンブリ言語を照らし合わせて解析することになります。

本 Hack ではユーザ空間で見られませんが、カーネルではよく見るアセンブリ命令を説明します。objdump の出力は Linux2.6.19 カーネルを例にしています。

BUG：ud2 命令

カーネルのアセンブリ言語を見ると ud2 命令がよく現れます。以下は free_buffer_head() です。

```
# objdump -d vmlinux-2.6.19
…
c0184731 <free_buffer_head>:
c0184731:   89 c2                mov    %eax,%edx
c0184733:   8d 40 28             lea    0x28(%eax),%eax
c0184736:   39 42 28             cmp    %eax,0x28(%edx)
c0184739:   74 08                je     c0184743 <free_buffer_head+0x12>
c018473b:   0f 0b                ud2a ——————————❶
c018473d:   8b 0b                mov    (%ebx),%ecx
c018473f:   dc 34 33             fdivl  (%ebx,%esi,1)
c0184742:   c0 a1 a0 35 4a c0 e8 shlb   $0xe8,0xc04a35a0(%ecx)
c0184749:   10 11                adc    %dl,(%ecx)
…
```

❶に ud2a とあります。これはカーネルのソースコードを見るとすぐにわかります。❶はソースコードの①になります。

[fs/buffer.c]

```
void free_buffer_head(struct buffer_head *bh)
{
        BUG_ON(!list_empty(&bh->b_assoc_buffers));  ——①
```

```
        kmem_cache_free(bh_cachep, bh);
        get_cpu_var(bh_accounting).nr--;
        recalc_bh_state();
        put_cpu_var(bh_accounting);
}
```

[include/asm-generic/bug.h]

```
#ifndef HAVE_ARCH_BUG_ON
#define BUG_ON(condition) do { if (unlikely((condition)!=0)) BUG(); }
while(0)
#endif
```

[include/asm-i386/bug.h]

```
#define BUG() __asm__ __volatile__("ud2\n") ————①
```

ud2 命令があればすぐに BUG_ON()、または BUG() であるということがわかります。

この命令を Intel のマニュアルは**図 3-5** のようになっています。

「無効オペコード例外」とは CPU が発生させる例外（割り込み番号は 6）で、カーネルが ud2 命令を実行させることで、CPU からこの例外を受け取ります。カーネルはこの例外を受け取ると handle_BUG() で BUG() をコールします。

割り込み禁止 / 許可：sti、cli 命令

カーネルのアセンブリコードを見ると sti、cli 命令もよく現れます。以下は on_each_cpu() です。

```
# objdump -d vmlinux-2.6.19
…
c0129e5a <on_each_cpu>:
…
c0129e64:   8b 44 24 14         mov    0x14(%esp),%eax
c0129e68:   89 04 24            mov    %eax,(%esp)
c0129e6b:   89 f8               mov    %edi,%eax
```

UD2-Undefined Instruction

オペコード	命令	説明
OF OB	UD2	無効オペコード例外を発生させる。

図 3-5　Intel のマニュアルの説明

```
c0129e6d:   e8 70 cc fe ff      call   c0116ae2 <smp_call_function>
c0129e72:   89 c3               mov    %eax,%ebx
c0129e74:   fa                  cli  ―――――――――――――――❷
c0129e75:   89 f0               mov    %esi,%eax
c0129e77:   ff d7               call   *%edi
c0129e79:   fb                  sti  ―――――――――――――――❸
c0129e7a:   5a                  pop    %edx
c0129e7b:   89 d8               mov    %ebx,%eax
c0129e7d:   5b                  pop    %ebx
…
```

カーネルのソースコードでは以下のようになります。

[kernel/softirq.c]

```
int on_each_cpu(void (*func) (void *info), void *info, int retry, int wait)
{
        int ret = 0;

        preempt_disable();
        ret = smp_call_function(func, info, retry, wait);
        local_irq_disable(); ―――――――――――――――――②
        func(info);
        local_irq_enable(); ―――――――――――――――――③
        preempt_enable();
        return ret;
}
```

[include/linux/irqflags.h]

```
#ifdef CONFIG_TRACE_IRQFLAGS
…
#else
# define trace_hardirqs_on()            do { } while (0)
# define trace_hardirqs_off()           do { } while (0)
…
#endif

#define local_irq_enable() \
        do { trace_hardirqs_on(); raw_local_irq_enable(); } while (0)
#define local_irq_disable() \
        do { raw_local_irq_disable(); trace_hardirqs_off(); } while (0)
…
```

```
[include/asm-i386/irqflags.h]
  static inline void raw_local_irq_disable(void)
  {
        __asm__ __volatile__("cli" : : : "memory"); ———②
  }

  static inline void raw_local_irq_enable(void)
  {
       __asm__ __volatile__("sti" : : : "memory"); ———③
  }
```

cli 命令は local_irq_disable() で割り込みを禁止します。また sti 命令は local_irq_enable() で割り込みを許可します。

まとめ

カーネルで多く見られるアセンブリ命令 ud2、sti、cli を紹介しました。

参考文献

- IA-32 インテル ® アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル、中巻 A：命令セット・リファレンス A-M
  http://download.intel.com/jp/developer/jpdoc/IA32_Arh_Dev_Man_Vol2A_i.pdf
- IA-32 インテル ® アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル、中巻 B：命令セット・リファレンス N-Z
  http://download.intel.com/jp/developer/jpdoc/IA32_Arh_Dev_Man_Vol2B_i.pdf

—— Naohiro Ooiwa

HACK #25 カーネル特有のアセンブリ命令（その 2）

カーネルにて頻繁に現れる処理である current マクロについて、x86 アーキテクチャにおけるアセンブラレベルでの説明をします。

current とは

カーネルの処理において、現在実行中であるプロセスの task_struct 構造体を得る処理がよくあります。カーネルのソースでは、その処理の記述には current マクロが使用されます。カーネルソース上 current が出てきたら、それは現在実行中のプロセスにおける task_struct

構造体の取得を意味します。task_struct 構造体とは、カーネル内部でプロセスの状態を管理するためのデータ構造です。

カーネルのクラッシュダンプからのバグ解析などにおいて、アセンブラを読まなければならない場面が多々あります。current は頻繁に参照される要素なので、カーネルソースとアセンブラを対応づけるときに、この current の処理を目印にすると便利です。

task_struct 構造体と thread_info 構造体

カーネル 2.4 と違い、カーネル 2.6 において、プロセスを管理する task_struct 構造体は SLAB にあります。

task_struct 構造体は thread_info へのポインタを、thread_info 構造体は task_struct 構造体へのポインタを保持しています。

thread_info 構造体が置かれている領域はそのプロセスのカーネルにおけるスタック領域の先頭です。なお、カーネルスタックは 2 ページ、8KB 確保され、最後から使われます。4KB スタックの場合は 1 ページとなります。

カーネル 2.4 においては、thread_info 構造体は存在せず、task_struct 構造体がカーネルスタックの先頭に置かれています。

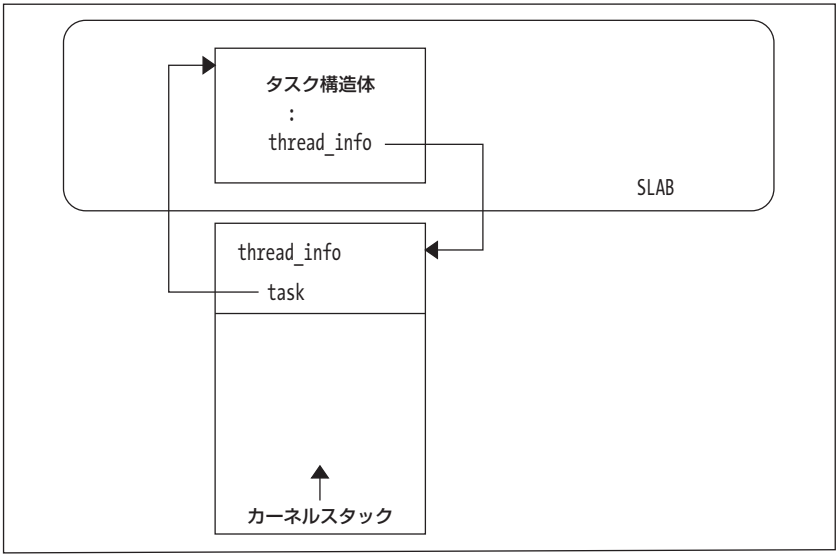

図 3-6　task_struct 構造体と thread_info 構造体

current 取得処理の詳細

それでは、current 取得処理のアセンブラでの見え方について説明します。

current 取得処理はアーキテクチャ、カーネルのバージョンによって違いがあります。ここでは、x86 アーキテクチャ、32 ビット（i386）と 64 ビット（x86_64）について説明します。

i386: 32 ビット

2.6.19 以前のカーネルでは下記のようになります。

```
c0101628:       be 00 e0 ff ff          mov    $0xffffe000,%esi
c010162d:       21 e6                   and    %esp,%esi
c010162f:       8b 1e                   mov    (%esi),%ebx
```

現在のスタックポインタ esp から thread_info 構造体へのポインタを取得し、thread_info 構造体から task_struct 構造体へのポインタを取得しています。この処理のポイントは 0xffffe000 とスタックポインタ esp の論理積（AND）です。スタックポインタの下位 13 ビットをクリアすることで、8KB のカーネルスタックとして確保された領域の先頭アドレスが取得できます。したがって、カーネルスタックに対する論理積（AND）が current に対応する箇所を見つけるキーとなります。

2.6.20 からは percpu 領域に current へのポインタを保持しています。その際セグメントセレクタ fs を percpu 領域へのアクセスに使用します。2.6.20 以降では下記のようになります。

```
c1002173:       64 8b 35 00 f0 69 c1    mov    %fs:0xc169f000,%esi
```

セグメントセレクタ fs とオフセット 0xc169f000 でアクセスしています。このように、2.6.20 以降は current へのポインタが percpu 領域にあるため、セグメントセレクタ fs を介した percpu 領域へのアクセスから current に対応する箇所を見つけるためのキーとなります。

x86_64: 64 ビット

x86_64 では PDA（per processor data structure）領域に current へのポインタが保持されており、gs セグメントレジスタを利用した PDA へのアクセスで current が取得されます。

```
ffffffff8010cfdf:       65 48 8b 3c 25 00 00    mov    %gs:0x0,%rdi
ffffffff8010cfe6:       00 00
```

セグメントセレクタ gs とオフセット 0 でアクセスしています。現在のプロセスの task_

struct 構造体へのポインタは PDA 領域の先頭にあるため、セグメントセレクタ gs とオフセット 0 によるアクセスがそのまま current に対応する処理となります。

開発版のカーネル（2.6.30 マージ予定）では x86_64 専用で存在した PDA 領域が削除され、percpu に統合される予定です。

```
ffffffff81009186:       65 48 8b 04 25 00 b0    mov    %gs:0xb000,%rax
ffffffff8100918d:       00 00
```

このようにセグメントセレクタの違いはありますが、2.6.20 以降の i386 における処理と似たものになります。

まとめ

本 Hack では Linux カーネルでよく使われる current が x86 アーキテクチャにおけるアセンブラでどうなっているかを説明しました。current は頻繁に参照される要素なので、カーネルソースを追いかけるときに有効です。

参考文献

- Intel® 64 and IA-32Architectures Software Developer's Manuals
  http://www.intel.com/products/processor/manuals/index.htm

—— Hiroshi Shimamoto

4章
実践アプリケーションデバッグ
Hack #26-32

この章では、ユーザアプリケーションの実践的なデバッグ方法について記しています。スタックオーバーフローによるセグメンテーションフォルト（SIGSEGV）、バックトレースが正しく表示されない、配列の不正アクセスによるスタック破壊、ウォッチポイントを活用した不正メモリアクセスの検知、malloc()/free() での障害、アプリケーションのストールなどさまざまな事例によるデバッグ方法を記しています。

HACK #26 SIGSEGV でアプリケーションが異常終了した
スタックオーバーフローによるセグメンテーションフォルトのデバッグ

アプリケーションプログラムが不正なメモリアクセスなどをした場合、SIGSEGV という例外を発生し異常終了します。SIGSEGV が発生する場合は、(1) NULL ポインタによるアクセス、(2) ポインタ破壊などによる不正アドレスへのアクセス、(3) スタックオーバーフローなどにより、確保したアドレス領域を越えてのアクセス、などがあります。

ここでは、スタックオーバーフローにより SIGSEGV が発生した場合のデバッグ方法について解説します。

以下にセグメンテーションフォルトを発生させる例を示します。

```
$ ruby1.8 -e 'eval("1+" * 100000 + "1")'
Segmentation fault
```

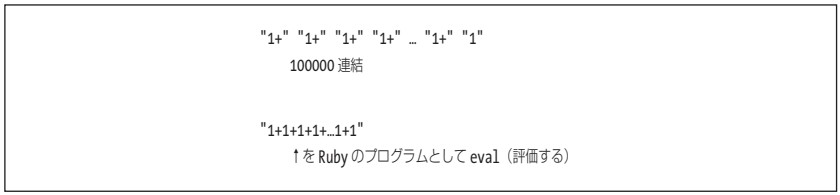

図 4-1　eval("1+" * 100000 + "1") プログラム

eval("1+" * 100000 + "1") というのは、"1+" という文字列を 100000 個連結したものに、"1" という文字列を連結した文字列を Ruby のプログラムとして評価（eval）するプログラムです。

コアファイルを生成するように設定します。

```
$ ulimit -c unlimited
$ ruby1.8 -e 'eval("1+" * 100000 + "1")'
Segmentation fault (core dumped)
$ ls core
core
```

アプリケーションプログラムがコアファイルを生成したので、デバッガで分析してみましょう。

```
$ gdb ruby1.8 core
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(no debugging symbols found)
中略
Loaded symbols for /lib/ld-linux.so.2
(no debugging symbols found)
Core was generated by `ruby1.8 -e eval("1+" * 100000 + "1")'.
Program terminated with signal 11, Segmentation fault.
[New process 24488]
#0  0xb7e22cb7 in ?? () from /usr/lib/libruby1.8.so.1.8
```

システムにインストールされているアプリケーション（この場合は ruby1.8）にはデバッグ情報が付加されていないので、シンボル情報が表示できません。しかし、スタックフレームの情報などから、ある程度原因を推定できる場合があります。

bt（backtrace）コマンドでスタックフレームを取得してみます。そうすると、大量にスタックフレームが表示されました。これは再帰的に関数が呼ばれているということを示しています。そこで最初のいくつかだけを表示することにします。この場合 10 個取得することにします。

bt ‹数字› という形式です。

```
(gdb) bt 10
#0  0xb7e22cb7 in ?? () from /usr/lib/libruby1.8.so.1.8
#1  0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#2  0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#3  0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#4  0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#5  0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#6  0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#7  0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#8  0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#9  0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
```

スタックフレームを眺めてみると、0xb7e22f3a というアドレスから何度も呼ばれていることがわかります。これは再帰的に関数が呼ばれスタックオーバーフローでアプリケーションが異常終了したことが疑われます。

ソースコードレベルのデバッグ

それでは gdb でソースコードを追跡してみましょう。あらかじめ、gcc の -g オプション付きでビルドしたアプリケーション（ruby）を gdb で起動します。

```
(gdb) run -e 'eval("1+" * 100000 + "1")'
Starting program: /home/hyoshiok/work/ruby_trunk/ruby/ruby -e 'eval("1+" * 100000 + "1")'
[Thread debugging using libthread_db enabled]
[New Thread 0xb7d3d6b0 (LWP 24646)]
[New Thread 0xb7f24b90 (LWP 24649)]

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0xb7d3d6b0 (LWP 24646)]
iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b138, node=0x931dd74, poped=0) at compile.c:2883
```

アプリケーションが SIGSEGV を引き起こしました。gdb の場合、シグナルを受け取るとあらかじめ定義された動作を起こします。SIGSEGV の場合は当該の箇所で自動的に停止してくれます。

info signal で gdb が処理するシグナルの一覧が表示できます。

ソースコードは下記のところです。emacs で gdb を起動すれば自動的に表示してくれますので便利ですね。

```
/**
  compile each node

  self:  InstructionSequence
  node:  Ruby compiled node
  poped: This node will be poped
 */
static int
iseq_compile_each(rb_iseq_t *iseq, LINK_ANCHOR *ret, NODE * node, int poped)
{   /* ここで停止する */
    enum node_type type;

    if (node == 0) {
         if (!poped) {
             debugs("node: NODE_NIL(implicit)\n");
             ADD_INSN(ret, iseq->compile_data->last_line, putnil);
         }
         return COMPILE_OK;
    }
```

スタックフレームを bt コマンドで取ります。せいぜい 5 つまで取れば十分でしょう。

```
(gdb) bt 5
#0  iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b138, node=0x931dd74, poped=0) at compile.c:2883
#1  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b278, node=0x931dd38, poped=0) at
compile.c:3954
#2  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b3b8, node=0x931dcfc, poped=0) at
compile.c:3954
#3  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b4f8, node=0x931dcc0, poped=0) at
compile.c:3954
#4  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b638, node=0x931dc84, poped=0) at
compile.c:3954
(More stack frames follow...)
```

そうすると、0x0811154d というアドレスから何度も呼ばれていることがわかります。スタックフレームを 1 つ上にのぼってみましょう。up コマンドを利用します。

```
(gdb) up
#1  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b278, node=0x931dd38, poped=0) at
compile.c:3954
```

当該アドレスのソースコードは以下の COMPILE() のところです。

```
                    else {
                        ADD_LABEL(ret, label);
                    }
                    break;
                }
            }
        }
#endif
        /* reciever */
        if (type == NODE_CALL) {
            COMPILE(recv, "recv", node->nd_recv); /* ここから呼んでいる */
        }
        else if (type == NODE_FCALL || type == NODE_VCALL) {
            ADD_CALL_RECEIVER(recv, nd_line(node));
        }

        /* args */
        if (nd_type(node) != NODE_VCALL) {
            argc = setup_args(iseq, args, node->nd_args, &flag);
        }
```

ソースコードを見ると、COMPILE は下記のようにマクロの定義になっていて iseq_compile_each() 関数を再帰的に呼んでいることがわかります。

```
/* compile node */
#define COMPILE(anchor, desc, node) \
  (debug_compile("== " desc "\n", \
                 iseq_compile_each(iseq, anchor, node, 0)))
```

ソースコードを分析した結果、何度も再帰的に関数を呼んだためスタックオーバーフローを引き起こしたということがわかります。

スタックオーバーフローで SIGSEGV への対応

一般的に言ってシグナルを捕獲したら、そのためのシグナルハンドラを用意して、何がしかの作業をすればいいわけです。しかしスタックオーバーフローで SIGSEGV を発生させた場合は、スタック領域があふれ不正アクセスになったため、シグナルハンドラを起動するスタックすら確保できないので、そのままでは対処できません。そのため、スタックオーバーフローを捕捉するために代替シグナルスタックを設定する必要があります。それには

sigaltstack(2) を使います。

man page に掲載されている例は以下のとおりです。

```
stack_t ss;

ss.ss_sp = malloc(SIGSTKSZ);
if (ss.ss_sp == NULL)
    /* ハンドルエラー */;
ss.ss_size = SIGSTKSZ;
ss.ss_flags = 0;
if (sigaltstack(&ss, NULL) == -1)
    /* ハンドルエラー */;
```

さて、それを参考に以下のようなパッチを作成してみました。

```
$ svn diff signal.c
Index: signal.c
===================================================================
--- signal.c          (リビジョン 20086)
+++ signal.c          (作業コピー)
@@ -47,6 +47,10 @@
 # define NSIG (_SIGMAX + 1)      /* For QNX */
 #endif

+#ifdef SIGSEGV
+static int is_altstack_defined = 0;
+#endif
+
 static const struct signals {
     const char *signm;
     int  signo;
@@ -410,6 +414,28 @@
 typedef RETSIGTYPE (*sighandler_t)(int);

 #ifdef POSIX_SIGNAL
+#define ALT_STACK_SIZE (4*1024)
+#ifdef SIGSEGV
+/* alternate stack for SIGSEGV */
+static void register_sigaltstack() {
+    stack_t newSS, oldSS;
```

```
+
+    if(is_altstack_defined)
+      return;
+
+    newSS.ss_sp = malloc(ALT_STACK_SIZE);
+    if(newSS.ss_sp == NULL)
+      /* should handle error */
+       rb_bug("register_sigaltstack. malloc error\n");
+    newSS.ss_size = ALT_STACK_SIZE;
+    newSS.ss_flags = 0;
+
+    if (sigaltstack(&newSS, &oldSS) < 0)
+        rb_bug("register_sigaltstack. error\n");
+    is_altstack_defined = 1;
+}
+#endif
+
 static sighandler_t
 ruby_signal(int signum, sighandler_t handler)
 {
@@ -432,7 +458,12 @@
     if (signum == SIGCHLD && handler == SIG_IGN)
         sigact.sa_flags |= SA_NOCLDWAIT;
 #endif
-    sigaction(signum, &sigact, &old);
+#ifdef SA_ONSTACK
+    if (signum == SIGSEGV)
+        sigact.sa_flags |= SA_ONSTACK;
+#endif
+    if (sigaction(signum, &sigact, &old) < 0)
+        rb_bug("sigaction error.\n");
     return old.sa_handler;
 }

@@ -663,6 +694,7 @@
 #ifdef SIGSEGV
       case SIGSEGV:
         func = sigsegv;
+        register_sigaltstack();
         break;
```

```
 #endif
 #ifdef SIGPIPE
@@ -1070,6 +1102,7 @@
     install_sighandler(SIGBUS, sigbus);
 #endif
 #ifdef SIGSEGV
+    register_sigaltstack();
     install_sighandler(SIGSEGV, sigsegv);
 #endif
     }
```

このパッチを当てたプログラムを実行した結果は下記のとおりです。

```
$ ./ruby -e 'eval("1+" * 100000 + "1")'
-e:1: [BUG] Segmentation fault
ruby 1.9.0 (2008-11-01 revision 20086) [i686-linux]

-- control frame ----------
c:0004 p:---- s:0010 b:0010 l:000009 d:000009 CFUNC  :eval
c:0003 p:0017 s:0006 b:0006 l:000005 d:000005 TOP    -e:1
c:0002 p:---- s:0004 b:0004 l:000003 d:000003 FINISH :inherited
c:0001 p:0000 s:0002 b:0002 l:000001 d:000001 TOP    <dummy toplevel>:17
---------------------------
-e:1:in `eval': stack level too deep (SystemStackError)
         from -e:1:in `<main>'
```

いずれにせよセグメンテーションフォルト（SIGSEGV）で異常終了するのですが、何も付加情報を出力しないで終了するのと違って、異常終了のヒントを出してくれるので、アプリケーションをデバッグするときの助けになります。

なお、http://redmine.ruby-lang.org/repositories/revision/ruby-19?rev=20293 において、このパッチを元に修正が ruby に加えられました。

まとめ

スタックオーバーフローのため SIGSEGV で異常終了した場合のデバッグ方法について記しました。

参考文献

- 『BINARY HACKS』の「sigaltstack でスタックオーバーフローに対処する」[HACK

#76]（pp. 291-300）

—— Hiro Yoshioka

HACK #27 バックトレースが正しく表示されない

マルチスレッドアプリケーションで、スレッド間競合によりスタックが破壊されたケースを題材に説明します。

概要

スタック破壊によって、問題の解析が困難となることがあります。特に、バックトレース情報が得られなくなることで、問題現象に至るルートを追いかけにくくなります。また、スタック破壊が存在することで、バックトレース情報は完全ではないと言えます。デバッガでのバックトレースは万能ではないことを覚えておきましょう。

問題内容

とあるスレッド間通信を行うプログラムにバグがあり、core が生成されました。

バックトレース確認

デバッガで解析を行う際に、とりあえずバックトレース、というのが定石です。しかし、再現プログラム実行によって生成された core ファイルにおいて、バックトレースを見てみましたが、何がコールされているのかさっぱりわかりません。nanosleep() を実行中に SIGSEGV となったようですが、th_req() から nanosleep() に至るルートはどうなっているのでしょうか？

```
(gdb) bt
#0  0x0000003b4869ac80 in nanosleep () from /lib64/libc.so.6
#1  0x000ee1c2000ee1c1 in ?? ()
#2  0x000ee1c4000ee1c3 in ?? ()
#3  0x000ee1c6000ee1c5 in ?? ()
#4  0x000ee1c8000ee1c7 in ?? ()
#5  0x000ee1ca000ee1c9 in ?? ()
#6  0x000ee1cb000ee1ca in ?? ()
#7  0x000ee1cd000ee1cc in ?? ()
#8  0x0000000000000002 in ?? ()
#9  0x0000000001877c90 in ?? ()
#10 0x0000000004162f130 in ?? ()
#11 0x0000000000400d02 in th_req (p=0x1877c90) at bug.c:167
```

なぜ、このようなバックトレース情報となってしまったのでしょう？

これを解明するには、gdb などのデバッガがどのようにバックトレースを出力しているかを理解しておく必要があります。

バックトレースとは

デバッガのバックトレースは、スタック領域に積まれた関数のリターンアドレスを元にしています。スタック領域上のリターンアドレスとデバッグ情報からのスタック使用量をもとに、次々と呼び出し元関数を求めています。デバッガのバックトレースの元になるアドレス値はプロセスのスタック上に存在することになります。スタックについては**「デバッグに必要なスタックの基礎知識」[HACK #9]**を参照してください。

上記のとおり、バックトレース情報はスタック情報に依存しています。したがって、この例のようにバックトレースがおかしい場合、スタックが破壊されていると考えて、ほぼ間違いありません。

今回の例はかなり極端なもので、実際には、もう少しまともに見えるものもあります。しかし、スタック破壊となっている場合、デバッガの生成したバックトレース情報は信頼できません。少し極端に言うと、バックトレースを信頼することは、スタックが破壊されていないことを前提にしないと成り立ちません。デバッガのバックトレース情報が絶対正しいと思い込むことは危険です。

レジスタとスタックの確認

gdb による解析において、レジスタ情報を無視することはできません。現在のレジスタ情報を見てみましょう。

```
(gdb) info reg
…
rsp            0x4162f0c8       0x4162f0c8
…
rip            0x3b4869ac80     0x3b4869ac80 <nanosleep+96>
…
```

現在実行中の命令を確認します。命令ポインタ RIP の値は 0x3b4869ac80 となっています。何の命令を実行しようとしたのか、確認します。

```
(gdb) x/i 0x3b4869ac80
0x3b4869ac80 <nanosleep+96>:    retq
```

これは、retq 命令、関数からのリターン命令です。x86 における関数のリターンでは、

現在のスタックポインタのアドレスからリターンアドレスを取り出し、そのアドレスへジャンプする処理が行われます。したがって、ジャンプ先のアドレス、すなわちスタック上のリターンアドレスが不正であることが考えられます。

```
(gdb) x/g 0x4162f0c8
0x4162f0c8:     0x000ee1c2000ee1c1
```

この例では、アドレス 0x000ee1c2000ee1c1 にリターン、すなわちジャンプすることになります。この時点で、リターンアドレスが破壊されていることは疑いようがありません。明らかなスタック破壊と言えるでしょう。

```
(gdb) bt
#0  0x0000003b4869ac80 in nanosleep () from /lib64/libc.so.6
#1  0x000ee1c2000ee1c1 in ?? ()
```

そう、最初のバックトレースに出てきた結果は、デバッガとしては正当なもので、このプロセスにおけるスタックのデータ上、最後の nanosleep() はアドレス 0x000ee1c2000ee1c1 に存在する関数から呼ばれたことになっています。もちろん、このアドレス 0x000ee1c2000ee1c1 は正当なアドレスではありません。

このようにスタック破壊はバックトレース情報が得られないなど、デバッグを行う上での大きな障害となります。また、スタック領域は関数のローカル変数保存領域としても使用されます。ローカル変数の内容も破壊される可能性があります。つまり、gdb によるロー

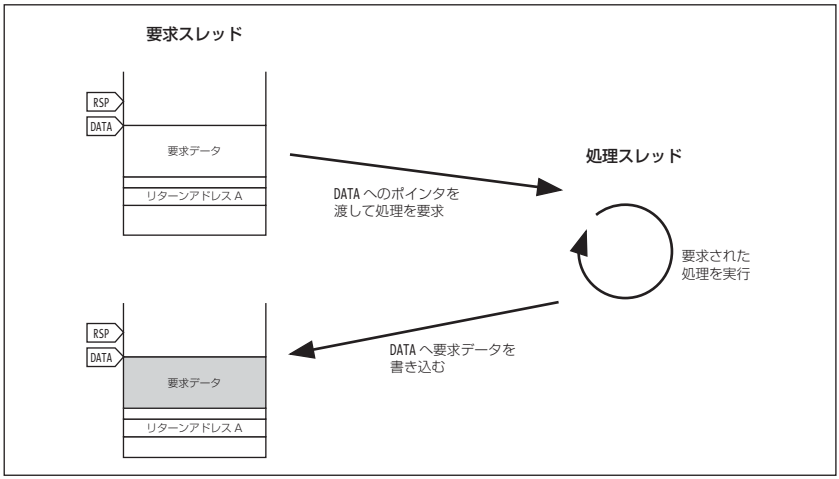

図 4-2　正常時のアプリケーションの動作

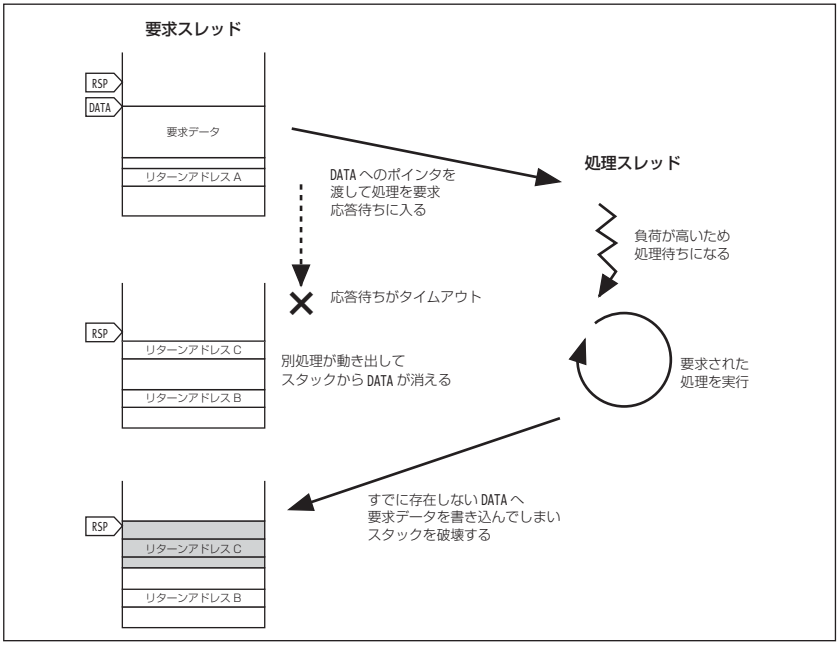

図 4-3　問題発生時のアプリケーションの動作

カル変数出力も信頼できなくなります。

スタック破壊の調査方法としては、いくつか考えられますが、もっとも現実的な手法は、破壊されたデータ内容から書き込んでいる箇所を特定することです。スタック領域、すなわち、自動変数領域への参照、ポインタを渡す処理がないかを確認しましょう。

このアプリケーションの場合、スレッド間のデータ処理において、スタックへのポインタを渡すことによって、他スレッドのスタックへの書き込み処理が存在しました。

そしてこの他スレッドのスタックへの書き込み処理が遅延することで、スタック破壊が発生していました。

図 4-2 と図 4-3 はそれぞれ、このアプリケーションの正常時の動作イメージと問題発生時の動作イメージを示しています。要求スレッドについては、スタックの状態を合わせて図示しています。

まとめ

本 Hack ではスタック破壊による問題について説明しました。スタック破壊が発生すると、バックトレース情報が取得できない、予想不能な変数破壊など、デバッグ自体が非常に困難になります。

逆にバックトレース情報がおかしいと感じたとき、スタック破壊を疑いましょう。

参考文献

- Intel® 64 and IA-32Architectures Software Developer's Manuals
  http://www.intel.com/products/processor/manuals/index.htm

—— Hiroshi Shimamoto

HACK #28 配列の不正アクセスによるメモリ内容の破壊

セグメンテーションフォルトを起こす原因のひとつである誤った配列操作をデバッグする方法を説明します。

配列の不正操作

配列を不正に操作する典型的なバグのひとつは、バッファオーバーランです。それは、確保されたメモリ領域の境界を越えたデータの書き込みです。とりわけ、スタック上のバッファに対するこの種のバグは、セキュリティホールを発生させ得るので、バッファサイズを指定するより安全な関数、ソースコードの検査ツール、コンパイラによるビルド時の警告表示など、その予防策や対応策は多数存在します。にもかかわらず、このバグは、いまだ、しばしば登場します。そのため、次節で説明するような状況があれば、このバグを疑ってみるべきです。また、配列のインデックスを計算によって求める場合、計算の方法にバグがあると、負の値を算出することがあり、そのような場合にもバッファオーバーランと似た現象を引き起こすことがあります。このようなバグのデバッグ方法も本 Hack の後半で説明します。

バッファオーバーランを疑う状況

バッファオーバーランを疑う状況のひとつは、ビルドオプションに -g を指定しているにも関わらず、core を GDB で読み込み、バックトレースを表示させると、次のように、スタックフレームに、シンボル名が表示されない場合です。通常は、-g オプションが指定されている場合、各スタックフレームに関数名が表示されます。

```
(gdb) bt
#0  0x20656c62 in ?? ()
#1  0x72727563 in ?? ()
#2  0x08040079 in ?? ()
#3  0x0804948c in ?? ()
#4  0xb8008ff4 in ?? () from /lib/libc.so.6
#5  0xb802aca0 in ?? () from /lib/ld-linux.so.2
#6  0x080483c0 in main ()
```

「バックトレースが正しく表示されない」[HACK #27]でも述べたように、このような場合、バックトレースは信用できません。実際、停止アドレスのコードをGDBで表示させようとしても、以下のように表示できません。これは、バックトレースで表示されている内容は、実際のトレースを示していない可能性が高いことを意味します。つまり、存在しない不正なアドレス0x20656c672への突然のジャンプまたはコールが行われ、その結果、セグメンテーションフォルトが発生したと考えることができます。

```
(gdb) x/i 0x20656c62
0x20656c62:     Cannot access memory at address 0x20656c62
(gdb) x/i 0x72727563
0x72727563:     Cannot access memory at address 0x72727563
```

スタックフレーム#0や#1に表示されているアドレスは、プログラムや共有メモリが配置されにくいアドレスです。i386アーキテクチャのLinuxディストリビューションでは多くの場合、プログラムは0x08000000番地付近に、共有ライブラリは0xb0000000番地以降に配置されます。x86_64ではプログラムは、0x400000や0x0600000番地付近に、共有ライブラリは、0x3000000000番地や0x2aaaaaaa0000番地付近に配置されます。これらは、リンカによって決められます。使用している環境によっては、リンカの配置アドレスに関するオプションが変更されている可能性はありますが、デバッグ対象の環境では、標準的にどのアドレスにプログラムや共有ライブラリが配置されるかを覚えておくとバックトレースを見るときの参考になります。

実行アドレスの変更

では、どこで存在しないアドレスへのジャンプやコールが行われたのでしょうか？この部分を特定する前に、プログラムに実行アドレスを変更させる方法を整理します。それらの方法は、大きく次の3つに分類されます。1つ目は直接アドレスを指定してのコールやジャンプです。これは、C言語でifやforなどによって条件分岐を行う場合や、同じソースファイル内の関数をコールする場合に使用されます。2つ目は、ジャンプ先のアドレスが格納されているメモリ領域のアドレスを指定する方法で、ライブラリ関数をコールする場合などに使用されます。（「GOT/PLTを経由した関数コールの仕組みを理解する」[HACK #63]参照）3つ目は、ret命令の実行で、関数の終了時、呼び出し元の関数に戻る時に使用されます。ret命令は、スタックポインタが指す領域の値をジャンプ（リターン）アドレスとして使用します「デバッグに必要なスタックの基礎知識」[HACK #9]参照）。

これらの方法で使用されるアドレスの値が、バグによって破壊されて（不正なアドレスに書き換えられて）しまった場合、不正なアドレスにジャンプする可能性があります。ただし、1つ目の方法で使用されるアドレスをバグが破壊することは困難です。なぜなら、

1つ目の方法で使用されるアドレスは、一般的に書き込みが禁止されているコード領域に存在するからです。そのため、そのアドレスを破壊しようとすると、その破壊命令が実行された時にセグメンテーションフォルトが発生します。この場合、core ファイルに、その瞬間のプログラムカウンタの値が記録されるので、解析は比較的容易です。

一方、2つ目と3つ目の方法が使用するアドレスは、GOT（Global Offset Table）やスタックなどの書き込み可能領域にあるので、バグがその領域を内容を破壊しても、その瞬間には、破壊を検出することができません。そのため、それよりも後になって、本 Hack の最初で示したセグメンテーションフォルトのような形で問題は顕在化します。

ジャンプ先のアドレス値を破壊する箇所の特定（スタック破壊）

アドレスの値を破壊する箇所を求める方法は、簡単に言えば、不正なアドレス値そのものをデータとしてコピーする箇所を探すことです。本 Hack 冒頭のスタックトレースでは、0x20656c62 というデータが不正に書き込まれる箇所を探索します。

このような調査では、データが文字列の一部でないかを疑うことは重要です。なぜなら、不正なアドレスへのデータ書き込みの典型的なパターンのひとつは、文字列のコピーだからです。文字列の入力長さは、予測されにくいため、バッファサイズが小さく、かつ、入力文字列長に対するチェックが不完全な場合に、しばしば、このような事態が発生します。この Hack の例でも、アドレス 0x20656c62 を、そのような観点で見るとアスキー文字列では「elb」であることに気づきます。x86_64 および i386 アーキテクチャはリトルエンディアンなので、「...ble ...」という文字列の一部が書き込まれたのではないかという仮説を立てることができます。ただし、偶然文字コードに対応する数値が4つ連続している可能性もあるので、この仮説の検証をさらに進めなければなりません。前の節で説明した3番目の方法では、スタックからリターンアドレスを取得しますので、スタックを調べてみます。

```
(gdb) x/30c $esp-15
0xbfc6f651:     100 'd' 105 'i' 110 'n' 103 'g' 32 ' '  118 'v'  101 'e'103 'g'
0xbfc6f659:     101 'e' 116 't' 97 'a' 98 'b'  108 'l' 101 'e' 32 ' ' 99 'c'
0xbfc6f661:     117 'u' 114 'r' 114 'r' 121 'y' 0 '\0'  4 '\004'        8 '\b'  -116 '\214'
0xbfc6f669:     -108 '\224'     4 '\004'        8 '\b'  -12 '??' -1 '??'  4 '\004'
(gdb) p (char*)$esp-20
$9 = 0xbfc6f64c " building vegetable curry"
```

上記のようにスタックポインタが指す領域の前後を表示するとほとんどが文字列ですので、スタックの誤った位置にこのような文字列を書き込んだ可能性が高まってきました。実際この問題を引き起こすソースコードを下記に示します。「building ...curry」という文字は、ソースコード中の names という文字列の一部であることがわかります。names が使用

されるのは、strcpy() で buf にコピーする箇所です。ここまで来ると、buf は 5 バイトしか確保されていないので、それ以上の長さの文字列をコピーしたこと（バッファオーバーラン）が原因とすぐにわかります。

```
char names[] = "book cat dog building vegetable curry";

void func(void)
{
    char buf[5];
    strcpy(buf, names);
}

int main(void)
{
    func();
    return EXIT_SUCCESS;
}
```

この様子を図にすると**図 4-4** のようになります。strcpy() が、main() へのリターンアドレスを格納している領域に文字列を書き込みます。そのため、func() を終了して、main に戻る際、リターンアドレスとして「ble」に相当する 0x20656c62 が使用され、セグメンテーションフォルトが発生しました。

この例では、アドレスが文字列の一部だったので、比較的特定がしやすいですが、単に数値を誤って書き込んだ場合、調査はいささか難しくなります。それでも、その数値が、そのプログラムによく登場する数値であれば、その数値を使用している部分に絞り込んで調査をすることができます。

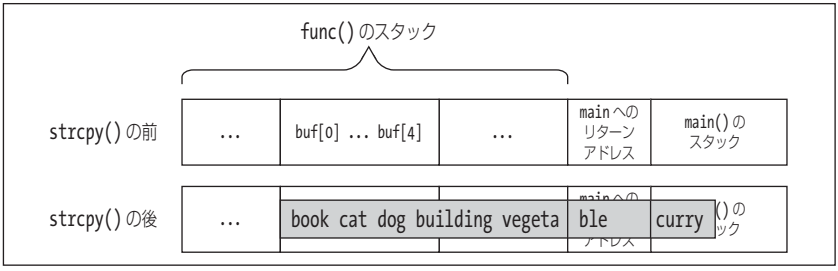

図 4-4 func() のスタック

ジャンプ先のアドレス値を破壊する箇所の特定 (GOT 破壊)

データ領域に静的に確保された配列のアクセスにバグあった場合も、似ような現象が発生します。ここでは、以下のようなバックトレースを持つ core を生成するプログラムのセグメンテーションフォルトを取り上げます。

```
(gdb) bt
#0  0x00000008 in ?? ()
#1  0x0000000a in ?? ()
#2  0x00000008 in ?? ()
#3  0x080483ca in main () at bufov2.c:19
```

この例でも、バックトレースの #0 に示されているアドレスは存在しません。また、その数値自体も 8 と原因を特定しにくいありふれた値です。唯一、スタックフレームの最後の bufov2.c:19 というのが、手がかりになりそうです。この 0x80483ca でどういう操作が行われているか調べるために逆アセンブルしてみます。

```
(gdb) disas 0x080483ca
...
0x080483be <main+64>:   movl   $0x80484a8,(%esp)
0x080483c5 <main+71>:   call   0x80482b0 <_init+56>
0x080483ca <main+76>:   mov    $0x0,%eax
..
```

これを見ると、スタックトレースに表示されている 0x08040483ca は、その前の call 命令によってスタックに積まれたのではないかと考えることができます。すなわち、0x080483c35 番地の call 命令が発行されてから、0x80483ca 番地に返ってくるまでの間に何らかの問題が起こっている可能性があります。コール先の 0x80482b0 では何が行われているか調べると、次のように 0x80495b0 番地に格納されている値のアドレス（0x080482b6）にジャンプすることがわかります。このようにして命令を追いかけていくと、0x08048296 番地の jmp 命令が参照しているアドレスの値が 0x8 ということを発見できます。この番地にジャンプしたことによってセグメンテーションフォルトが発生したと考えるとつじつまが合います。

```
(gdb) disas 0x80482b0 0x80482c0
Dump of assembler code from 0x80482b0 to 0x80482c0:
0x080482b0 <_init+56>:  jmp    *0x80495b0
0x080482b6 <_init+62>:  push   $0x8
0x080482bb <_init+67>:  jmp    0x8048290 <_init+24>
(gdb) x 0x80495b0
```

```
0x80495b0 <_GLOBAL_OFFSET_TABLE_+16>:   0x080482b6
(gdb) disas
0x080482b6 <_init+62>:  push   $0x8
0x080482bb <_init+67>:  jmp    0x8048290 <_init+24>
(gdb) disas 0x8048290 0x80482a0
Dump of assembler code from 0x8048290 to 0x80482a0:
0x08048290 <_init+24>:  pushl  0x80495a4
0x08048296 <_init+30>:  jmp    *0x80495a8
0x0804829c <_init+36>:  add    %al,(%eax)
0x0804829e <_init+38>:  add    %al,(%eax)
End of assembler dump.
(gdb) x 0x80495a8
0x80495a8 <_GLOBAL_OFFSET_TABLE_+8>:    0x00000008
```

このあたりは、PLTによってライブラリ関数を呼ぶコードです。詳しくは**「GOT/PLTを経由した関数コールの仕組みを理解する」[HACK #63]**を参照してください。

ではここに8を書き込んだ箇所はどこなのでしょうか？先にも書いたように、これは平凡な数値なので、特定は若干困難です。ただ、0x80495a8はGDBが表示しているようにGOTの一部なので、通常ユーザプログラムは、この領域に書き込みを行いません。そのため、8を格納するべきアドレスを誤って、GOT領域に書き込んだという可能性が考えられます。アドレスの間違い方（バグ）はさまざまなので、一概にどこが問題かを調べるのは難しいのですが、本来、書くべきアドレスが近くにあるという仮説に基づき、このアドレス付近の大局的な構造を調べると以下のようになっています。

```
$ objdump -s bufov2
...
Contents of section .fini:
 8048484 5589e553 e8000000 005b81c3 13110000  U..S.....[......
 8048494 50e876fe ffff595b c9c3               P.v...Y[..
Contents of section .rodata:
 80484a0 03000000 01000200 54686973 20697320  ........This is
 80484b0 61206d65 73736167 650a00             a message..
Contents of section .eh_frame:
 80484bc 00000000                             ....
Contents of section .ctors:
 80494c0 ffffffff 00000000                    ........
Contents of section .dtors:
 80494c8 ffffffff 00000000                    ........
```

```
Contents of section .jcr:
 80494d0 00000000                             ....
Contents of section .dynamic:
 80494d4 01000000 24000000 0c000000 78820408  ....$.......x...
...
 8049594 00000000 00000000                    ........
Contents of section .got:
 804959c 00000000                             ....
Contents of section .got.plt:
 80495a0 d4940408 00000000 00000000 a6820408  ................
 80495b0 b6820408                             ....
Contents of section .data:
 80495b4 00000000 00000000 cc940408 01000000  ................
 80495c4 02000000
...
```

これによると、破壊されたアドレス 0x80495a8 は、.got.plt セクションにあり、すぐ後ろには、ユーザの静的データが格納される .data セクションがあります。この状況から考えられるひとつの可能性は、.data セクションのどこかに書き込むべきデータを誤って、.got.plt セクションに書き込んだということです。アドレスの間違いの多くは、ポインタに対する操作に誤りか、配列のインデックスの間違いです。このような観点でソースを見てみましょう。

```
int my_data[2] = {1, 2};

int calc_index(void)
{
    /* この関数にバグがあり、誤った値を返す */
    return  -7;
}

int main(void)
{
    int idx = calc_index();
    my_data[idx] = 0x0a;
    my_data[idx+1] = 0x08;
    printf("This is a message\n");
    return EXIT_SUCCESS;
}
```

ソース中には、静的な配列 my_data があり、これに対して、書き込みをしている箇所があります。そのインデックスは、calc_index() にて計算されています。ここまでくると、calc_index() の返す値が誤っているのではないかと考えることができます。実際、このソースでは、見てすぐわかるように calc_index() は誤った負の値を返しています。実際の例では、もう少しインデックスの算出が複雑でしょうから、なんとなくソースを眺めているだけではバグを発見しにくいものです。しかし、この関数が怪しいというところまで突き止めれば、注意してその関数内の処理を調べることでバグを格段に発見しやすくなります。

まとめ

配列の不正操作でメモリが破壊されることによるセグメンテーションフォルトをデバッグする方法を紹介しました。メモリ内容が破壊されていることを確認するまでは、一定の手順で行えますが、破壊を行う箇所を特定するのは、ある程度勘と経験に頼らざるを得ません。そこで、**「ウォッチポイントを活用した不正メモリアクセスの検知」[HACK #29]** では、GDB の watch コマンドを使ってメモリ内容を破壊する瞬間を検出する方法を紹介します。

—— Kazuhiro Yamato

HACK #29 ウォッチポイントを活用した不正メモリアクセスの検知

指定した変数やアドレスのデータにアクセスがあった場合にプログラムを停止するウォッチポイントの使い方について例を交えながら説明します。

ウォッチポイントが有効な状況

「配列の不正アクセスによるメモリ内容の破壊」[HACK #28] では、不正にメモリが書き換えられることが原因でセグメンテーションフォルトが発生する例を紹介しました。**[HACK #28]** の解析アプローチは、基本的に core ファイルを解析し、いくつかの状況証拠から不正なメモリ書き換えが行われる箇所を特定します。しかし、デバッグ対象のプログラムが手元にあり、すぐに現象を再現できる場合、GDB のウォッチポイントを利用して、効率的にバグを特定することができます。本 Hack では、このウォッチポイントを用いたデバッグ方法を説明します。

[HACK #28] では、不正に書き換えられる領域のアドレスが、0x80495a8 であることまでは、core ファイルを調査することで、比較的容易に特定できました。以下では、実際に core ファイルを生成したプログラムを実行させて、その不正な書き換え箇所をウォッチポイントで検出する方法を説明します。ウォッチポイントは、指定された変数などが読み出しや書き込みを行う時に、実行を中断します。したがって、その中断箇所を調べることで意図しない書き込みか否かを調べることができます。

ウォッチポイントの設定方法

ウォッチポイントを設定するには、下記のように入力します。

```
(gdb) watch *0x80495a8
Hardware watchpoint 1: *134518184
```

変数名やシンボルでなく、直接アドレスを指定する場合は、アドレスの前に * を付けます。これは、break コマンドでアドレスを直接指定する場合と同じです。それでは、この状態でプログラムを実行させます。

```
(gdb) run
Starting program: /home/yamato/tmp/bufovrn.work/bufov2
Hardware watchpoint 1: *134518184
Hardware watchpoint 1: *134518184

Old value = 0
New value = -1208018240
_dl_relocate_object (l=0xb7ffc710, scope=0xb7ffc8c8, lazy=1,
    consider_profiling=0) at dl-reloc.c:268
268         ELF_DYNAMIC_RELOCATE (l, lazy, consider_profiling);
(gdb) x/i -1208018240
0xb7ff1ac0 <_dl_runtime_resolve>:       push   %eax
```

0x80495a8 に書き込みが発生したため、プログラムが停止しました。New value の右辺の数値が、指定したアドレスに新たに書き込まれた値です。しかし、この場合、New value は、16 進数では 0xb7ff1ac0 であり、これは、glibc に含まれる _dl_runtime_resolve という関数（ライブラリ関数のアドレスを調べて GOT に設定する関数）のアドレスです。また、core 調査時、不正に書き込まれた数値は 0x8 でしたので、これは正常な、この領域への書き込みだと判断できます。そのため、さらにプログラムを続行させます。

```
(gdb) c
Continuing.
Hardware watchpoint 1: *134518184
Hardware watchpoint 1: *134518184

Old value = -1208018240
New value = 8
main () at bufov2.c:19
19          printf("This is a message\n");
```

上記の New value は 8 であり、まさにわれわれが探していた書き込みです。表示されているソースコードは 19 行目の printf 文が、ウォッチポイントになる停止位置ですが、停止を引き起こしたのは、この printf() ではなさそうです。というのも、ウォッチポイントを使っての停止は、ブレークポイントと違い、データへのアクセスが行われた後に発生します。そのため、C のソースの次の行が表示されることがあります。

上記ではウォッチポイントを GOT 領域に設定しています。GOT については、**「GOT/PLT を経由した関数コールの仕組みを理解する」[HACK #63]** を参照ください。

問題の絞り込み

さらに問題を解析するためには、アセンブリコードを見たほうがよさそうです。

```
(gdb) p $pc
$4 = (void *) 0x80483be
(gdb) disas
Dump of assembler code for function main:
...
0x0804839a <main+28>:   call   0x8048374 <calc_index>
0x0804839f <main+33>:   mov    %eax,0xfffffffc(%ebp)
0x080483a2 <main+36>:   mov    0xfffffffc(%ebp),%eax
0x080483a5 <main+39>:   movl   $0xa,0x80495c0(,%eax,4)
0x080483b0 <main+50>:   mov    0xfffffffc(%ebp),%eax
0x080483b3 <main+53>:   movl   $0x8,0x80495c4(,%eax,4) ——①
0x080483be <main+64>:   movl   $0x80484a8,(%esp)
0x080483c5 <main+71>:   call   0x80482b0 <_init+56>
...
End of assembler dump.
```

停止アドレスの 1 つ前①の movl 命令は 8 という数値をどこかのアドレスに書き込んでいます。そのアドレスは、具体的には 0x80495c4 + %eax*4 です。%eax の値と、アドレスの計算結果は次のようにして求められます。

```
(gdb) p $eax
$5 = -7
(gdb) p/x (0x80495c4 + $eax*4)
$6 = 0x80495a8
```

このアドレス 0x80495a8 は、不正な書き込みが行われていたアドレスですので、この部

分が直接の原因であることが判明しました。 対応する C のソースは、次のようにして求められます。

```
(gdb) list *0x080483b3
0x80483b3 is in main (bufov2.c:18).
13

14      int main(void)
15      {
16          int idx = calc_index();
17          my_data[idx] = 0x0a;
18          my_data[idx+1] = 0x08;
19          printf("This is a message\n");
20          return EXIT_SUCCESS;
21      }
```

ここまでくると、idx の値がおかしいのではないかという仮説を容易に立てることができます。実際に調べてみると、下記のようにこの状況ではおかしい負の値になっており、バグが calc_index() の中にあることがわかります。

```
(gdb) p idx
$10 = -7
```

上記のように list コマンドの引数に直接アドレスを指定すると、そのアドレスに対応するソースファイル : 行番号と、その行を含む関数が表示されます。

まとめ

指定した変数やアドレスのデータに書き込みあった場合にプログラムを停止するウォッチポイントの使い方を例を交えながら説明しました。

—— Kazuhiro Yamato

HACK #30

malloc() や free() で障害が発生

メモリの二重解放によるバグとその対処法を紹介します。

メモリ関連ライブラリ関数の不正使用によるバグ

アプリケーション、特に C 言語によるバグとして、メモリ関連ライブラリ関数の不正使用によるバグが良くあります。いわゆる、メモリの二重解放や、確保した領域範囲外の

使用がこれにあたります。

この手のバグは主にメモリ操作ライブラリ関数の不正処理として顕在化します。つまり、malloc() や free() の先で SIGSEGV となる現象が発生します。

```
$ gdb ./membug -c core
Core was generated by `./membug'.
Program terminated with signal 11, Segmentation fault.
#0  0x0000003b4867217c in _int_free () from /lib64/libc.so.6
(gdb) bt
#0  0x0000003b4867217c in _int_free () from /lib64/libc.so.6
#1  0x0000003b48675f2c in free () from /lib64/libc.so.6
#2  0x0000000000400534 in do_free () at membug.c:30
#3  0x0000000000400588 in main (argc=<value optimized out>,
  argv=<value optimized out>) at membug.c:39
```

このバックトレース情報から、メモリ解放を行うライブラリ関数 free() やその内部で使用されている関数 _int_free() に問題があるよう見えますが、実際は free() を使用しているアプリケーションの問題で、ライブラリ関数 free() に問題はありません。

なお、このように malloc() や free() の延長で SIGSEGV となるものは、運が良いパターンだと考えられます。

本当に危険なパターンとしては、メモリ破壊に気づかず動作し続けるというものがありえます。その場合、

- 全く関係ない箇所で SIGSEGV となる
- 壊れたデータを使って演算し続け、間違った結果を返す

といった状況が考えられます。

malloc() や free() の延長で SIGSEGV となった場合、ライブラリ関数のバグを疑う前に使用方法に問題がないか、特に、二重解放を行っていないかをしっかり確認しましょう。

MALLOC_CHECK_ によるデバッグ

最近の glibc の場合、環境変数を用いた便利なデバッグ用フラグがあります。

```
$ man malloc
```

で確認できます。

その効果を確認してみましょう。

環境変数 MALLOC_CHECK_ に値を入れることでメモリ操作関連のバグを見つけることが可能です。

下記例では、double free、すなわち二重解放を検出しています。

```
$ env MALLOC_CHECK_=1 ./membug
*** glibc detected *** ./membug: double free or corruption (top): 0x00000000020b2010 ***
======= Backtrace: =========
/lib64/libc.so.6[0x3b48672832]
/lib64/libc.so.6(cfree+0x8c)[0x3b48675f2c]
./membug[0x400534]
./membug[0x400588]
/lib64/libc.so.6(__libc_start_main+0xf4)[0x3b4861e074]
./membug[0x400449]
======= Memory map: ========
00400000-00401000 r-xp 00000000 08:06 42927111                           /data/mywork/mines/0002/membug
00600000-00601000 rw-p 00000000 08:06 42927111                           /data/mywork/mines/0002/membug
020b2000-020d3000 rw-p 020b2000 00:00 0                                  [heap]
3b48200000-3b4821b000 r-xp 00000000 08:02 7057200                        /lib64/ld-2.7.so
3b4841a000-3b4841b000 r--p 0001a000 08:02 7057200                        /lib64/ld-2.7.so
3b4841b000-3b4841c000 rw-p 0001b000 08:02 7057200                        /lib64/ld-2.7.so
3b48600000-3b4874d000 r-xp 00000000 08:02 7057202                        /lib64/libc-2.7.so
3b4874d000-3b4894d000 ---p 0014d000 08:02 7057202                        /lib64/libc-2.7.so
3b4894d000-3b48951000 r--p 0014d000 08:02 7057202                        /lib64/libc-2.7.so
3b48951000-3b48952000 rw-p 00151000 08:02 7057202                        /lib64/libc-2.7.so
3b48952000-3b48957000 rw-p 3b48952000 00:00 0
3b53a00000-3b53a0d000 r-xp 00000000 08:02 7057488                        /lib64/libgcc_s-4.1.2-20070925.
so.1
3b53a0d000-3b53c0d000 ---p 0000d000 08:02 7057488                        /lib64/libgcc_s-4.1.2-20070925.
so.1
3b53c0d000-3b53c0e000 rw-p 0000d000 08:02 7057488                        /lib64/libgcc_s-4.1.2-20070925.
so.1
7f8540000000-7f8540021000 rw-p 7f8540000000 00:00 0
7f8540021000-7f8544000000 ---p 7f8540021000 00:00 0
7f854738c000-7f854738e000 rw-p 7f854738c000 00:00 0
7f85473ae000-7f85473b0000 rw-p 7f85473ae000 00:00 0
7fff4f39b000-7fff4f3b0000 rw-p 7ffffffea000 00:00 0                      [stack]
7fff4f3ff000-7fff4f400000 r-xp 7fff4f3ff000 00:00 0                      [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
Aborted (core dumped)
```

このように環境変数 MALLOC_CHECK_ を利用することで、アプリケーションにおける二重解放バグを容易に発見することができます。

ただし、すべての場合において対応できるわけではないことも覚えておく必要があるでしょう。MALLOC_CHECK_ を用いた実行で問題検出されなければ、アプリケーションにバグがないとは言いきれません。

まとめ

本 Hack では、malloc() や free() などメモリ操作ライブラリ関数で顕在化する問題、すなわちメモリの二重解放、不正な領域の解放、について説明しました。

malloc() や free() など標準 C ライブラリのメモリ操作関数の延長で問題が発生する場合、最初に二重解放などアプリケーションのバグを疑いましょう。その際、最近の glibc を用いているのであれば環境変数 MALLOC_CHECK_ によるデバッグが効果的です。

メモリ二重解放によるバグはメモリリークと並んで真の原因を発見することが難しいバグのひとつと言えます。

参考文献

- Manpage of MALLOC
  http://www.linux.or.jp/JM/html/LDP_man-pages/man3/malloc.3.html

—— Hiroshi Shimamoto

HACK #31 アプリケーションのストール（デッドロック編）

pthread_mutex_lock でデッドロックを起こし、ストールするプログラムのデバッグ方法を紹介します。

デッドロックのサンプル

ここでは、ストールの典型であるデッドロックを、以下のサンプルコード（astall.c）を使って、そのデバッグ手順を見ていきましょう。このプログラムは、ロック（mutex）を取得した状態で、ある関数 cnt_reset() を呼び、その中で再度、同じロックを取得してデッドロックします。設計が不十分なプログラムに、潜みがちなバグです。

[astall.c]

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
int cnt = 0;

void cnt_reset(void)
{
    pthread_mutex_lock(&mutex);
    cnt = 0;
    pthread_mutex_unlock(&mutex);
}

void* thr(void *arg)
{
    while(1) {
        pthread_mutex_lock(&mutex);
        if (cnt > 2)
            cnt_reset();
        else
            cnt++;
        pthread_mutex_unlock(&mutex);

        printf("%d\n",cnt);
        sleep(1);
    }
}

int main(void)
{
    pthread_t tid;

    pthread_create(&tid, NULL, thr, NULL);
    pthread_join(tid, NULL);

    return EXIT_SUCCESS;
}
```

このようなプログラムは、実行中、突然動作が停止します。いわゆるストール状態になります。上記のプログラムを実行させると、次のように3まで表示されところで、ストールします。

```
$ ./astall
1
2
3
<< ここで0の表示を期待しているが、しばらく待っても表示されない >>
```

ストールした場合の対応方法

一般的に、ストールが発生した場合、まず、暴走しているのか、スリープしているのかを確認するべきです。これには、ps コマンドを使います。ps コマンドの出力の左から 3 つの項目は、プロセスの状態を表します。ここが R なら、そのプロセスは実行状態にあります。つまり、暴走している可能性が高いと言えます。一方、S ならスリープしています。意図しない長時間スリープが続く場合、デッドロックに陥っていることが多くあります。では、上記の例、astall の場合、実行状態はどうなっているでしょうか。ps コマンドで確認すると、以下のようになっており、状態が S なので、デッドロックが疑われます。

```
$ ps ax -L | grep astall
 2365  2365 pts/4    Sl+    0:00 ./astall
 2365  2366 pts/4    Sl+    0:00 ./astall
```

オプション -L は、すべてのスレッドを表示します。

そこで、GDB でこのプロセスにアタッチして、どこでスリープしているのか調べてみます。

```
$ gdb -p `pidof astall`
...
(gdb) bt
#0  0xb7f47430 in __kernel_vsyscall ()
#1  0xb7f18bf7 in pthread_join () from /lib/tls/i686/cmov/libpthread.so.0
#2  0x08048634 in main () at astall.c:35
```

起動して、単に bt コマンドを入力すると、動作が止まっているスレッドではなく、最初に起動されたスレッドのバックトレースが表示されました。このスレッドが pthread_join() でスリープしているのは、期待どおりなので、もう一方のスレッドに切り替え、bt コマンドを実行します。

```
(gdb) i thr
  2 Thread 0xb7db1b90 (LWP 29894)  0xb7f47430 in __kernel_vsyscall ()
  1 Thread 0xb7db26b0 (LWP 29893)  0xb7f47430 in __kernel_vsyscall ()
(gdb) thr 2
[Switching to thread 2 (Thread 0xb7db1b90 (LWP 29894))]#0  0xb7f47430 in __kernel_vsyscall ()
(gdb) bt
```

```
#0  0xb7f47430 in __kernel_vsyscall ()
#1  0xb7f1ed09 in __lll_lock_wait () from /lib/tls/i686/cmov/libpthread.so.0
#2  0xb7f1a114 in _L_lock_89 () from /lib/tls/i686/cmov/libpthread.so.0
#3  0xb7f19a42 in pthread_mutex_lock () from /lib/tls/i686/cmov/libpthread.so.0
#4  0x08048576 in cnt_reset () at astall.c:10
#5  0x080485af in thr (arg=0x0) at astall.c:20
#6  0xb7f1850f in start_thread () from /lib/tls/i686/cmov/libpthread.so.0
#7  0xb7e957ee in clone () from /lib/tls/i686/cmov/libc.so.6
```

すると、#3 の pthread_mutex_lock() をきっかけにしてカーネルモードになり、スリープしていることがわかります。つまり、pthread_mutex_lock() を別の箇所や別のスレッドが取得したまま、再度、pthread_mutex_lock() が呼ばれている可能性が高いと考えられます。

上記のバックトレースより、このスレッドは、astall.c:10 行目で、呼ばれている pthread_mutext_lock() でデッドロックしていることはわかりましたが、このような問題では、先に pthread_mutex_lock() をコールした箇所がどこかを把握する必要があります。一般的に、デッドロックは、ロックやアンロックが何度もコールされた後に発生します。そのため、ブレークポイントを設定しても、手動でデッドロックが発生するまで動作を確認していくことは、困難です。ここでは、以下のような次の GDB コマンドファイルを使用し、デッドロックが発生するまでの動作を自動的に記録します。このコマンドファイルは、pthread_mutex_lock() と pthread_mutex_unlock() が呼ばれる度にバックトレースを表示し、かつ、その内容を debug.log というファイルに記録します。

```
set pagination off                                  ①
set logging file debug.log
set logging overwrite                               ②
set logging on
start
set $addr1 = pthread_mutex_lock
set $addr2 = pthread_mutex_unlock                   ③
b *$addr1
b *$addr2
while 1
    c
    if $pc != $addr1 && $pc != $addr2               ④
        quit
    end
    bt
end
```

上記コマンドファイル中の①は、端末の行数以上のメッセージが表示されても、一時停止をせず、そのまま表示させ続けるための設定です。②は、画面に表示される内容をファイルにも記録します。set logging overwrite は、すでに同様のファイル名がある場合に上書きをします。③は、pthread_mutex_lock() と pthread_mutex_unlock() にブレークポイントを設定します。また、これらのアドレスを、後の④でも用いるため、コンビニエンス変数 $addr1、$addr2 に保存します。デバッグ対象のプログラムは、実行するとデッドロックします。終了するためには、キーボードで、Ctrl-C を入力します。④は、そのときのブレークで、GDB を終了するための処理です。

上記のコマンドファイルを debug.cmd という名前で保存して、以下のように実行します。

```
$ gdb astall -x debug.cmd
…
<< 約 60 行表示される >>
…
Breakpoint 2, 0xb7fc39d0 in pthread_mutex_lock () from /lib/tls/i686/cmov/libpthread.so.0
#0  0xb7fc39d0 in pthread_mutex_lock () from /lib/tls/i686/cmov/libpthread.so.0
#1  0x08048576 in cnt_reset () at astall.c:10
#2  0x080485af in thr (arg=0x0) at astall.c:20
#3  0xb7fc250f in start_thread () from /lib/tls/i686/cmov/libpthread.so.0
#4  0xb7f3f7ee in clone () from /lib/tls/i686/cmov/libc.so.6
<< ここでメッセージ出力が止まるので、Ctrl-C で終了 >>
```

上記の作業で、生成されたログ debug.log を確認していけば、どのようにしてデッドロックに至ったのかを確認できます。ただし、多くの不要な情報もログの中に含まれており、少々見にくいので、次のように grep や sed、あるいは Ruby などで整形してもよいでしょう。

```
$ cat debug.log | grep -A1 "^#0.*pthread_mutex_" | sed s/from\ .*$// | sed s/.*\ in\ //
pthread_mutex_lock ()
_dl_addr ()
--
pthread_mutex_unlock ()
_dl_addr ()
--
pthread_mutex_lock ()
thr (arg=0x0) at astall.c:18
--
pthread_mutex_unlock ()
thr (arg=0x0) at astall.c:23
```

```
--
pthread_mutex_lock ()
thr (arg=0x0) at astall.c:18
--
pthread_mutex_unlock ()
thr (arg=0x0) at astall.c:23
--
pthread_mutex_lock ()
thr (arg=0x0) at astall.c:18
--
pthread_mutex_lock ()
cnt_reset () at astall.c:10
```

これを見ると、最後の部分で、astall.c:18 からのロックした後に、astall.c:10 で再びロックしたことがわかります。

複数の mutex を用いている場合のデバッグ

前節の例では、ロックとアンロックが表示された時、実行をブレークさせ、バックトレースを表示させました。複数の mutex が使われる場合、これだけでは不十分です。どの mutex に対する操作かを明確にする必要があります。pthread_muxte_lock() や pthread_mutex_unlock() の引数は、mutex のアドレスなので、**「関数コール時の引数の渡され方（x86_64 編）」[HACK #10]** や **「関数コール時の引数の渡され方）i386 編）」[HACK #11]** で紹介した方法で引数を調べれば、どの mutex に対する操作かがわかります。例えば、i386 なら以下のようなコマンドを、先ほどのコマンドファイルの bt コマンドの前後に挿入するとよいでしょう。

```
printf "## addr: %08x\n", *(int*)($esp+4)
```

まとめ

pthread_mutex_lock でデッドロックを起こし、ストールするプログラムのデバッグ方法を紹介しました。

—— Kazuhiro Yamato

HACK #32 アプリケーションのストール（無限ループ編）

tcpdump にあった実際のバグを例に、ユーザアプリケーションの無限ループを解析します。

ネットワークプロトコル SCTP のセキュリティについて試験をしたときのバグについて紹介します。

SCTP とは SIGTRAN のワーキンググループで定義された電話網の制御信号（SS7）を

IP上で転送するためのプロトコルで、TCPと同じトランスポートレイヤプロトコルです。

セキュリティの脆弱性でも定番であるパケット長を0にしたデータ送受信を実施しました。TPは複雑であるため、本Hackには載せませんが、RAWソケットでSCTPパケットの中身を自由に変更できるものを使用しました。

TPではSCTP DATAチャンクのlengthメンバを0に設定してSCTPパケットを送信します。意図したパケットが送信されているか、対向のマシンでtcpdumpを実行しパケットの中身を確認します。

tcpdumpをコンパイルする

tcpdumpのバージョンはコミュニティのオリジナル3.8.2を使います。これはRHEL4に含まれるtcpdumpのベースバージョンです。

```
[@target ]# wget -t0 -c http://www.tcpdump.org/release/tcpdump-3.8.2.tar.gz
[@target ]# tar zxvf tcpdump-3.8.2.tar.gz
[@target ]# cd tcpdump-3.8.2
[@target tcpdump-3.8.2]# ./configure ; make
```

環境によっては以下のようなエラーになりますので、config.hを変更して再度makeします（RHEL4のRPMパッケージではこのエラーがでないように修正されています）。

```
# make
tcpdump.o(.text+0x894): In function `main':
: undefined reference to `pcap_debug'
collect2: ld はステータス 1 で終了しました
make: *** [tcpdump] エラー 1
# vi config.h
...
/* define if libpcap has pcap_debug */
//#define HAVE_PCAP_DEBUG 1                /* この行をコメントアウト */
...
# make clean ; make
```

パケットの確認

パケットを確認するためにtcpdumpを実行します。

tcpdumpでは詳細を表示させるため、習慣的にオプション-vvvXを付けました。すると以下のようなメッセージが大量に出力され異常な動作をしました。

```
[@target tcpdump-3.8.2]# ./tcpdump -ieth2 -vvvX sctp
…
00:21:12.747262 IP (tos 0x2,ECT(0), ttl  64, id 0, offset 0, flags [DF], proto
132, length: 56) 192.168.0.145.56934 > 192.168.0.155.49560: sctp
  1) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
  2) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
  3) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
  4) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
…
  2764) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
  2765) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
  2766) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
  2767) [DATA] (B)(E) [TSN: 4051946038] [SID: 0] [SSEQ 0] [PPID 0x0] [Payload]
…
```

「数字）[DATA] (B)(E) ・・・」が大量に出力され続け、ループしているように見えます。Ctrl-cでは終了しなかったため、Ctrl-zでプロセスを止めます。

```
[1]+  Stopped                 ./tcpdump -ieth2 -vvvX sctp
```

kill -9 でプロセスを終了させることはできました。

今回は前提として length を 0 にしているわけですから、おそらく原因もそれなはずです。ソースコードで SCTP DATA チャンクの length を処理している箇所を見ればよいのですが、ここではすぐにソースコードを見ずに、まずは調査範囲をできる限り狭めることをしたいと思います。

オプションにより動作に変化があるか確認する

tcpdump のオプションを変えて確認します。-v オプションのみの場合は正常に動作しました。-vv と -vvv でも正常に動作はしましたが、以下のようなメッセージが出力されました。

```
[@target tcpdump-3.8.2]# ./tcpdump -ieth2 -vvv sctp
…
00:21:41.678060 IP (tos 0x2,ECT(0), ttl  64, id 0, offset 0, flags [DF], proto
132, length: 56) 192.168.0.145.13727 > 192.168.0.155.59671: sctp
        1) [DATA] (B)(E) [TSN: 2466762381] [SID: 0] [SSEQ 0] [PPID 0x0]
[Payload:bogus chunk length 0]
…
```

DATA チャンクの length が 0 と出力されています。このことから TP は意図した SCTP パケットを送信しているようです。

-X、-vX の場合を確認しましたが、現象は発生しませんでした。オプションを -vvX にしたところ現象が再現しました。ソースコードを調査するときはこの情報を利用します。

ブレークポイントの選定

gdb でブレークポイントの設定箇所を選定します。main() にブレークポイントを設置してもよいのですが、解析しにくいためループの可能性がある箇所にブレークポイントを設定します。

[tcpdump-3.8.2/print-sctp.c]

```
 62 void sctp_print(
    …
128   if (vflag < 2)       /* -v の数が 2 より少ない場合はここで return するため */
129     return;            /* これより下にブレークポイントを設定する */
        …
156     case SCTP_DATA :  /* DATA チャンクの場合は以下に入る */
          …
160       printf("[DATA] ");   /* ブレークポイント */
          …
196       printf("[Payload");

198       if (!xflag && !qflag) {   /* -X オプションがないと以下を処理するため */
          …                       /* この if 文内にブレークポイントを設定してはいけない */
204               printf("bogus chunk length %u]",
205                   htons(chunkDescPtr->chunkLength));
206               return;
207           }
          …
212       } else
213           printf("]");
214         }
215       break;
    …
```

[DATA] が連続して出力されていましたので、print-sctp.c の 160 行目にブレークポイントを設定します。

tcpdump をデバッグオプションをつけて再コンパイルし、再現させます（以下の手順は一部省略します）。

```
[@target tcpdump-3.8.2]# ./configure CFLAGS=-g
...
[@target tcpdump-3.8.2]# gdb ./tcpdump
...
(gdb) b print-sctp.c:160
Breakpoint 1 at 0x42d60b: file ./print-sctp.c, line 160.
(gdb) run -ieth2 -vvX sctp
Starting program: /root/nooiwa/tcpdump-3.8.2/tcpdump -ieth2 sctp -vvX
...
Breakpoint 1, sctp_print (bp=0x6da324 "K\210<\220 \b, bp2=Variable "bp2" is not
available.
) at ./print-sctp.c:160
160                 printf("[DATA] ");
(gdb)
```

現象を再現させると、gdb はブレークポイントで止まります。

ステップ実行による現象の確認

ステップ実行をすると同じコードを繰り返し、ループになっていました。DATA チャンクの length が原因なのか確認します。

```
(gdb) s
...
145           chunkEnd = ((const u_char*)chunkDescPtr + EXTRACT_16BITS(&chunkDescPtr->chunkLength));
(gdb) p chunkDescPtr
$1 = (const struct sctpChunkDesc *) 0x6da330
(gdb) p chunkDescPtr->chunkLength
$2 = 0
(gdb) s
...
151           nextChunk = (const void *) (chunkEnd + align);
(gdb) p nextChunk
$3 = (const void *) 0x6da330
(gdb) p chunkEnd
$4 = (const u_char *) 0x6da330 ""
(gdb) s
```

SCTP パケットの構造

SCTP パケットのフォーマットを図 4-5 に示します。

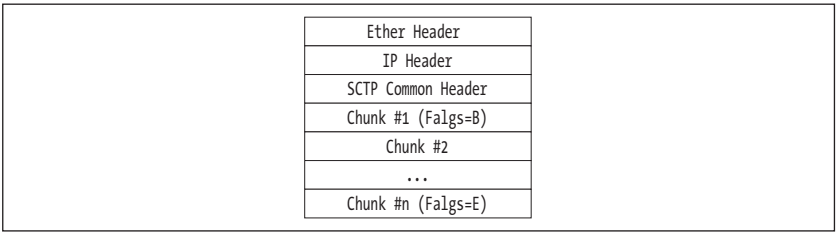

図 4-5　SCTP パケットフォーマット

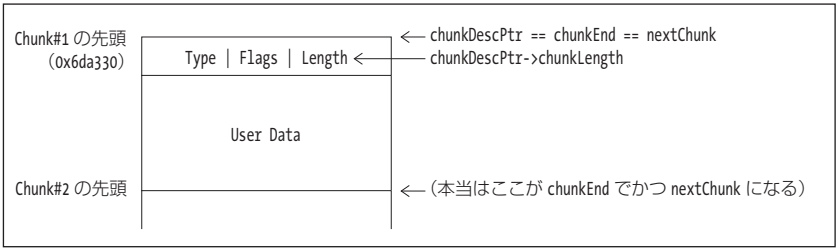

図 4-6　SCTP DATA チャンクフォーマット

SCTP は共通ヘッダとチャンクと呼ばれるデータの集まりで構成されます。チャンクには複数の種類がありますが DATA チャンクの場合、SCTP はチャンクの長さを見てパケットにできる限り複数の DATA チャンクを詰めようとします。

SCTP DATA チャンクのフォーマットを図 4-6 に示します。

Type は DATA チャンクであること示します。length は DATA チャンクの長さを示します。tcpdump はまず Chunk#1 を見ます。その次に Chunk#2 を見ますが、そのときは Chunk#1 のポインタを Length だけずらす作りになっていました。ポインタに Length を加え Chunk#2 の先頭になるはずが、今回は length が 0 なため Chunk#1 の先頭になってしまいます。

これが繰り返されるため無限ループとなっています。大量に出力されていたメッセージ「数字) [DATA] (B)(E) …」ですが、数字はパケットにあるチャンクの数になります。(B)(E) は図 4-6 の Flags で最初のチャンク（(B)eginning）と最後のチャンク（(E)nding）になります。これはパケットの中でこのチャンクが最初で最後を同時に示しているため次のチャンク（Chunk#2）がないことを示しています。

上位のバージョンを確認する

原因は判明しましたが、修正方法はいろいろ考えられます。

上位のバージョンですでに修正されている可能性もありますので、tcpdump 3.9.8 で試

してみると、正常に動作しました。

```
[@target tcpdump-3.9.8]# ./tcpdump -ieth2 -vvX sctp
...
00:16:51.581458 IP (tos 0x2,ECT(0), ttl 64, id 0, offset 0, flags [DF], proto
SCTP (132), length 56) 192.168.0.145.51816 > 192.168.0.155.10295: sctp
        1) [Bad chunk length 0]
        0x0000:  4502 0038 0000 4000 4084 b7c3 c0a8 0091  E..8..@.@.......
...
```

パッチを確認したい場合はtcpdumpプロジェクトのcvsで確認します。anonymous CVSが用意されており、チェックアウト方法も書いてあります。

```
# cvs -d :pserver:tcpdump@cvs.tcpdump.org:/tcpdump/master login
Logging in to :pserver:tcpdump@cvs.tcpdump.org:2401/tcpdump/master
CVS password:                                  <- anoncvs と入力
# cvs -d :pserver:tcpdump@cvs.tcpdump.org:/tcpdump/master checkout tcpdump
cvs checkout: Updating tcpdump
...
```

logやdiffの内容から対象となるパッチを確認します。今回のバグに対する修正は以下のコマンドで確認できます。

```
# cd tcpdump/
# cvs log print-sctp.c
# cvs diff -N -u -p -r 1.17 -r1.18 print-sctp.c
```

まとめ

本Hackではtcpdumpを例にユーザアプリケーションの無限ループを解析しました。範囲を限定し適切な箇所にブレークポイントを設定することで簡単に解決できます。

今回は大量に同じメッセージが出力されたため、すぐに無限ループと気づきやすい例でしたが、メッセージを出力せずに黙ったままプロンプトに返らないような現象もあります。そのときはtopコマンドやvmstatコマンドでCPU使用率などを確認してください。

参考文献

- tcpdump/libpcapプロジェクト
  http://www.tcpdump.org/

- [RFC2960] Stream Control Transmission Protocol
  http://www.ietf.org/rfc/rfc2960.txt

—— Naohiro Ooiwa

SCTP プロトコルについて

SCTP は、SS7 を伝送する専用のプロトコルではなく、通常のプロトコルとしても使用できます。TCP と比較すると以下のような特長があります。

- SCTP は TCP スタイル、UDP スタイルと 2 種類あり、信頼性のある通信と転送効率を優先する通信を選択できるようになっています。
- UDP スタイルでは同じポート番号で複数コネクションを 1 つのソケットで管理できます。
- ソケットで複数の IP アドレスを管理することができ、また動的に IP アドレス（パス）の追加 / 削除が可能で、マルチホーム環境に対応しています。メインで通信する IP アドレス以外はプロトコル内に実装されているハートビートで監視されており、メインの IP アドレスが通信できなくなると、自動で他のパスに切り替えます。
- コネクションが切断されたなどのイベントが非同期で SCTP プロトコルからユーザアプリケーションに通知されます。TCP では導通確認のためパケットを送信しなくてはなりませんが、SCTP ではそのような監視は不要になります。

5章 実践カーネルデバッグ
Hack #33-42

この章では、カーネル障害のデバッグ方法について記しています。カーネルパニック（NULLポインタ参照、リスト破壊、レースコンディション）、カーネルストール（無限ループ、スピンロック、セマフォ、リアルタイムプロセス）、動作のスローダウン、CPU負荷が高くなる不具合についてデバッグ方法を記しています。

HACK #33 カーネルパニック（NULLポインタ参照編）

実際にあったカーネルがOopsする問題を取り上げてカーネルデバッグ手法を説明します。

LTP（Linux Test Project）のテストプログラムを使ってIPv6のネットワーク負荷試験を行っていたときに、カーネルパニックが発生しました。その時コンソールに次のようなメッセージが出ていました。問題のあったOSは、カーネルバージョン2.6.9をベースにしたディストリビューションです。

```
Unable to handle kernel NULL pointer dereference at 0000000000000160 RIP:<ffffffffa0130ac2>{:ipv6:icmpv6_
send+1235}
PML4 226fd067 PGD 0
Oops: 0000 [1] SMP
CPU 0
Modules linked in: ah6 deflate twofish serpent aes blowfish des sha256 crypto_null af_key i2c_dev i2c_
core 8021q md5 ipv6 ide_dump scsi_dump diskdump zlib_deflate dm_mirror dm_multipath dm_mod button battery
ac joydev uhci_hcd ehci_hcd hw_random tg3 e100 mii floppy ext3 jbd ata_piix libata sd_mod scsi_mod
Pid: 6574, comm: ping6 Not tainted 2.6.9-prep
RIP: 0010:[<ffffffffa0130ac2>] <ffffffffa0130ac2>{:ipv6:icmpv6_send+1235}
RSP: 0018:000001003351f3918  EFLAGS: 00010216
RAX: ffffffffa0158eb0 RBX: 0000000000000000 RCX: 0000000000000001
RDX: 000000000000060c RSI: 00000100280848 7f RDI: ffffffffa0158eb0
RBP: 00000000000004d0 R08: 00000100351f37d8 R09: 0000000000000002
R10: 0000000000000000 R11: 0100000000000000 R12: 0000010032cbd180
```

```
R13: 0000000000000040 R14: 000001001f76a058 R15: 0000010026a72c00
FS:  0000002a959b5e20(0000) GS:ffffffff8050cf00(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 000000008005003b
CR2: 0000000000000160 CR3: 0000000000101000 CR4: 00000000000006e0
Process ping6 (pid: 6574, threadinfo 00000100351f2000, task 000001003e4a2030)
Stack: 00000000000004d0 0000000000000202 0000000000000000 0000010026a72f58
       0000000200000000 00000100370c3d00 9c05000000000002 003a000000000007c
       0000010032cbd180 0000010000000000
Call Trace:<ffffffffa013f56f>{:ipv6:xfrm6_output+135}
       <ffffffff802cfcdd>{ip_generic_getfrag+66}
       <ffffffffa011de71>{:ipv6:ip6_push_pending_frames+798}
       <ffffffffa012fd00>{:ipv6:rawv6_sendmsg+2324}
       <ffffffffa0048106>{:jbd:journal_dirty_metadata+391}
       <ffffffff802a9883>{sock_sendmsg+271}
       <ffffffff801381c8>{release_console_sem+369}
       <ffffffff80238bc8>{do_con_write+7903}
       <ffffffff8022f2b6>{vt_ioctl+61}
       <ffffffff80227a41>{tty_ldisc_try+60}
       <ffffffff8013560e>{autoremove_wake_function+0}
       <ffffffff802ab1f3>{sys_sendmsg+463}
       <ffffffff8012370f>{do_page_fault+575}
       <ffffffff802ac381>{release_sock+16}
       <ffffffff8018baf8>{sys_ioctl+924}
       <ffffffff8011026a>{system_call+126}

Code: 48 8b 9b 60 01 00 00 48 85 db 74 07 f0 ff 83 d0 00 00 00 48
RIP <ffffffffa0130ac2>{:ipv6:icmpv6_send+1235} RSP <00000100351f3918>
CR2: 0000000000000160
```

このメッセージの先頭行からNULLポインタアクセス（正確には0x160というアドレス）によるOopsが発生していることがわかります。Oopsメッセージの読み方については「Oopsメッセージの読み方」**[HACK #15]**を参照してください。

再現プログラムを作る

調査を進めていくと、LTPに含まれる特定のテストを実行すると100%の確率でOopsが発生することがわかりました。そして、そのテスト内容をさらに単純化して、次のようなスクリプトで再現させることができました。

```
$ cat reproducer.sh
#!/bin/sh
```

```
IFACE=$1
CONTENT="add fd00:1:1:1::2 fd00:1:1:1::1 ah 1000 -m tunnel \
        -A hmac-sha1 \"beef_fish_pork_salad\" ; \
        spdadd fd00:1:1:1::2 fd00:1:1:1::1 any -P out \
        ipsec ah/tunnel/fd00:1:1:1::2-fd00:1:1:1::1/use ; \
        spdadd fd00:1:1:1::1 fd00:1:1:1::2 any -P in \
        ipsec ah/tunnel/fd00:1:1:1::1-fd00:1:1:1::2/use ;"
ip -f inet6 addr add fd00:1:1:1::2/64 dev $IFACE
echo $CONTENT | setkey -c
sleep 5
ping6 -I $IFACE fd00:1:1:1::1 -c 1 -s 1500
```

IPsec トンネルを作成し、その中に ping6 コマンドを使って IPsec トンネルの MTU より大きなパケットを送信するというスクリプトです。次のようにネットワークインタフェースの名前をオプションとして指定し、このスクリプトを実行すると Oops が発生します。スクリプト実行にはスーパーユーザ権限が必要です。

```
$ sudo ./reproducer.sh eth1
```

カーネルダンプからどこで NULL ポインタアクセスが発生したか確認する

この問題では Oops が発生するので、カーネルクラッシュダンプを採取してデバッグすることにしました。まず冒頭の Oops メッセージから icmpv6_send() 関数のオフセット 1235 バイトの位置を実行時にパニックしていることがわかります。まずは当該関数を逆アセンブルしてみて、何が起こっていたのか確認してみます。

```
crash> disas icmpv6_send
No symbol "icmpv6_send" in current context.
```

何やら crash コマンドからエラーが出てしまいました。シンボルがないと怒られています。icmpv6_send() は ipv6.ko というローダブルモジュール内に定義されています。これを読み込んであげないと表示できません。

```
crash> mod -s ipv6
     MODULE       NAME            SIZE  OBJECT FILE
ffffffffa015d180  ipv6          284512  /lib/modules/2.6.9-prep/kernel/net/ipv6/ipv6.ko
```

これで icmpv6_send() を見ることができます。

```
crash> disas icmpv6_send
Dump of assembler code for function icmpv6_send:
0xffffffffa01305ef <icmpv6_send+0>:     push   %r15
0xffffffffa01305f1 <icmpv6_send+2>:     mov    $0xffffffff80543930,%rax
…
0xffffffffa0130a91 <icmpv6_send+1186>:  callq  0xffffffff802ba11e <net_ratelimit> ———①
0xffffffffa0130a96 <icmpv6_send+1191>:  test   %eax,%eax
0xffffffffa0130a98 <icmpv6_send+1193>:  je     0xffffffffa0130bef <icmpv6_send+1536>
0xffffffffa0130a9e <icmpv6_send+1199>:  mov    $0xffffffffa01413ba,%rdi
0xffffffffa0130aa5 <icmpv6_send+1206>:  xor    %eax,%eax
0xffffffffa0130aa7 <icmpv6_send+1208>:  callq  0xffffffff80138413 <printk>
0xffffffffa0130aac <icmpv6_send+1213>:  jmpq   0xffffffffa0130bef <icmpv6_send+1536>
0xffffffffa0130ab1 <icmpv6_send+1218>:  mov    0x30(%r12),%rbx
0xffffffffa0130ab6 <icmpv6_send+1223>:  mov    $0xffffffffa0158eb0,%rdi ————————②
0xffffffffa0130abd <icmpv6_send+1230>:  callq  0xffffffff8030e7d0 <_read_lock> ———③
0xffffffffa0130ac2 <icmpv6_send+1235>:  mov    0x160(%rbx),%rbx ————————————④
0xffffffffa0130ac9 <icmpv6_send+1242>:  test   %rbx,%rbx
0xffffffffa0130acc <icmpv6_send+1245>:  je     0xffffffffa0130ad5 <icmpv6_send+1254>
0xffffffffa0130ace <icmpv6_send+1247>:  lock incl 0xd0(%rbx)
…
```

問題箇所は icmpv6_send+1235 なので④の箇所です。これは RBX レジスタで表されるアドレスからオフセット 0x160 バイトの位置にある内容を RBX レジスタに格納する命令です。Oops メッセージ内のレジスタダンプから RBX が 0 であるため、④の命令を C 言語のように表すと次のようになります。

```
RBX = *(0x160 + 0x0)
```

これが Oops した原因です。では、この箇所がカーネルソースコード上、どの部分に対応しているのか調べましょう。ポイントとなるのは①でコールされている関数 net_ratelimit() です。ここでは net_ratelimit() は LIMIT_NETDEBUG() というマクロに置き換えられており、icmpv6_send() 内で LIMIT_DETDEBUG() がコールされるのは 3 ヶ所あります。

[include/net/sock.h]

```
#define LIMIT_NETDEBUG(x) do { if (net_ratelimit()) { x; } } while(0)
```

net_ratelimit() はカーネル内ネットワークスタックのコードでよく用いられる関数で、次のように printk() で情報表示させる時にセットで使われます。

```
if (net_ratelimit())
   printk(...);
```

printk() でメッセージを表示させるのはそれなりにコストがかかる処理です。ネットワークコードの場合、外部の悪意あるユーザから DoS 攻撃の対象になりえるため、毎回必ず printk() するのではなく、ある一定の頻度以下に抑える仕組みがこの net_ratelimit() です。カーネルネットワークコードの場合、逆アセンブルされたコードとの対応を見る際にはこの net_ratelimit() を目印として使うと、だいぶ見やすくなります。

もう 1 つ注目したいポイントは③にある _read_lock() です。直前の②で read_lock() に渡す引数を設定しています。これが何なのかを crash コマンドで調べてみます。

```
crash> sym 0xffffffffa0158eb0
ffffffffa0158eb0 (d) addrconf_lock
```

つまり、③の箇所は read_lock(&addrconf_lock); を実行しているということです。

以上をまとめると LIMIT_NETDEBUG() コールのすぐ近くで read_lock(&addrconf_lock) をしている箇所ということになり、次の場所であることがわかりました。

```
[net/ipv6/icmp.c]
void icmpv6_send(struct sk_buff *skb, int type, int code, __u32 info,
                 struct net_device *dev)
{
…
        if (len < 0) {
                LIMIT_NETDEBUG( ―――――――――――――――――――――――――――⑤
                        printk(KERN_DEBUG "icmp: len problem\n"));
                goto out_dst_release;
        }

        idev = in6_dev_get(skb->dev);
…
```

逆アセンブル結果の①に対応する部分が、⑤です。一方、in6_dev_get() を見てみると探していた read_lock() がありました。inline 宣言されているので、逆アセンブル上では関数コー

ルとして表れず、icmpv6_send() 関数内にインライン展開されています。

```
[include/net/addrconf.h]
  static inline struct inet6_dev *
  in6_dev_get(struct net_device *dev)
  {
          struct inet6_dev *idev = NULL;
          read_lock(&addrconf_lock); ──────────────────────⑥
          idev = dev->ip6_ptr; ────────────────────────────⑦
          if (idev)
                  atomic_inc(&idev->refcnt);
          read_unlock(&addrconf_lock);
          return idev;
  }
```

逆アセンブル結果の③に対応する部分が⑥であるとわかります。また⑦で参照されている net_device 構造体の ip6_ptr メンバのオフセットを調べてみます。

```
  crash> hex
  output radix: 16 (hex)
  crash> struct -o net_device.ip6_ptr
  struct net_device {
    [0x160] void *ip6_ptr;
  }
```

オフセット 0x160 バイトでした。ビンゴです。Oops が発生した箇所、つまり逆アセンブル結果の④は、この⑦のコードに対応していることがわかりました。

```
   mov    0x160(%rbx),%rbx
```

このオフセット 0x160 は net_device 構造体の ip6_ptr メンバへのオフセットであるため、RBX が指すアドレスは in6_dev_get() 関数の引数である、dev のようです。Oops メッセージのレジスタダンプでは RBX が 0 だったことから、問題発生時の状況は dev ポインタが NULL であったとわかります。in6_dev_get() には skb->dev を引数として渡しているため、skb->dev が NULL であったことが直接の原因のようです。

ソースコードから処理の内容を調べる

さらにソーストレースを行い、問題発生時のコールフローは図 5-1 のようになっていたことがわかりました。

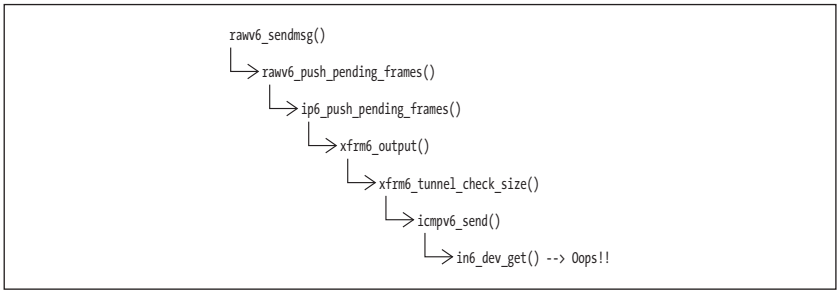

図 5-1　Oops 発生までのコールフロー

xfrm6_tunnel_check_size() 関数を見るとよくわかりますが、IPsec トンネルの MTU を超えたパケットを送信しようとしたため、ICMPv6 のパケット過大メッセージを返す処理でした。

```
[net/ipv6/xfrm6_output.c]
  static int xfrm6_tunnel_check_size(struct sk_buff *skb)
  {
  …
          if (skb->len > mtu) {
          /* IPsec トンネルの MTU を超えたパケットなら、ICMP エラーを返信 */
                  icmpv6_send(skb, ICMPV6_PKT_TOOBIG, 0, mtu, skb->dev);
                  ret = -EMSGSIZE;
          }
  …
  }
```

まとめると、ping6 コマンドの延長で icmpv6_send() がコールされました。しかし icmpv6_send() が送信しようとしたデバイスを表す skb->dev が NULL だったため、Oops が発生していました。もう少し説明すると、このタイミングでは IPsec トンネルにパケットを送信しようとして、ソケットバッファ（skb）を組み立てている最中です。そのためまだ skb->dev が設定されておらず、NULL となっている状態でした。現状のコードでは、今回のような MTU を超えたために送信元へエラーを返すというコードパスに対して、考慮もれがあったということです。

コミュニティの履歴をチェックする

Linux カーネルのようなオープンソースソフトウェアの場合、コミュニティですでに修正されているケースが多々あります。そこで Linus Torvals 氏の git ツリーから関連する

修正を検索したところ、次のパッチを発見しました。コミュニティではすでに発見された問題で、カーネル 2.6.12 の時点で修正されていました。

```
commit 180e42503300629692b513daeb55a6bb0b51500c
Author: Herbert Xu <herbert@gondor.apana.org.au>
Date:   Mon May 23 13:11:07 2005 -0700

    [IPV6]: Fix xfrm tunnel oops with large packets

    Signed-off-by: Herbert Xu <herbert@gondor.apana.org.au>
    Acked-by: Hideaki YOSHIFUJI <yoshfuji@linux-ipv6.org>
    Signed-off-by: David S. Miller <davem@davemloft.net>

diff --git a/net/ipv6/xfrm6_output.c b/net/ipv6/xfrm6_output.c
index 601a148..6b98677 100644
--- a/net/ipv6/xfrm6_output.c
+++ b/net/ipv6/xfrm6_output.c
@@ -84,6 +84,7 @@ static int xfrm6_tunnel_check_size(struct sk_buff *skb)
                        mtu = IPV6_MIN_MTU;

        if (skb->len > mtu) {
+                       skb->dev = dst->dev;
                        icmpv6_send(skb, ICMPV6_PKT_TOOBIG, 0, mtu, skb->dev);
                        ret = -EMSGSIZE;
        }
```

まとめ

実際に発生した NULL ポインタアクセスにより Oops する典型的な例を取り上げ、カーネルクラッシュダンプからデバッグする手法を説明しました。本 Hack で行ったように、再現プログラムを単純化することはとても重要です。単純化すればするほど解析する時の調査範囲が狭くなり、解決が早くなることが多いからです。

参考文献

- Linux Test Project
  http://ltp.sourceforge.net/
- Linus Torvals 氏の git ツリー
  http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary

—— Toyo Abe

HACK #34 カーネルパニック（リスト破壊編）

テストプログラムを使ったリスト破壊のデバッグ方法を紹介します。

以前に体験したリスト破壊を再現させるテストプログラム（TP）を作成しました。このバグはSMP環境で発生します。本Hackではこれを元にリスト破壊を説明します。カーネルは2.6.9系のディストリビューションです。

カーネルのリスト

再現TPに使用するリストはカーネルで実装されている双方向リストlist_headです。これは最もシンプルで一般的なリストです。小規模で単純な管理に適しており、ファイルシステム、メモリ管理、ネットワークなどいたるところで使われています。

include/linux/list.hにリスト操作の関数とマクロが定義されています。再現TPはこのファイルをインクルードしています。**表5-1**に代表的な関数とマクロを示します。

表5-1　リスト操作の関数とマクロ

リスト操作関数とマクロ	内容
list_add	エントリを先頭に追加する
list_del	エントリをリストから外す
list_empty	リストが空か確認する
list_entry	エントリを取得する（リストのポインタからエントリの構造体を取得）
list_for_each	リストのエントリを走査する

再現TPの内容

再現TPはカーネルモジュールです。以下のコードはその一部です。リストの操作をする3つのカーネルスレッドがモジュールのinsmodで起動させます†。

1つ目のスレッドlist_add_thread()は10000のエントリをリストに追加します。バグを再現させるためにはある程度多くのエントリを追加させる必要があったため10000としています。

```
#include <linux/list.h>

static int list_add_thread(void *data)
{
    int i;
```

† TPは本書のサポートページ（http://www.oreilly.co.jp/books/9784873114040/）からダウンロードしてください。

```
    do {
        spin_lock(&trouble.lock); ──────────────────────────────①
        for (i = 0; i < 10000; i++) {          /* エントリを10000作成 */
            struct k_entry *entry;

            entry = kmalloc(sizeof(struct k_entry),GFP_ATOMIC);
            INIT_LIST_HEAD(&entry->list);      /* リストの初期化 */
            list_add(&entry->list, &trouble.list);   /* エントリをtroubleリストに追加 */
        }
        spin_unlock(&trouble.lock); ────────────────────────────②
        msleep(200);
    } while (!kthread_should_stop());

    return 0;
}
```

2つ目のスレッドlist_release_thread()はリストが空でなければすべてのエントリを削除します。

```
static int list_release_thread(void *data)
{
    do {
        spin_lock(&trouble.lock); ──────────────────────────────③
        while (!list_empty(&trouble.list)){   /* リストが空か確認 */
            struct k_entry *entry;

            entry = list_entry(trouble.list.next, struct k_entry, list);  /* リストの先頭からエントリを
取得 */
            list_del(&entry->list);           /* リストからエントリを外す */
            kfree(entry);                     /* エントリ解放 */
        }
        spin_unlock(&trouble.lock); ────────────────────────────④
        msleep(100);
    } while (!kthread_should_stop());

    return 0;
}
```

3つ目のスレッドlist_del_thread()はリストが空でなければリストのエントリを1つだけ削除します。

```
static int list_del_thread(void *data)
{
    do {
        if (!list_empty(&trouble.list)){      /* リストが空か確認 */ ―――⑤
            struct k_entry *entry;

            spin_lock(&trouble.lock); ―――――――――――――――――――⑥
            entry = list_entry(trouble.list.next, struct k_entry, list);
            list_del(&entry->list);
            kfree(entry);
            spin_unlock(&trouble.lock); ―――――――――――――――――⑦
        }
        msleep(1);
    } while (!kthread_should_stop());

    return 0;
}
```

kthread_should_stop() は rmmod できるようにするためです。

spin_lock() でリストを保護しており、問題がないように見えますが、このモジュールを insmod するとパニックしてダンプが採取されます。

ダンプ解析

以下は採取されたダンプのバックトレースです。

```
Unable to handle kernel paging request at 0000000000100108 RIP: ―――――⑧
<ffffffff80162bb6>{kfree+168}
PML4 0
Oops: 0000 [1] SMP
CPU 1
Modules linked in: trouble_list · · ·    /* 再現 TP のモジュール */
...
Pid: 4825, comm: list_del Not tainted 2.6.9
RIP: 0010:[<ffffffff80162bb6>] <ffffffff80162bb6>{kfree+168}
RSP: 0018:000001007c753ee8  EFLAGS: 00010002
RAX: 0000000000000001 RBX: 0000000000000000 RCX: 000001000000c000
RDX: 0000000000000000 RSI: 0000000000000008 RDI: 0000000000100100
RBP: ffffffffa0196d40 R08: 0000000000000003 R09: 0000000000000040
R10: 0000000000000001 R11: 0000000000000001 R12: 0000010061ed7e18
```

```
R13: 00000000ffffffc R14: 0000010061ed7e08 R15: ffffffff8014b4f8
FS:  0000000000000000(0000) GS:ffffffff8050d300(0000) knlGS:0000000000000000
CS:  0010 DS: 0018 ES: 0018 CR0: 00000008005003b
CR2: 0000000000100108 CR3: 00000000016e8000 CR4: 00000000000006e0
Process list_del (pid: 4825, threadinfo 000001007c752000, task 0000010059f9a7f0)
Stack: 0000000000000006 0000000000000282 0000000000000000 ffffffffa01960e0
       0000010061ed7e18 ffffffffa0196123 0000000000000000 ffffffff8014b4cf
       ffffffffffffffff 0000010061ed7e08
Call Trace:<ffffffffa01960e0>{:trouble_list:list_del_thread+0}
       <ffffffffa0196123>{:trouble_list:list_del_thread+67}
       <ffffffff8014b4cf>{kthread+200} <ffffffff80110f53>{child_rip+8}
       <ffffffff8014b4f8>{keventd_create_kthread+0} <ffffffff8014b407>{kthread+0}
       <ffffffff80110f4b>{child_rip+0}
```

ここで通常であれば逆アセンブルの結果から kfree()+168 を確認し、どの変数を参照しようとしてパニックしたのか調べます。しかしここではアクセスしたポインタのアドレス⑧を見ます。少し変わったアドレスで、100108（0x100100+0x8）となっています。これは LIST_POISON です。

LIST_POISON

Linux のカーネル 2.6 系では list_del() でエントリの next、prev メンバに NULL ではなく、LIST_POISON を入れます。

[include/linux/list.h]

```
#define LIST_POISON1  ((void *) 0x00100100)
#define LIST_POISON2  ((void *) 0x00200200)

static inline void list_del(struct list_head *entry)
{
        __list_del(entry->prev, entry->next);
        entry->next = LIST_POISON1;
        entry->prev = LIST_POISON2;
}
```

list_del() したエントリに間違ってアクセスしたときや、初期化をしないで使用したことを検出（パニック）するため意図的に汚しています。

⑧が 100108 となっているのは kfree() 内で trouble.list.next(0x100100) のオフセット 0x8 を見ているためです。

リスト破壊の仕組み

このバグの仕組みを図 5-2 に示します。

リストが空の状態で list_entry() により取得したエントリは 0x100100 になってしまい、それを kmalloc() した通常のエントリとして扱ってしまうため、不正アクセスになります。

通常のリストと今回のリストを図 5-3 に示します。

list_release_thread() の処理が終わるとリストは空になります。パニックしたとき、つまりリストが空の状態で list_del_thread() が動いたときには、空のリストから無理にエントリを取得して list_del() をするため、リストの next、prev が LIST_POISON で汚れます。

以下はこのダンプでグローバルの trouble リストを参照しようとした結果です。同じように不正アクセスになります。

```
crash> struct list_head trouble
struct list_head {
  next = 0x100100,
  prev = 0x200200
}
```

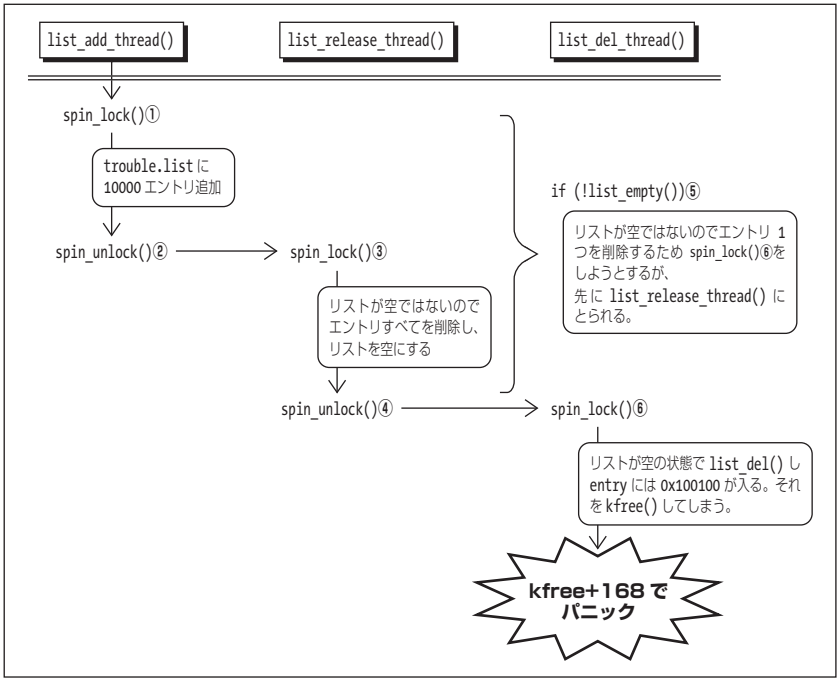

図 5-2　リスト破壊の仕組み

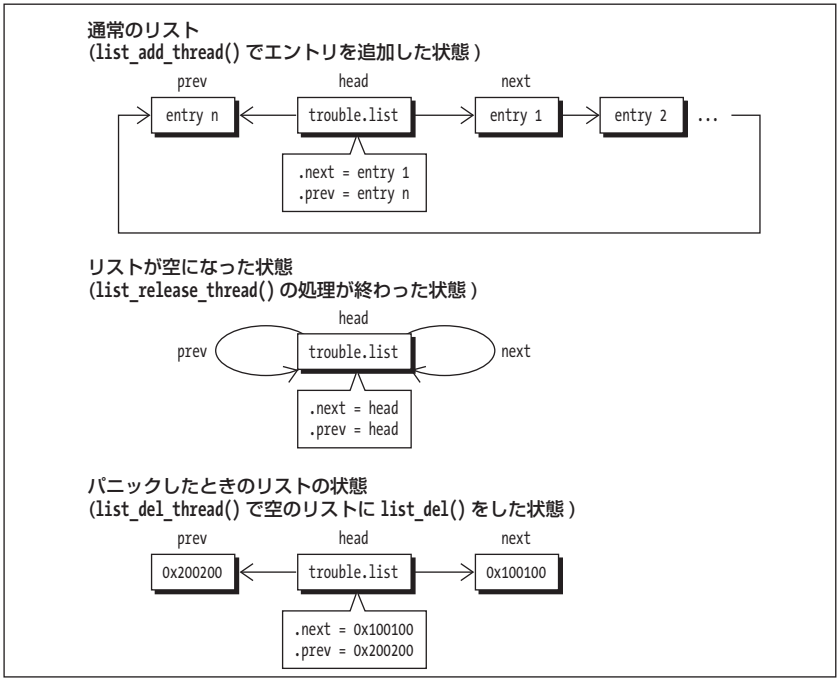

図 5-3 リスト破壊の仕組み（2）

```
crash> list trouble
ffffffffa0196d40
100100
list: invalid kernel virtual address: 100100  type: "list entry"
crash>
```

Debug memory allocations

今回は再現 TP を使用したため、これまで説明したように原因を明確にできましたが、実際のバグは難しいです。list_del() によるリスト破壊は LIST_POISON でわかりますが、それ以外にアロケートしたメモリ領域や二重解放をチェックする Debug memory allocations というデバッグ機能があります。kmalloc()/kfree() でアロケートしたメモリ（スラブキャッシュ）をチェックします。

この機能を使用するには make menuconfig で Kernel hacking -> Debug memory allocations(CONFIG_DEBUG_SLAB=y）を有効にして、再コンパイルします。

以下は Debug memory allocations を有効にしたカーネルで再現 TP を実行した結果です。BUG() によりパニックしますが、その直前にメッセージが出力されます。

```
kfree_debugcheck: bad ptr ffffffffa0196d40h.        ―――⑨
---------- [cut here ] --------- [please bite here ] ---------
Kernel BUG at slab:1862                    /* kernel/slab.cの1862行目の意味 */
invalid operand: 0000 [1] SMP
CPU 0
Modules linked in: trouble_list ...
...
Pid: 4380, comm: list_del Not tainted 2.6.9-42.28AXinode
RIP: 0010:[<ffffffff80162b3a>] <ffffffff80162b3a>{kfree_debugcheck+436}
RSP: 0018:0000010077eafed8  EFLAGS: 00010012
RAX: 0000000000000030 RBX: ffffffffa0196d40 RCX: ffffffff803e8d68
...
Call Trace:<ffffffff80163833>{kfree+29} <ffffffffa01960e0>{:trouble_list:list_de
l_thread+0}
       <ffffffffa0196132>{:trouble_list:list_del_thread+82}
       <ffffffff8014b4cf>{kthread+200} <ffffffff80110f53>{child_rip+8}
       <ffffffff8014b4f8>{keventd_create_kthread+0} <ffffffff8014b407>{kthread+0
}
       <ffffffff80110f4b>{child_rip+0}
...
```

kfree_debugcheck() は CONFIG_DEBUG_SLAB=y のときだけ kfree() の中で呼ばれるメモリチェックの関数です。⑨のポインタは trouble 変数のアドレスです。このメッセージはスラブキャッシュではない（trouble はグローバル変数で kmalloc()、または kmem_cache_alloc() で確保されたメモリではない）メモリ領域にアクセスしたことを意味しています。

リスト破壊の修正

本 Hack で扱ったリスト破壊の原因は排他処理の仕方にあります。リストが空かどうかの確認も spin_lock() 内で行うべきです。以下は修正パッチです。

```
--- trouble_list.c      2009-01-14 23:18:27.000000000 +0900
+++ trouble_list.c.new  2009-01-14 23:19:59.000000000 +0900
@@ -63,15 +63,15 @@ static int list_release_thread(void *dat
 static int list_del_thread(void *data)
 {
        do {
+               spin_lock(&trouble.lock);
                if (!list_empty(&trouble.list)){
                        struct k_entry *entry;
```

```
-                       spin_lock(&trouble.lock);
                        entry = list_entry(trouble.list.next, struct k_entry, list);
                        list_del(&entry->list);
                        kfree(entry);
-                       spin_unlock(&trouble.lock);
                }
+               spin_unlock(&trouble.lock);
                msleep(1);
        } while (!kthread_should_stop());
```

このパッチを適用するとパニックせず、正常に動作します。

まとめ

本Hackではリスト破壊のパニックを紹介しました。0x100100かまたは0x200200があれば、リスト破壊と予想できます。

LIST_POISONはlist_del()で設定されますが、list_del_init()はリストのエントリをNULLで初期化してしまうため、LIST_POISONは入りません。そのため万能ではありませんが、Linuxカーネルのデバッグにおける1つのテクニックです。

また本HackではDebug memory allocationsを紹介しました。この機能を使えばLIST_POISONなどで汚されていないメモリ領域であっても、二重解放やスラブキャッシュの不正な扱いを検出することができます。

類似の機能でメモリページをチェックするPage alloc debugging（CONFIG_DEBUG_PAGEALLOC）もあります。

（今回のリスト破壊は、2.6.15カーネルまであったkernel/posix-timers.cのclock_was_set()で実際に発生したリスト破壊を元にしています。）

—— Naohiro Ooiwa

HACK #35 カーネルパニック（レースコンディション編）

カーネルのソースコードからOSの動作を確認して、レースコンディションによるバグの再現TPを作ります。

ダンプからある程度原因がわかってパッチを作成しても本当に正しく修正されたか、他に考慮すべき点があるか確認したいときや、カーネルソースを見てバグのような箇所があっても実際に問題があるのか確認するときには、再現させる必要があります。

ただし、まれにしか発生しないバグの場合はとても困難です。カーネルでもユーザプログラムでも内部でバグの発生するルートを通り、さらにタイミングを合わせる必要があります。

本Hackはカーネル2.6.9系のディストリビューションでinodeのコードにバグの可能

性を見つけました。これを例にバグの修正までを説明します。使用したマシンはメモリが2GB、スワップが2GBです。

修正までのフロー

バグを修正するまでの主な流れを**図5-4**に示します。

本Hackではソースコードからバグの可能性を見つけた場合で、以下の流れに沿って説明します。

1. バグの可能性を発見
2. ソースコード調査
3. ルート1の確認

3-1 WARN_ON()による確認

4 ルート2の確認

4-1 パラメータvfs_cache_pressureを調整する

5. バグの再現

5-1 通常のカーネルで再現試験

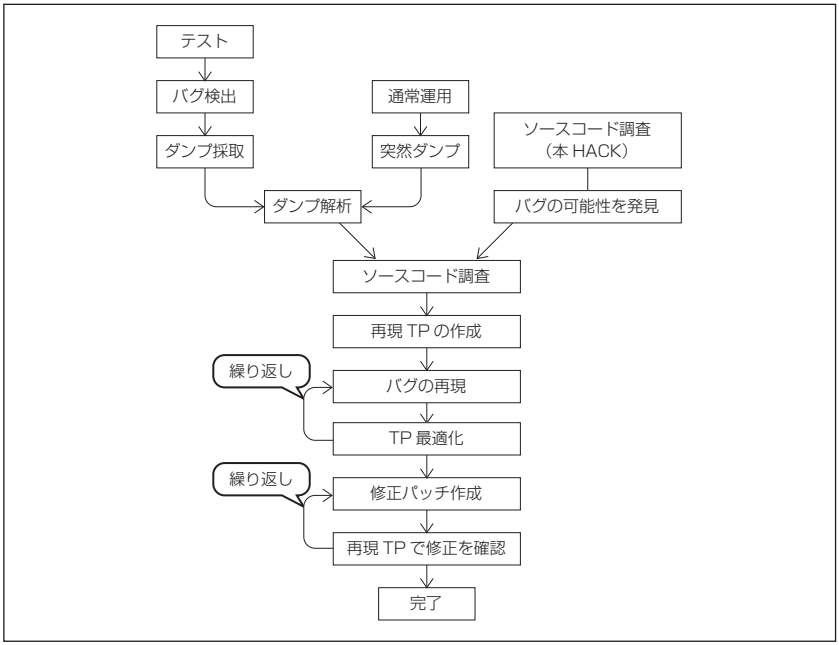

図5-4 バグを修正するまでの流れ

5-2 mdelay() で再現確率を上げる
6. コミュニティの履歴を確認

1. バグの可能性を発見

Linux カーネルには inode というものがあります。inode とは通常のファイルやシンボリックリンク、ディレクトリの情報を管理するキャッシュです。

今回見つけたバグの可能性は fs/inode.c の generic_forget_inode() にありました。一時的に inode_lock をアンロックしています。

```
static void generic_forget_inode(struct inode *inode)
{                                                        iput() の atomic_dec_and_lock() で
    struct super_block *sb = inode->i_sb;                ロックは取得されている

    if (!hlist_unhashed(&inode->i_hash)) {
        if (!(inode->i_state & (I_DIRTY|I_LOCK)))
            list_move(&inode->i_list, &inode_unused);
        inodes_stat.nr_unused++;
        spin_unlock(&inode_lock);                        ここでアンロック
        if (!sb || (sb->s_flags & MS_ACTIVE))
            return;
        write_inode_now(inode, 1);
        spin_lock(&inode_lock);                          ロック
        inodes_stat.nr_unused--;
        hlist_del_init(&inode->i_hash);
    }
    list_del_init(&inode->i_list);
    inode->i_state|=I_FREEING;                           I_FREEING フラグを設定
    inodes_stat.nr_inodes--;                             nr_inodes はアトミックではないためロックの
                                                         中で減らす
    spin_unlock(&inode_lock);                            アンロック
    if (inode->i_data.nrpages)
        truncate_inode_pages(&inode->i_data, 0);
    clear_inode(inode);
    destroy_inode(inode);
}
```

このように途中でアンロックをすると、他の CPU から inode_lock を横取りされる可能性があります。CONFIG_PREEMP を有効にしている場合は inode_lock の取得待ちをしているプロセスがいれば自ら切り替えるため、さらに可能性が高くなります。

2. ソースコード調査

ソースコードを確認して競合の仕組みを図 5-5 にまとめます。

プロセス X と Y で競合するとパニックまたは何らかの不具合が発生すると予想できます。この仮説を証明するために再現 TP を作ってみます。

まずはソースコードを調査して、ある程度仮説が成り立つか確認します。

3. ルート 1 の確認

図 5-5 のプロセス X を考えてみます。generic_forget_inode() は iput() から呼ばれます。iput() はいろんなルートから呼ばれます。数が多いため全部を確認するには時間がかかります。そのため iput() は何をする関数なのか調べます。iput() は解放することを示すフラグ（I_FREEING）を inode に付け、解放できるように管理リストからその inode を外します。そのあとは clear_inode()、destroy_inode() で inode を削除します。

inode は同じファイルへのアクセスを高速化させるためのキャッシュです。そのためファイルの作成・削除を繰り返せば、いずれ inode（最初に作られアクセスしなくなったファイル）は iput() により解放されるはずです。今回は WARN_ON() を使ってルート 1 を通るか確認してみます。

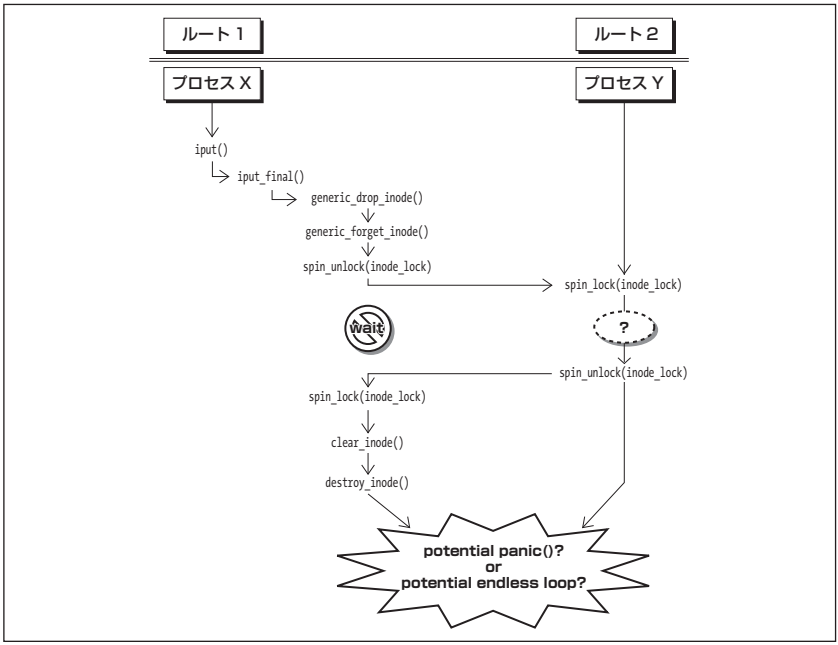

図 5-5　競合の仕組み

3-1. WARN_ON() による確認

WARN_ON() はカーネル内部のマクロで、注意を促すためにスタックトレースを表示します。本来の用途とは違いますが、今回は関数コールのルートを確認するために使います。メッセージ出力を視覚で感覚的に確認できるのも利点です。タイミングが合ったときだけまれにメッセージが出力されるのか、それとも頻繁に通るルートなのか確認できます。

それでは以下のパッチを適用してカーネルを再コンパイルします。

```
--- fs/inode.c.orig     2005-06-18 04:48:29.000000000 +0900
+++ fs/inode.c  2008-10-13 21:47:40.000000000 +0900
@@ -1035,6 +1036,7 @@ static void generic_forget_inode(struct
                        list_move(&inode->i_list, &inode_unused);
                inodes_stat.nr_unused++;
                spin_unlock(&inode_lock);
+               WARN_ON(1);
                if (!sb || (sb->s_flags & MS_ACTIVE))
                        return;
                write_inode_now(inode, 1);
```

このカーネルで起動し stress で I/O 負荷をかけます。stress については「**OOM Killer の動作と仕組み**」**[HACK #56]** を参照してください。inode はファイルの数に依存するのでファイルサイズは 10MB と小さい値にします。ファイルシステムは ext3 です。

```
# stress --hdd 1 --hdd-bytes 10M
```

コンソール画面には WARN_ON() の出力はありません。つまりルート 1 を通っていません。メモリにゆとりがある状態では新しいファイルを作成しても ext3 の inode が増えるだけです。ext3 の inode 数は以下のコマンドで確認できます。

```
# cat /proc/slabinfo | grep ext3_inode_cache
```

メモリ不足の状態になればキャッシュを持つゆとりがなくなり、inode が解放されるはずです。そこで以下のように stress のオプションでメモリ負荷を追加しました。実行した環境はメモリ 2GB、スワップ 2GB なので、8 プロセスで 500MB ずつ消費しています。

```
# stress --hdd 1 --hdd-bytes 10M --vm 8 --vm-bytes 500M --vm-keep
```

実行したところ以下の 2 パターンの出力がされました。上のコマンドで generic_forget_inode() を通ることがわかります。

```
Badness in generic_forget_inode at fs/inode.c:1038

Call Trace:<ffffffff801932c8>{generic_drop_inode+152} <ffffffff80190446>{prune_dcache+806}
       <ffffffff80190999>{shrink_dcache_memory+20} <ffffffff8016541f>{shrink_slab+188}
       <ffffffff80166705>{try_to_free_pages+348} <ffffffff8015ed0f>{__alloc_pages+527}
       <ffffffff8015dfb2>{get_zeroed_page+26} <ffffffff80167fdd>{pte_alloc_map+47}
       <ffffffff8016a9bc>{handle_mm_fault+226} <ffffffff80123720>{do_page_fault+520}
       <ffffffff8030dc3e>{thread_return+0} <ffffffff8030dc96>{thread_return+88}
       <ffffffff80110d9d>{error_exit+0}

Badness in generic_forget_inode at fs/inode.c:1038

Call Trace:<ffffffff801932c8>{generic_drop_inode+152} <ffffffff80190446>{prune_dcache+806}
       <ffffffff80190999>{shrink_dcache_memory+20} <ffffffff8016541f>{shrink_slab+188}
       <ffffffff80166705>{try_to_free_pages+348} <ffffffff8015ed0f>{__alloc_pages+527}
       <ffffffff8015c4a2>{generic_file_buffered_write+413}
       <ffffffff8015cd19>{__generic_file_aio_write_nolock+731}
       <ffffffff8015cfb7>{generic_file_aio_write_nolock+32}
       <ffffffff8015d081>{generic_file_aio_write+126} <ffffffffa005af01>{:ext3:ext3_file_write+22}
       <ffffffff8017a291>{do_sync_write+173} <ffffffffa0062dfb>{:ext3:ext3_unlink+410}
       <ffffffff80135646>{autoremove_wake_function+0} <ffffffff801898cb>{sys_unlink+313}
       <ffffffff801941ad>{dnotify_parent+34} <ffffffff8017a38c>{vfs_write+207}
       <ffffffff8017a474>{sys_write+69} <ffffffff80110276>{system_call+126}
```

ソースコードを調査すると、他にアンマウント処理の延長でも iput() が呼ばれることがわかりました。アンマウントをするとファイルの sync をして関連するキャッシュはすべて削除するからです。

そこでマウント先に大量のファイルを書き、削除してからアンマウントします。

```
# mount /dev/sda10 /mnt/10
# cd /mnt/10
# stress --hdd 5 --hdd-bytes 10M --hdd-noclean -t 5
# rm -rfv *
# cd -
# umount /mnt/10
```

以下は上の umount コマンドで表示されたコンソール画面です。

```
Badness in generic_forget_inode at fs/inode.c:1038

Call Trace:<ffffffff801932c8>{generic_drop_inode+152} <ffffffffa004d80f>{:jbd:journal_destroy+557}
       <ffffffff80135646>{autoremove_wake_function+0} <ffffffff80135646>{autoremove_wake_function+0}
       <ffffffffa00643dd>{:ext3:ext3_put_super+38} <ffffffffa00643b7>{:ext3:ext3_put_super+0}
       <ffffffff8017f855>{generic_shutdown_super+198} <ffffffff80180677>{kill_block_super+13}
       <ffffffff8017f776>{deactivate_super+95} <ffffffff80195177>{sys_umount+925}
       <ffffffff80182a44>{sys_newstat+17} <ffffffff80110d9d>{error_exit+0}
       <ffffffff80110276>{system_call+126}
```

これで先ほどの stress コマンドと umount を実行するとルート 1 を通ることが確認できました。

4. ルート 2 の確認

ルート 2 を通るプロセス Y はどのようなパターンがあるか調査します。今回プロセス X の iput() は inode_lock をロックして inode_unused リストを処理します。そのため fs/inode.c の prune_icache() に着目しました。prune_icache() も inode_lock と inode_unused リストを処理しているからです。

それではソースコードから prune_icache() のルートを確認します。調査の結果、**図 5-6** のようになっていました。

__alloc_page() は空きメモリがない状態でメモリを確保しようとすると、必要のない inode をまとめて解放しようとします。このときに prune_icache() が動きそうです。これは Linux の構造でファイルの削除やメモリを解放しても、すぐに解放処理は動きません。メモリがなくなったときにはじめて解放させるためです。

もうひとつは kswapd からのルート（**図 5-7**）がありました。

kswapd() なので、メモリを消費してスワップを動作させれば良さそうです。prune_icache()

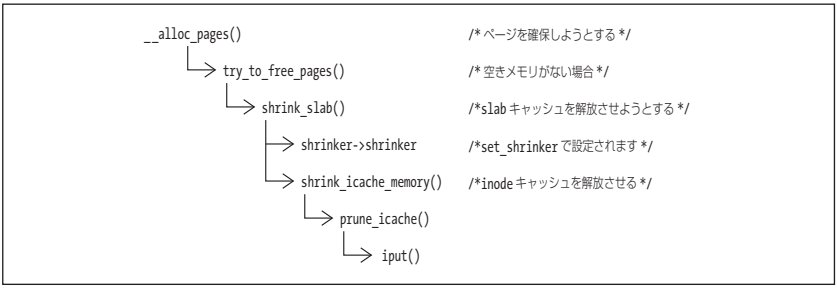

図 5-6　ルート 2 の確認

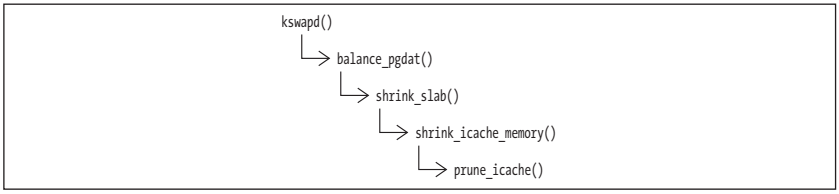

図 5-7　ルート２の確認２

の中では iput() も呼ばれます。排他処理が不完全であればプロセス X と二重解放のようなバグがあるかもしれません。

それではカーネルを再コンパイルして同じように WARN_ON() で prune_icache() が通るか確認をします。generic_forget_inode() の WARN_ON(1) は削除します。以下はパッチです。

```
--- fs/inode.c  2008-10-14 22:47:19.000000000 +0900
+++ fs/inode.c.next     2008-10-14 22:30:04.000000000 +0900
@@ -443,6 +443,7 @@ static void prune_icache(int nr_to_scan)
                if (inode_has_buffers(inode) || inode->i_data.nrpages) {
                        __iget(inode);
                        spin_unlock(&inode_lock);
+                       WARN_ON(1);
                        if (remove_inode_buffers(inode))
                                reap += invalidate_inode_pages(&inode->i_data);
                        iput(inode);
@@ -1035,7 +1036,6 @@ static void generic_forget_inode(struct
                        list_move(&inode->i_list, &inode_unused);
                inodes_stat.nr_unused++;
                spin_unlock(&inode_lock);
-               WARN_ON(1);
                if (!sb || (sb->s_flags & MS_ACTIVE))
                        return;
                write_inode_now(inode, 1);
```

このカーネルでこれからメモリ負荷をかけるわけですが、長時間実行すると OOM Killer により途中で終了させられます。__alloc_pages() はできる限り inode など未使用のキャッシュを解放させようとします。しかし十分なメモリを確保できないと __alloc_pages() 自身が OOM Killer を実行します。そのためメモリ負荷に強弱をつけます。以下のスクリプト vm_stress.sh をバックグラウンドで実行することにします。これは 20 秒間メモリ負荷をかけたあと、2 秒休みます。

```
# cat vm_stress.sh
#!/bin/bash

while [ 0 ] ; do
        stress --vm 8 --vm-bytes 500M --vm-keep --timeout 20
        sleep 2
done
# ./vm_stress.sh &
# stress --hdd 1 --hdd-bytes 10M
```

しかし WARN_ON() のメッセージが出力されません。つまりルート 2 を通っていないことになります。そこでパラメータを調整してみます。

4-1. パラメータ vfs_cache_pressure を調整する

shrink_icache_memory() のコードを見ると sysctl_vfs_cache_pressure の値で return 値が変化することがわかりました。sysctl_vfs_cache_pressure は sysctl のパラメータで /proc/sys/vm/vfs_cache_pressure で変更できます。この値を大きくすると shrink_icache_memory() で解放する inode の目標数を増やすことができます。

通常、shrink_icache_memory() は未使用の inode 数である nr_unused だけ処理します。nr_unused は crash のライブシステムで確認できます。

```
crash> struct -o inodes_stat_t inodes_stat
struct inodes_stat_t {
  nr_inodes = 1658,
  nr_unused = 15,
  dummy = {0, 0, 0, 0, 0}
}
crash>
```

nr_unused はシステムのその時点の値であり、負荷状態では増加し続けます。vfs_cache_pressure を 100 以上に設定すると shrink_icache_memory() が動いた時点で未使用 inode の数が nr_unused であっても、prune_icache() が動いている間に増えた未使用の inode も解放できることになります。そこで vfs_cache_pressure を 5000（50 倍）に設定します。

```
# echo 5000 > /proc/sys/vm/vfs_cache_pressure
```

vm_stress.sh を実行すると WARN_ON() メッセージが出力されました。

```
Badness in prune_icache at fs/inode.c:446

Call Trace:<ffffffff80192759>{shrink_icache_memory+309} <ffffffff8016541f>{shrink_slab+188}
       <ffffffff80166705>{try_to_free_pages+348} <ffffffff8015ed0f>{__alloc_pages+527}
       <ffffffff8016a48e>{do_no_page+651} <ffffffff8016aa4f>{handle_mm_fault+373}
       <ffffffff80123720>{do_page_fault+520} <ffffffff8030dc3e>{thread_return+0}
       <ffffffff8030dc96>{thread_return+88} <ffffffff8016da00>{do_mmap_pgoff+1581}
       <ffffffff801eb99d>{__up_write+20} <ffffffff80110d9d>{error_exit+0}
```

しかし kswapd のパターンが出力されませんので、プロセスを増やしてみます。I/O 負荷テスト用のパーティションを 4 つ用意し、各パーティションで 4 つの stress プロセスにより I/O を行います。

スクリプト inode.sh を実行したところ、kswapd のパターンも確認できました（スクリプト inode.sh は本書のサポートページ（http://www.oreilly.co.jp/books/9784873114040/）からダウンロードしてください）。

```
# ./vm_stress.sh &
# ./inode.sh

Badness in prune_icache at fs/inode.c:446

Call Trace:<ffffffff80192759>{shrink_icache_memory+309}
       <ffffffff8016541f>{shrink_slab+188}
       <ffffffff80166a39>{balance_pgdat+538}
       <ffffffff80166c63>{kswapd+252}
       <ffffffff80135646>{autoremove_wake_function+0}
       <ffffffff80135646>{autoremove_wake_function+0}
       <ffffffff80110f53>{child_rip+8}
       <ffffffff80166b67>{kswapd+0}
```

これで stress のメモリ負荷と I/O 負荷、また vfs_cache_pressure を調整することでルート 2 を通ることが確認できました。

5. バグの再現

これまでに作成したスクリプトで、通常の WARN_ON() のないカーネルに戻してバグが発生するか確認します。ルート 1 を通る umount コマンドとルート 2 の再現 TP である vm_stress.sh、inode.sh を同時に実行します。どちらもすぐにルートを通っていたので、不具合もすぐに発生するはずです。

5-1. 通常のカーネルで再現試験

WARN_ON() のない通常のカーネルで数時間スクリプトを実行しましたが、バグは発生しませんでした。

そのため再現させやすくするために mdelay() を入れます。mdelay() はスケジューリングせずにただそこで遅延するだけです。spin_unlock() してから spin_lock() するまでの時間が長くなるためプロセスXとプロセスYが競合するタイミングが取りやすくなります。

カーネル内には msleep() というスリープする関数もありますが、これはスケジューリングしてしまうため今回は使用しません。

5-2. mdelay() で再現確率を上げる

以下の修正で試します。mdelay() で 50 ミリ秒遅延するようにします。

```
--- fs/inode.c.orig     2008-10-16 20:47:38.000000000 +0900
+++ fs/inode.c  2008-10-16 20:39:42.000000000 +0900
@@ -443,6 +443,7 @@ static void prune_icache(int nr_to_scan)
                if (inode_has_buffers(inode) || inode->i_data.nrpages) {
                        __iget(inode);
                        spin_unlock(&inode_lock);
+                       mdelay(50);
                        if (remove_inode_buffers(inode))
                                reap += invalidate_inode_pages(&inode->i_data);
                        iput(inode);
@@ -1035,6 +1036,7 @@ static void generic_forget_inode(struct
                        list_move(&inode->i_list, &inode_unused);
                inodes_stat.nr_unused++;
                spin_unlock(&inode_lock);
+               mdelay(50);
                if (!sb || (sb->s_flags & MS_ACTIVE))
                        return;
                write_inode_now(inode, 1);
```

さらに再現 TP を改善しました。スリープ時間や I/O のサイズ、メモリ負荷の割合などを調整しています（最終的な再現 TP は本書のサポートページ（http://www.oreilly.co.jp/books/9784873114040/）からダウンロードしてください）。

mdelay() を入れたカーネルと最適化した TP を実行したところ、数分で再現するようになりました。以下は実際に取得したダンプのバックトレースです。umount コマンドの延長で generic_forget_inode() が呼ばれ、パニックしています。

```
crash> bt
PID: 4733   TASK: 100139c27f0       CPU: 2   COMMAND: "umount"
 #0 [10042975b40] start_disk_dump at ffffffffa01a336d
 #1 [10042975b70] try_crashdump at ffffffff8014bd01
 #2 [10042975b80] do_page_fault at ffffffff80123978
 #3 [10042975c00] find_get_pages_tag at ffffffff8015b24c
 #4 [10042975c60] error_exit at ffffffff80110d9d
    [exception RIP: __writeback_single_inode+643]
    RIP: ffffffff80199088  RSP: 0000010042975d18  RFLAGS: 00010246
...
    R13: 000001013ab46800  R14: 000001013a187d78  R15: 0000010042975d58
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
 #5 [10042975d10] __writeback_single_inode at ffffffff80198f96
 #6 [10042975d50] write_inode_now_err at ffffffff80199204
 #7 [10042975db0] generic_drop_inode at ffffffff80193388
 #8 [10042975dd0] journal_destroy at ffffffffa007a80f
 #9 [10042975e50] ext3_put_super at ffffffffa00913dd
#10 [10042975e80] generic_shutdown_super at ffffffff8017f855
#11 [10042975ea0] kill_block_super at ffffffff80180677
#12 [10042975eb0] deactivate_super at ffffffff8017f776
#13 [10042975ed0] sys_umount at ffffffff80195217
#14 [10042975ef0] sys_newstat at ffffffff80182a44
#15 [10042975f50] error_exit at ffffffff80110d9d
...
```

6. コミュニティの履歴を確認

このバグはカーネル 2.6.12 のパッチをバックポートすると直ります。

mdelay() を入れたカーネルでも再現しないことを確認するのが重要です。

```
commit 991114c6fa6a21d1fa4d544abe78592352860c82
Author: Alexander Viro <aviro@redhat.com>
Date:   Thu Jun 23 00:09:01 2005 -0700

    [PATCH] fix for prune_icache()/forced final iput() races

commit 4a3b0a490d49ada8bbf3f426be1a0ace4dcd0a55
```

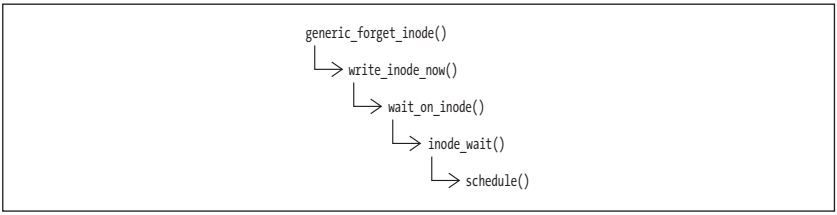

図 5-8 修正後のコールシーケンス

```
Author: Jan Blunck <jblunck@suse.de>
Date:   Sat Feb 10 01:44:59 2007 -0800

    [PATCH] igrab() should check for I_CLEAR
```

この修正はアンロックする前に I_WILL_FREE フラグを立てます。inode を解放しようとするときに I_WILL_FREE フラグが立っていれば、解放を中止します。I_WILL_FREE フラグが解放の予約済みを意味するような役割になり、2 つのプロセスで同じ inode を解放しなくなります。

簡単に考えると、途中でアンロックしなければいいのですが、**図 5-8** の関数シーケンスでスケジュールするため、アンロックしなければならず、このような修正になっています。

今回はたまたま正しく修正されるパッチがありましたが、自分で修正する場合は、このような仕組みを理解する必要があります。

まとめ

ソースコードをひとつずつ確認し、バグ発生のルートを通るように OS を操作することで、再現させることができました。再現 TP を作ることで、バックポートや自作のパッチを検証することができます。また今回は WARN_ON(1) や sysctl のパラメータをうまく利用できた例だと思います。状況や OS の動作によって適切に使い分けましょう。

—— Naohiro Ooiwa

HACK #36

カーネルのストール（無限ループ編）

実際にあった OS がストールする問題を取り上げてカーネルデバッグ手法を説明します。

ある日、OS がストールしたという報告を受けました。リアルタイムプロセスに kill コマンドでコアダンプさせるシグナルを送ると数十秒間 OS がストールしたように見えるという報告でした。問題のあった OS は、カーネルバージョンが 2.6.9 をベースにしたディストリビューションです。

問題の発生した状況を詳しくヒアリングする

まずはできる限り情報を集め、整理することが大切です。さらに詳しくヒアリングすると、次のような状況で問題が発生していることがわかりました。

- プロセスのスケジューリングポリシーが SCHED_FIFO
- マルチスレッド
- kill コマンドで SIGSEGV や SIGABRT などのコアダンプさせるシグナルを送る

これらすべてを満たした状況で発生していました。他にも報告者が気がつかないような条件があるかもしれませんし、この条件を満たさなければ発生しないという保証はまだありません。ただ、有効な情報であることは確かです。

次にこのリアルタイムプロセスがどんな処理をしているのかヒアリングしたところ、ネットワークサービスを提供するサーバアプリケーションで、複数の子スレッドで select() システムコールを使いクライアントからの要求を受け付けるものでした。また、select() システムコールは timeout 指定つきで実行していました。

またシステム負荷は低い状態で、再現頻度は高いようでした。

再現プログラムを作る

問題を手元の環境で再現させることができれば、解決までの道のりはぐっと短くなります。どんな問題でも再現できるものではありませんが、筆者はなるべく再現手順や再現プログラムが作れないか試してみるようにしています。今回は次のような再現プログラムでうまく再現させることができました。説明のためエラー処理などの細かい部分は省略しています。

```
$ cat segfault.c
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>

void thread (void* buf)
{
        struct timespec tm = { 0, SLEEP_NSEC };
        do {
                nanosleep(&tm, NULL);
```

```
        } while (1);
}

int main (void)
{
        pthread_t thr;

        for (i=0; i< NUM_THRAEDS; i++)
                pthread_create( &thr, NULL, thread, NULL );
        /* 子スレッドが動き出すために少しの間 CPU を手放す */
        sleep(2);
        /* 自分自身にコアダンプシグナルを送る */
        kill(0, SIGSEGV);
        return 0;
}
```

NUM_THREADS や SLEEP_NSEC はコンパイルする際に、gcc に -D オプションで渡すなどして変更できるようにしています。ポイントとなりそうなパラメータは簡単に変更できるようにしておくとテストする際に便利です。このプログラムを chrt コマンドを使って実行すると再現するようになりました。chrt コマンドは RedHat 系のディストリビューションなどに入っているコマンドで、スケジューリングポリシーを変更することができます。スケジューリングポリシーをリアルタイムに変更するためには root 権限が必要ですので、sudo コマンドをつけています。パラメータは SLEEP_NSEC を 100 ミリ秒、NUM_THREADS を CPU 数＋1 にしました。報告では数十秒間ストールするということでしたが、このプログラムでは無期限にストールするようになりました。

```
$ gcc -DSLEEP_NSEC=100000000 -DNUM_THREADS=3 -lpthread -o segfault segfault.c
$ sudo chrt -f 99 ./segfault
```

いろいろな条件で再現プログラムを試してみる

再現させることができたら、今度は少しずつ違う条件で再現するかどうかを確認します。筆者がよくやることは、同じ環境でカーネルだけ新しいバージョンに変えて実験してみることです。他にもいくつかやりましたが、それにより発生条件を細かく知ることができました。

- SCHED_FIFO の時だけ発生
- スレッド数が CPU 数＋1 以上で発生
- 意図的に NULL ポインタアクセスした場合は発生しない

- 再現頻度は 100% で、すぐに発生する
- システム負荷は関係ない
- 外部からの ping には反応する
  →ただし、反応が遅くなる
- キーボードを受け付けない
- ユーザランドは動いてなさそう
  →一切のコマンドを受け付けない
- 新しいバージョンのカーネルでは発生しない
- 古いバージョンのカーネルでも発生する

これらの情報をヒントに解析を進めていきます。

カーネルダンプ解析

この問題はストールしてしまうため、ウォッチドッグを使いカーネルクラッシュダンプを採取してデバッグすることにしました。ウォッチドッグについては「**IPMI watchdog timer により、フリーズ時にクラッシュダンプを取得する**」**[HACK #22]** を参考にしてください。

まず各スレッドが何をしているかを確認します。

```
crash> ps | grep segfault
   PID    PPID  CPU       TASK        ST  %MEM     VSZ    RSS  COMM
...
   3817   3781   1    1007debf7f0    RU   0.0   24108    528  segfault
> 3818   3781   0    1007cec5030    RU   0.0   24108    528  segfault
> 3819   3781   1    1007dddb030    IN   0.0   24108    528  segfault
```

PID:3817 のタスクが親スレッドで、kill() システムコールを実行したスレッドです。子スレッドが各 CPU のカレントタスクとなっていることがわかります。カレントタスクも気になりますが、まずは親スレッドが何をしているのか調べてみることにします。

カレントタスクは ps コマンド実行結果の最左列に「>」印がついているタスクです。

```
crash> bt 3817
PID: 3817   TASK: 1007debf7f0       CPU: 1   COMMAND: "segfault"
 #0 [100780b3c28] schedule at ffffffff8030d7b4
```

```
#1 [100780b3d00] sys_sched_yield at ffffffff80134802
#2 [100780b3d20] do_coredump at ffffffff80184e02
#3 [100780b3e10] get_signal_to_deliver at ffffffff801433a1
#4 [100780b3e50] do_signal at ffffffff8010f6fb
#5 [100780b3f50] ptregscall_common at ffffffff801105df
    RIP: 0000003200e2e829  RSP: 0000007fbffffab8  RFLAGS: 00000206
    RAX: 0000000000000000  RBX: 0000000000000000  RCX: ffffffffffffffff
    RDX: 0000000000000000  RSI: 000000000000000b  RDI: 0000000000000000
    RBP: 0000007fbffffaf0   R8: 0000007fbffff8e0   R9: 0000002a9557b190
    R10: 0000007fbffffa01  R11: 0000000000000206  R12: 00000000004007d0
    R13: 0000007fbffffbc0  R14: 0000000000000000  R15: 0000000000000000
    ORIG_RAX: 000000000000003e  CS: 0033  SS: 002b
```

do_coredump() がコールされているので、親スレッドがコアダンプ処理をしています。その延長で sched_yield() を実行して CPU を手放してしまった状態のようです。カーネルソースを確認したところ、確かにこのような動きになるようです（図 5-9）。

さらに task_struct やランキューに残ったタイムスタンプ情報を見ていくと、いったん手放した CPU が自分に戻ってこなくなっていることがわかりました。プロセスは致命的なシグナルを受け取ったため、コアダンプを作って終了しようとしています。その途中で CPU を手放してしまってよいのでしょうか。

では、CPU を独占しているカレントタスクが何をしているかを調べてみます。下記ではウォッチドッグがクラッシュダンプさせる処理部分を省いてあります。

```
crash> bt -a
PID: 3818   TASK: 1007cec5030       CPU: 0   COMMAND: "segfault"
...
#16 [100778b1e10] get_signal_to_deliver at ffffffff801433c9
#17 [100778b1e50] do_signal at ffffffff8010f6fb
#18 [100778b1f50] ptregscall_common at ffffffff801105df
    RIP: 000000320170bab5  RSP: 0000000040a001a0  RFLAGS: 00000202
    RAX: fffffffffffffdfc  RBX: 0000000000000000  RCX: ffffffffffffffff
    RDX: 0000000000000002  RSI: 0000000000000000  RDI: 00000000004008d0
```

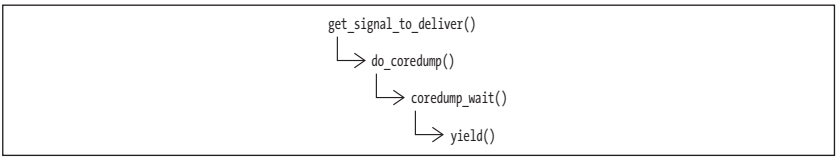

図 5-9　スレッドの動作確認

```
    RBP: 0000000040a001d0   R8: 0000000040a00960   R9: 0000000040a00960
    R10: 0000000040a00101  R11: 0000000000000202  R12: 0000003201706080
    R13: 000000320170d1c0  R14: 0000000000000000  R15: 000000320170d1c0
    ORIG_RAX: 00000000000000db  CS: 0033  SS: 002b

PID: 3819   TASK: 1007dddb030       CPU: 1   COMMAND: "segfault"
...
 #3 [100778b3ef0] schedule_timeout at ffffffff8030e1ad
 #4 [100778b3f50] nanosleep_restart at ffffffff8030e335
 #5 [100778b3f80] system_call at ffffffff8011026a
    RIP: 000000320170bab5  RSP: 0000000414011a0  RFLAGS: 00000202
    RAX: 00000000000000db  RBX: ffffffff8011026a  RCX: ffffffffffffffff
    RDX: 0000000000000002  RSI: 0000000000000000  RDI: 00000000004008d0
    RBP: 00000000414011d0   R8: 0000000041401960   R9: 0000000041401960
    R10: 00000000414019f0  R11: 0000000000000202  R12: 0000000000000000
    R13: 000000320170d1c0  R14: 0000000000000000  R15: 000000320170d1c0
    ORIG_RAX: 00000000000000db  CS: 0033  SS: 002b
```

PID:3818 のスレッドはシグナル処理をしているようです。PID:3819 は nanosleep() の再実行をしている最中のようです。nanosleep() などの一部のシステムコールはシグナルに割り込まれると、まずシグナルを処理してから残りの処理が再実行されます。カーネルダンプを見ることによって、問題が発生した時それぞれのスレッドが何を実行しているのかがわかりました。

デバッグコードを埋め込んで解析

バックトレースからは、カレントタスクにおかしなところはなさそうです。ダンプばかり見ていてもよくわからないので、カーネルソースに printk() を埋め込んで何に時間がかかっているのか調べました。ダンプ解析から、それぞれのスレッドがどこを実行しているのかがわかったので効果的にデバッグコードを埋め込むことができます。すると、次のようなシナリオで無限ループに陥っていることがわかりました。

nanosleep() はシグナルに割り込まれていると、スリープを中断してシグナルを処理しよ

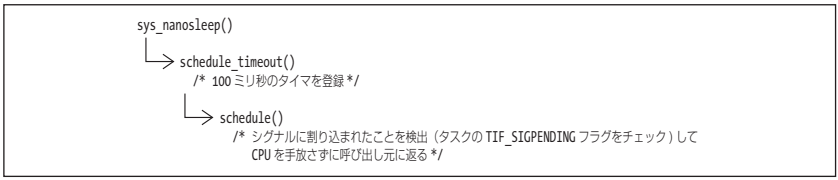

図 5-10　nanosleep() がシグナルによって中断される

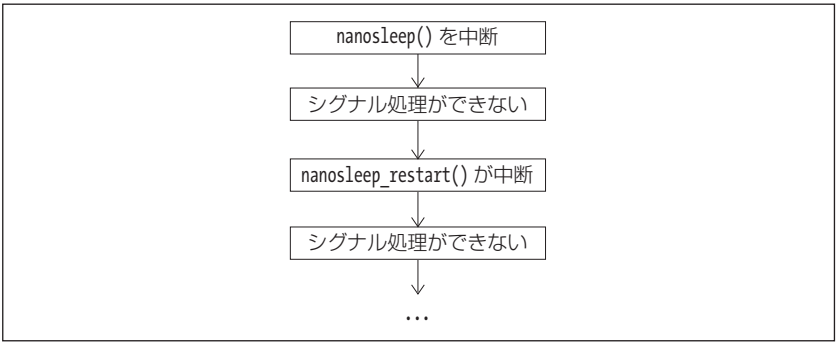

図 5-11　子スレッドが無限ループする様子

うとします（**図 5-10**）。そしてシグナル処理が終われば nanosleep_restart() を実行して、残り時間のスリープを始めます。ところが、今回はそのシグナル処理ができず**図 5-11** のようにスリープ中断とシグナル処理の失敗を繰り返していました。

プロセスにシグナルが送信されると、プロセス内のすべてのスレッドに対して TIF_SIGPENDING フラグを立て、シグナル情報をスレッド間で共有するキューへつなげます。nanosleep() が中断してしまうのは、この TIF_SIGPENDING が立っているためです。一方、シグナル処理失敗の原因は親スレッドがすでにシグナル受信処理を開始し、シグナル情報をスレッド間で共有するキューから取り出してしまっていたためでした。

通常であればシグナル受信処理を開始した親スレッドが SIGKILL を送信してすべての子スレッドの実行を終了させてしまうのですが、今回はそれをする前に子スレッドが暴走してしまっています。一連のシナリオを見ていると TIF_SIGPENDING フラグに問題がありそうです。処理すべきシグナル情報はすでに親スレッドが取り出してしまったにもかかわらず、子スレッドに対し処理すべきシグナルを受信していると知らせているわけですから。

TIF_SIGPENDING フラグは recalc_sigpending_tsk() 内でクリアされます。recalc_sigpending_tsk() は現在のシグナル受信状況をチェックするため、シグナル受信処理の延長でほとんどの場合実行されます。**図 5-12** にコールフローを示します。

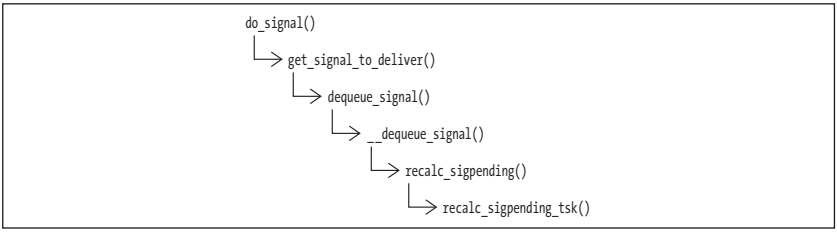

図 5-12　シグナル受信処理のコールフロー

recalc_sigpending_tsk() は次のようになっています。

[kernel/signal.c]
```
fastcall void recalc_sigpending_tsk(struct task_struct *t)
{
        if (t->signal->group_stop_count > 0 ||
            PENDING(&t->pending, &t->blocked) ||
            PENDING(&t->signal->shared_pending, &t->blocked))
                set_tsk_thread_flag(t, TIF_SIGPENDING);
        else
                /* TIF_SIGPENDINGをクリアする */
                clear_tsk_thread_flag(t, TIF_SIGPENDING);
}
```

今回の再現プログラムではシグナルマスクの変更はしていませんので、TIF_SIGPENDING がクリアされるためには group_stop_count が 0 以下である必要があります。この group_stop_count はシグナル送信処理の中（つまり kill() システムコールの延長）で、プロセスに属するスレッド数に設定されています。実はこのカウントは do_coredump() 内でクリアされるようになっています。

[fs/exec.c]
```
int do_coredump(long signr, int exit_code, struct pt_regs * regs)
{
…
        coredump_wait(mm);

        /*
         * Clear any false indication of pending signals that might
         * be seen by the filesystem code called to write the core file.
         */
        current->signal->group_stop_count = 0;
…
```

しかもよく見ると、親スレッドが CPU を手放すきっかけとなってしてしまった coredump_wait() よりも後にクリアされています。ここまでのデバッグによって、このストールの原因は TIF_SIGPENDING フラグであり、それをクリアするための group_stop_count を 0 にするタイミングに問題があることがわかりました。

コミュニティの履歴をチェックする

今回は再現プログラムで新しいバージョンのカーネルでは、問題が発生しないことを確認しています。そこで「**カーネルパニック（NULL ポインタ参照編）**」**[HACK #33]** と同様にLinus Torvals 氏の git ツリーから関連する修正を検索したところ、次のパッチを発見しました。

```
[PATCH] do_coredump() should reset group_stop_count earlier
commit bb6f6dbaa48c53525a7a4f9d4df719c3b0b582af
```

筆者の環境でこの修正を適用することで問題が解決することが確認できました。この修正は coredump_wait() を実行する前に group_stop_count を 0 にするという修正です。このパッチで問題は直るのですが、合わせて次のクリーンアップパッチも適用しました。このクリーンアップパッチでは筆者が疑問に思った do_coredump() の延長で yield() するコードを削除しています。

```
[PATCH] coredump_wait() cleanup
commit 2384f55f8aa520172c995965bd2f8a9740d53095
```

まとめ

障害解析をする前に、原因の切り分けが大切です。そのための重要なポイントを説明しました。

- 問題発生時の状況を詳しくヒアリングする
- 自分の環境で再現させること
- いろいろ条件を変えて試験する

また実際の解析では、カーネルダンプ解析をとおして問題発生時にカーネルがどのように動いていたかを調べることにより、効果的にデバッグコードを埋め込むことができます。git のログをチェックするのはその後です。原因がはっきりしてはじめて、どのパッチをポーティングすればよいのかわかるからです。

参考文献

- Linus Torvals 氏の git ツリー
  http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary

—— Toyo Abe

HACK #37 カーネルのストール（スピンロック編その 1）

スピンロックで単純なデッドロックになったときのダンプの見え方を紹介します。

カーネル内でスピンロックによりデッドロックしたときのダンプを確認します。本 Hack では Linux カーネル 2.6.9 を使用しています。

再現

以下の spinlock_stall.c はスピンロックでデッドロックするモジュールです。

```
# cat spinlock_stall.c
#include <linux/module.h>
#include <linux/kthread.h> // for kthread_run()
#include <linux/delay.h> // for msleep()

DEFINE_SPINLOCK(lock1);
DEFINE_SPINLOCK(lock2);
int thread2_flag;

static int spinlock_stall_thread1(void *data)
{
    spin_lock(&lock1);
    while(1){
        if (thread2_flag == 1) break;
        msleep(200);
    }
    spin_lock(&lock2);
    return 0;
}
static int spinlock_stall_thread2(void *data)
{

    spin_lock(&lock2);
    thread2_flag = 1;
    spin_lock(&lock1);
    return 0;
}
static int __init spinlock_init(void)
{
    struct task_struct *kthread1;
    struct task_struct *kthread2;
```

```
    spin_lock_init(&lock1);
    spin_lock_init(&lock2);

    kthread1 = kthread_run(spinlock_stall_thread1, NULL, "spinlock1");
    kthread2 = kthread_run(spinlock_stall_thread2, NULL, "spinlock2");

    return 0;
}
static void __exit spinlock_exit(void) { return; }

module_init(spinlock_init);
module_exit(spinlock_exit);
# cat Makefile
obj-m := spinlock_stall.o
```

このモジュールをカーネルに組み込むと thread1 と thread2 でデッドロックになります。make と insmod は以下のように行いました。make の -C オプションには現在起動しているカーネルのソースディレクトリを指定します。

```
# ls
Makefile spinlock_stall.c
# make -C /usr/src/linux M=`pwd` modules
...
# insmod spinlock_stall.ko
```

insmod した直後にストールし、IPMI watchdog でダンプが取れます。

ダンプ解析

ダンプのバックトレースを見てみます。

```
crash> bt -ta
PID: 13417  TASK: 1007de867f0       CPU: 0   COMMAND: "spinlock1"
...
--- <IRQ stack> ---
  [     1006ddb9e58] apic_timer_interrupt at ffffffff80110bf5
    [exception RIP: .text.lock.spinlock+2]
...
  [     1006ddb9f00] msleep at ffffffff801407f9
  [     1006ddb9f10] spinlock_stall_thread1 at ffffffffa01cf02e
```

```
  [     1006ddb9f20] kthread at ffffffff8014b6c3
  [     1006ddb9f50] child_rip at ffffffff80110f47
  [     1006ddb9f58] keventd_create_kthread at ffffffff8014b6ec
  [     1006ddb9fc8] kthread at ffffffff8014b5fb
...
PID: 13418  TASK: 1007dd88030       CPU: 1   COMMAND: "spinlock2"
             START: smp_call_function_interrupt at ffffffff8011c5f0
...
--- <IRQ stack> ---
  [     1006e613e58] call_function_interrupt at ffffffff80110b69
    [exception RIP: .text.lock.spinlock+5]
...
  [     1006e613f10] spinlock_stall_thread2 at ffffffffa01cf055
  [     1006e613f20] kthread at ffffffff8014b6c3
  [     1006e613f50] child_rip at ffffffff80110f47
  [     1006e613f58] keventd_create_kthread at ffffffff8014b6ec
  [     1006e613fc8] kthread at ffffffff8014b5fb
...
```

スピンロックはビジーウェイト（スケジュールしないで無限ループする）なのでバックトレースに素直に現れます。「.text.lock.spinlock」とあればほとんどの場合、スピンロックで止まっていると判断して間違いありません（「.text.lock.spinlock」はカーネルのバージョンによって変わることがあります。カーネル 2.6.28 では「_spin_lock」と表示されます）。

まとめ

スピンロックによるデッドロックの場合はダンプからすぐにわかります。ただし現在の Linux カーネルにはこのようなバグはほとんどありません。カーネルを修正したときやドライバ、モジュールなどを作って単純な間違いをしたときに見ることのほうが多いです。

—— Naohiro Ooiwa

HACK #38

カーネルのストール（スピンロック編その 2）

NMI watchdog timeout 発生時のカーネルデバッグ手法について実例を使って説明します。

新しいハードウェアを入手したので Kdump の連続試験をしていました。すると、たまに次のようなメッセージがコンソールに表示され、Kdump が失敗するという現象にぶつかりました。使用したのはカーネルバージョンが 2.6.18 をベースにしたディストリビューションです。

```
ide0 at 0x1f0-0x1f7,0x3f6 on irq 14NMI Watchdog detected LOCKUP on CPU 0
CPU 0
Modules linked in:
Pid: 1, comm: swapper Not tainted 2.6.18-prep #6
RIP: 0010:[<ffffffff80063b7c>]  [<ffffffff80063b7c>] .text.lock.spinlock+0x2/0x30 ―――①
RSP: 0000:ffffffff8040fd00  EFLAGS: 00000086 ―――――――――――――――――――――――――②
RAX: ffff8100014cdf00 RBX: ffffffff803af380 RCX: 0000000000000000
RDX: ffffffff80407f00 RSI: ffffffff8040fd48 RDI: ffffffff803af3bc
RBP: 0000000000000000 R08: 0000000000000003 R09: 000000000079e321
R10: 0000000000000096 R11: 0000000000000086 R12: 0000000000000000
R13: 000000000000000e R14: ffffffff803af3bc R15: ffffffff8040fd48
FS:  0000000000000000(0000) GS:ffffffff80397000(0000) knlGS:0000000000000000
CS:  0010 DS: 0018 ES: 0018 CR0: 000000008005003b
CR2: 00002aaaae176000 CR3: 0000000001001000 CR4: 00000000000006e0
Process swapper (pid: 1, threadinfo ffff810008d00000, task ffff8100019fd7a0)
Stack:  ffffffff800b5efb 0000000000000086 0000000000000000 ffffffff8040fd48
 0000000000000000 000000000000000e ffffffff8040feb8 0000000000000001
 ffffffff8006b3bf 0000000300000000 ffff8100084734c0 ffffffff8040fd70
Call Trace:
 <IRQ>  [<ffffffff800b5efb>] __do_IRQ+0x47/0x105
 [<ffffffff8006b3bf>] do_IRQ+0xe7/0xf5 ―――――――――――――――――――――――――③
 [<ffffffff8005c615>] ret_from_intr+0x0/0xa
 [<ffffffff8002e007>] __wake_up+0x38/0x4f
 [<ffffffff800107be>] handle_IRQ_event+0x1b/0x58 ―――――――――――――――――――――④
 [<ffffffff8000d276>] ide_intr+0x11f/0x1df
 [<ffffffff800b69bd>] note_interrupt+0x13a/0x227 ―――――――――――――――――――――⑤
 [<ffffffff800b5f7b>] __do_IRQ+0xc7/0x105
 [<ffffffff8006b3bf>] do_IRQ+0xe7/0xf5 ―――――――――――――――――――――――――⑥
 [<ffffffff8005c615>] ret_from_intr+0x0/0xa
 [<ffffffff80159597>] vgacon_cursor+0x0/0x1a5
 [<ffffffff80011cd5>] __do_softirq+0x53/0xd5
 [<ffffffff8005d2fc>] call_softirq+0x1c/0x28
 [<ffffffff8006b53c>] do_softirq+0x2c/0x85
 [<ffffffff8005cc8e>] apic_timer_interrupt+0x66/0x6c
 <EOI>  [<ffffffff80159597>] vgacon_cursor+0x0/0x1a5
 [<ffffffff8008e71e>] vprintk+0x29e/0x2ea
 [<ffffffff8005a5bb>] cache_alloc_refill+0x106/0x186
 [<ffffffff8008e7bc>] printk+0x52/0xbd
. . .
```

1 行目を見ると NMI watchdog によってデッドロックを検出していることがわかります。NMI watchdog ではデッドロックを検出した CPU の情報をメッセージとしてダンプします。NMI watchdog については「**NMI watchdog により、フリーズ時にクラッシュダンプを取得する**」**[HACK #23]** を参照してください。この現象が起きたのは /proc/sysrq-trigger によってパニック発生後、ダンプカーネルが起動する最中です。NMI watchdog によってデッドロックを検出するとダンプ採取することができるのですが、今回のケースは Kdump 中の問題であるためダンプさせることができません。まずはこのメッセージからデバッグしていくことにします。NMI watchdog によって表示されるメッセージの読み方は、ほとんど Oops メッセージの読み方と同じです。Oops メッセージについては「**Oops メッセージの読み方**」**[HACK #15]** を参照してください。

メッセージを読み解く

まずは簡単に問題発生時の状況を確認します。メッセージの①には現在実行中のコードが表示されます。「.text.lock.spinlock」とあるのでスピンロック獲得処理を実行中です。次に②の EFLAGS を見ると IF フラグ（割り込みフラグ）がクリアされています。つまり割り込み禁止状態のスピンロック獲得処理を実行中ということになります。

IF フラグは x86 および x86_64 アーキテクチャであれば EFLAGS レジスタの 10 番目のビットです。00000086 は IF フラグがクリアされているので割り込み禁止状態です。もしこれが 00000286 となっていたら IF フラグがセットされているので割り込み許可状態であることがわかります。

さて少し横道にそれますが、たいていのディストリビューションでは、Kdump のダンプカーネルの起動パラメータに次のオプションを指定するようになっています。

```
maxcpus=1 irqpoll
```

1 つ目はダンプカーネルは UP（ユニプロセッサ）で動かすということで、2 つ目は IRQ のポーリングを有効化するということです。カーネルの起動パラメータの詳細はカーネルソースツリーに同梱されている Documentation/kernel-parameters.txt を参照してください。今回のケースでは 1 つ目は特に重要で、デッドロックは複数 CPU によるレースコンディションではなく、同じ CPU が複数回同じロックを取ろうとしたために発生していることがわかります。つまり、このメッセージに表示されているスタックトレースに必要な情報のすべてが入っているということです。

ではスタックトレースを見ていきましょう。③で do_IRQ() コールがあるので IRQ 割り込みの受信ハンドラ実行中にデッドロックが発生していることがわかります。do_IRQ() は、

受信したIRQ割り込みに対応する割り込みハンドラを起動させるルーチンです。do_IRQ()関数で獲得するロックは1つしかありません。IRQ番号ごとに存在する割り込みディスクリプタ（struct irq_desc *desc）のロックdesc->lockです。このdesc->lockでデッドロックを引き起こしていたようです。

⑥を見ると、以前に同じdo_IRQ()がコールされていたことがわかります。IRQ割り込み処理中にさらにIRQ割り込みが入ったということでしょう。⑥でdesc->lockを獲得し、それを知らずに③で同じdesc->lockを得ようとしてデッドロック。こんな単純な問題があるのでしょうか？と疑いたくなります。Kdumpせずに普通に使っている分にはこのようなデッドロックを見たことがありません。IRQ割り込みを立て続けに受信しただけでデッドロックするなら、まともにOSが動かないはずです。何が普段と違っているのでしょうか？そこで注目したのがもうひとつの起動オプションであるirqpollです。IRQをポーリングする……いかにも怪しそうです。

irqpollオプションについて

IRQ割り込みが発生した際に対応するハンドラが見つからないケースがあります。デバイスのファームウェアにバグがあるために起こる場合もあります。しかしKdumpを使用している環境ではよくあることです。パニックするようなケースではデバイスのシャットダウンが行えずダンプカーネルが対応するドライバをロードする前に、デバイスから割り込みが上がってきてしまうことがあるからです。次のようなメッセージをよく見ることがあります。

```
irq X: nobody cared (try booting with the "irqpoll" option)
```

これは受信したIRQ割り込みに対応するハンドラが見つからなかった時に表示されるメッセージです。これが多発するとそのIRQは無効化されてしまいます。それを避けるためにたいていのディストリビューションではダンプカーネルの起動パラメータにirqpollオプションを付けて、回避させるようにしています。

irqpollに注目してソーストレース

⑤のnote_interrupt()に着目してください。これは受信したIRQ割り込みが正しく処理されたかどうかチェックする関数です。対応する割り込みハンドラが見つからなかった場合、通常は先ほど説明したメッセージを表示するだけなのですが、irqpollオプションが指定されている場合、少し動作が変わります。

[kernel/irq/handle.c]

```
fastcall unsigned int __do_IRQ(unsigned int irq, struct pt_regs *regs)
{
   struct irq_desc *desc = irq_desc + irq;
…
        /*
         * desc->lock をロックしたまま、do_IRQ() から note_interrupt()
         * がコールされる
         */
        spin_lock(&desc->lock); ――――――――――――――――――――⑦
        if (!noirqdebug)
              note_interrupt(irq, desc, action_ret, regs);
…
```

[kernel/irq/spurious.c]

```
void note_interrupt(unsigned int irq, struct irq_desc *desc,
                    irqreturn_t action_ret, struct pt_regs *regs)
{
…
  if (unlikely(irqfixup)) {
          /* Don't punish working computers */
          if ((irqfixup == 2 && irq == 0) || action_ret == IRQ_NONE) {
               int ok = misrouted_irq(irq, regs);
                 if (action_ret == IRQ_NONE)
                         desc->irqs_unhandled -= ok;
          }
  }
…
```

irqpoll オプションが指定されていると irqfixup が 2 になり、misrouted_irq() 関数がコールされるようになります。misrouted_irq() 関数は対応するハンドラが間違った IRQ 番号に登録されていないかを調べるため、他の IRQ 番号に登録されたハンドラを検索し、コールしていきます。うまく対応するハンドラが見つかれば、この関数は 1 を返します。misrouted_irq() の中身を少し見てみましょう。

[kernel/irq/spurious.c]

```
static int misrouted_irq(int irq, struct pt_regs *regs)
{
```

```
    for (i = 1; i < NR_IRQS; i++) {
        struct irq_desc *desc = irq_desc + i;
        struct irqaction *action;

        if (i == irq)   /* 他のIRQ番号だけが対象 */
                continue;
        …
        /* 他のIRQ番号に登録されたハンドラをコールしていく */
        …
        while ((desc->status & IRQ_PENDING) && action) {
                /*
                 * 上記処理中に発生したIRQ割り込みで未処理のもの
                 * (ペンディングされた割り込み)を処理していく
                 */
                work = 1;
                spin_unlock(&desc->lock); ──────────────────⑧
                handle_IRQ_event(i, regs, action);
                spin_lock(&desc->lock);
                desc->status &= ~IRQ_PENDING;
        }
…
```

コメント部分に書いた、ペンディングされた割り込みを処理する際にhandle_IRQ_event()関数をコールしています。冒頭のデッドロックメッセージの④を見てください。この関数名がスタックトレースに表示されていますね。怪しいです。⑧でdesc->lockをアンロックしていますが、これは⑦でnote_interrupt()がコールされた時にロックされていたdesc->lockとは別のロックです。ではhandle_IRQ_event()を見てみましょう。では handle_IRQ_event()を見てみましょう。

[kernel/irq/handle.c]

```
irqreturn_t handle_IRQ_event(unsigned int irq, struct pt_regs *regs,
                             struct irqaction *action)
{
…
  if (!(action->flags & IRQF_DISABLED))
          local_irq_enable_in_hardirq();
…
```

action->flagsのIRQF_DISABLEDフラグがない場合、割り込み許可にされてしまうようです。

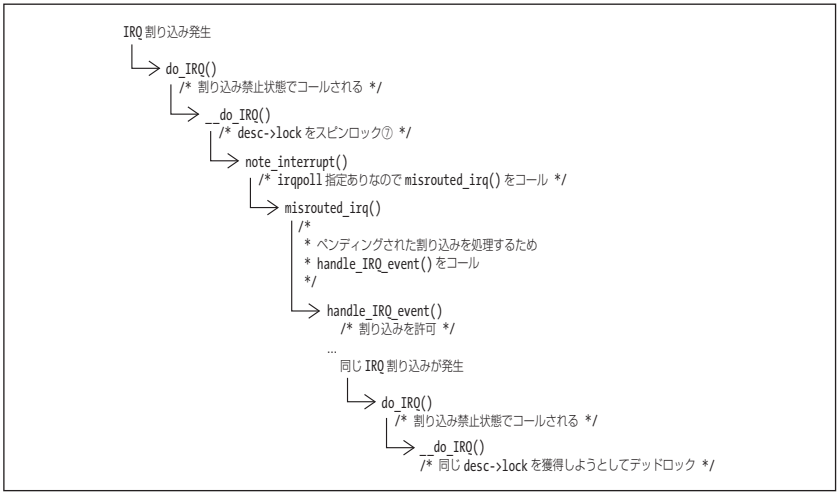

図 5-13　デッドロックに至るシナリオ

IRQ を他デバイスと共有するタイプのデバイスであれば、ここに引っかかってしまいます。察しの良い人ならもう気がついたと思いますが、シナリオを**図 5-13** にまとめます。

コミュニティの履歴をチェックする

ここまでの解析で問題点がハッキリしました。desc->lock を取ったまま note_interrupt() をコールしているのがまずそうです。他 IRQ のハンドラを探すために、この割り込みディスクリプタのロックを取りっぱなしにする必要はないと思います。

コミュニティではどうなっているのでしょうか？履歴をチェックすると次の 2 つのパッチで、筆者の考えと同じ修正が入っていました。

```
commit f72fa707604c015a6625e80f269506032d5430dc
Author: Pavel Emelianov <xemul@openvz.org>
Date:   Fri Nov 10 12:27:56 2006 -0800

    [PATCH] Fix misrouted interrupts deadlocks

commit b42172fc7b569a0ef2b0fa38d71382969074c0e2
Author: Linus Torvalds <torvalds@woody.osdl.org>
Date:   Wed Nov 22 09:32:06 2006 -0800

    Don't call "note_interrupt()" with irq descriptor lock held
```

まとめ

シングルCPUによるデッドロックという実例を取り上げ、NMI watchdogによるカーネル障害メッセージからデバッグしていく手法を説明しました。また、今回のケースのようにカーネルの起動パラメータに着目することによって、原因を絞り込むことができる場合があることを説明しました。

参考文献

- ミラクル・リナックス「Linux 110番」
  http://www.miraclelinux.com/support/?q=node/246

—— Toyo Abe

HACK #39 カーネルのストール（セマフォ編）

実際にあったセマフォのデッドロック問題を取り上げて解析事例を紹介します。

問題内容

Linuxカーネル2.6.9ベースのディストリビューションにおいてカーネルの評価を行っていたところ、ユーザプロセスが応答しなくなる問題が発生しました。状況を確認しようと、psコマンドを実行したところ、psコマンドの応答も返ってきません。

なお、その他のプロセスは問題なく動作しているようです。

- Linuxカーネル2.6.9ベースのディストリビューション
- CPU x86_64
- メモリ8GB

クラッシュダンプの収集

問題プロセスの詳細状況の確認と解析を行うためクラッシュダンプの収集を行いました。

ここから、クラッシュダンプからの問題解析の詳細を説明します。

プロセスの状態確認

応答のなくなったpsコマンドがどうなっているか確認します。

crashのpsコマンドを使用して、応答のないプロセス（psコマンド）の状態を見てみます。

```
crash> ps | grep ps
   2943   2596   2     1022f85d170    UN   0.0    5408   1036  ps
```

pid は 2943 でステータスは UN（=UNINTERRUPTABLE）です。このプロセスの応答がない理由を考えます。次のような考察があるでしょう。

- ディスク I/O 待ち。
- 何らかのイベント待ち、でも通常のスリープではない。
- ビジーループでもない。

プロセスのバックトレース確認

次にバックトレース情報を確認し、この状態（UNINTERRUPTABLE）に至ったルートを確認します。どういった処理において本現象が発生しているかを特定することは問題を解析する上で重要となります。

crash の bt コマンドを使用して、バックトレース情報を出力します。

```
crash> bt 2943
PID: 2943   TASK: 1022f85d170       CPU: 2   COMMAND: "ps"
 #0 [10226357c88] schedule at ffffffff805537d7 ──────────②
 #1 [10226357db0] __down_read at ffffffff80554bbf ──────────①
 #2 [10226357df0] access_process_vm at ffffffff801413ca ──────────③
 #3 [10226357e70] proc_pid_cmdline at ffffffff801bb0e5
 #4 [10226357eb0] proc_info_read at ffffffff801bb5e0
 #5 [10226357ef0] vfs_read at ffffffff8018613e
 #6 [10226357f20] sys_read at ffffffff801863dc
 #7 [10226357f80] no_syscall_entry_trace at ffffffff8010e539
    RIP: 0000002a95827232  RSP: 0000007fbfffe640  RFLAGS: 00010202
    RAX: 0000000000000000  RBX: ffffffff8010e539  RCX: 0000000000000000
    RDX: 00000000000007ff  RSI: 0000007fbfffddf0  RDI: 0000000000000006
    RBP: 0000000000000000   R8: 0000000000000000   R9: 0000000000000000
    R10: 00000000746f6f72  R11: 0000000000000246  R12: 0000000000000000
    R13: 0000000000000006  R14: 00000000005464a0  R15: 0000000000000000
    ORIG_RAX: 0000000000000000  CS: 0033  SS: 002b
```

このバックトレース情報から、ps コマンドのプロセスは __down_read() を呼び出し①、その延長で schedule() がコールされている②ことがわかります。これは読み込み用セマフォ待ちになっていると言えます。

デッドロック

セマフォ操作など排他処理はその構造上、デッドロックの可能性が常に存在します。ここで、セマフォ待ちから返ってこない状態、すなわちデッドロックに陥っている可能性が

あるとみて良いでしょう。

セマフォ待ちになっていることがわかりましたが、何のセマフォかわからないことには、どういった処理における問題であるのかはっきりしません。したがって、次は何のセマフォ操作を行おうとしているのかを調べる必要があります。そのために、down_read() の呼び出し元③である関数 access_process_vm() の確認を行います。

ソースから処理の確認を行います。

```
kernel/ptrace.c:
int access_process_vm(struct task_struct *tsk, unsigned long addr, void *buf, int len, int write)
{
        struct mm_struct *mm;
        struct vm_area_struct *vma;
        struct page *page;
        void *old_buf = buf;

        mm = get_task_mm(tsk);
        if (!mm)
                return 0;

        down_read(&mm->mmap_sem); ──────────────────────────────────④
        /* ignore errors, just check how much was sucessfully transfered */
        while (len) {
        …
        }
        up_read(&mm->mmap_sem); ────────────────────────────────────⑤
        mmput(mm);

        return buf - old_buf;
}
```

この関数の処理を見たところ、メインの while() ループの前後で、mm->mmap_sem に対して、セマフォによる排他制御を行っていることがわかります（④と⑤）。

これは、対象プロセスのメモリ構造体に対する処理の保護になります。読み込みに対するセマフォ要求なので、デッドロックのパターンとしては、他のプロセスが書き込みに対するセマフォを取得したままである可能性が考えられます。しかし、システムとしてはアイドル状態で、セマフォを持ったままビジー状態のプロセスがいる様子はありません。

読み込みセマフォが確保できない時は書き込みに対するセマフォが絡んでいるはずです。

ここでは、他のプロセスが書き込みセマフォを保持していないか調べることにします。

通常、メモリ構造体にアクセスするのはそのメモリ構造体を持っているプロセス自身です。他のプロセスのメモリ構造体にアクセスすることは滅多にありません。

バックトレース情報から、ps コマンドによる proc ファイルシステムを介してアクセスしていることがわかります。

では、proc ファイルシステムでアクセスしようとしている対象のプロセスは何でしょう？

/proc/pid/cmdline の読み込み中に見えます。

システムコールの引数を確認します。詳しくは AMD64 の ABI（Application Binary Interface）を参照願います。

```
RAX=0 => read システムコール
RDI=6 => fd=6
RSI   => ptr
RDX   => size
```

したがって、fd=6 のファイルが何かわかれば良いことになります。

crash の files コマンドを使用して、ps コマンドプロセスがオープンしているファイル一覧を取得して、fd=6 のファイルを確認します。

```
crash> files 2943
PID: 2943   TASK: 1022f85d170       CPU: 2   COMMAND: "ps"
ROOT: /    CWD: /root
 FD       FILE            DENTRY           INODE       TYPE PATH
  0    1022e3e5e40      102244026a8      1022e4813a0   CHR  /dev/pts/1
  1    1022e3e5e40      102244026a8      1022e4813a0   CHR  /dev/pts/1
  2    1022e3e5e40      102244026a8      1022e4813a0   CHR  /dev/pts/1
  3    1022ea1a2c0      1022dcaca48      1022957d320   REG  /proc/uptime
  4    1022ea1ab80      10224edfde8      1022957d098   REG  /proc/meminfo
  5    1022e621e80      100cff563f0      100cff51d00   DIR  /proc/
  6    1022e621200      10224402b30      1022dad0a78   REG  /proc/2639/cmdline
```

対象のメモリ構造体はプロセス pid=2639 のものであることがわかりました。それでは、pid=2639 の情報を見てみましょう。crash の ps コマンドと bt コマンドを使用します。

```
crash> ps | grep 2639
   2639   2554   3     1022e9920f0    UN   0.0   31248    696  MYAPL
```

```
crash> bt 2639
PID: 2639   TASK: 1022e9920f0       CPU: 3   COMMAND: "MYAPL"
 #0 [10229a65b88] schedule at ffffffff805537d7
 #1 [10229a65cb0] __down_read at ffffffff80554bbf
 #2 [10229a65cf0] do_page_fault at ffffffff80121fcc
 #3 [10229a65e20] error_exit at ffffffff8010f0dd
    [exception RIP: copy_user_generic+178]
    RIP: ffffffff802d1632  RSP: 0000010229a65ed8  RFLAGS: 00010202
    RAX: 0000000000000000  RBX: 0000000000000001  RCX: 0000000000000001
    RDX: 0000000000000001  RSI: 0000010225f44000  RDI: 0000002a95c018f0
    RBP: 0000000000000001   R8: 00000000ffffffffa   R9: ffffffff806bcd10
   R10: 0000000000000001  R11: 0000000000000001  R12: 0000000000000001
    R13: 0000000008000201  R14: 0000000000000001  R15: 0000010225f44000
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0000
 #4 [10229a65ee0] mincore_vma at ffffffff80177a9a
 #5 [10229a65f40] sys_mincore at ffffffff80177bf3
 #6 [10229a65f80] no_syscall_entry_trace at ffffffff8010e539
    RIP: 0000002a958323f9  RSP: 0000007fbffff3c0  RFLAGS: 00010206
    RAX: 000000000000001b  RBX: ffffffff8010e539  RCX: 000000000000000c
    RDX: 0000002a95c018f0  RSI: 0000000000001000  RDI: 0000000000400000
    RBP: 0000000000000000   R8: 0000000000020711   R9: 0000002a95c008e0
    R10: 0000000000000003  R11: 0000000000000206  R12: 0000000000000000
    R13: 0000000000000000  R14: 0000000000400000  R15: 0000000000000002
    ORIG_RAX: 000000000000001b  CS: 0033  SS: 002b
```

見つかったプロセスMYAPLは最初に応答のなくなったユーザプロセスです。バックトレース情報から、応答の返ってこないpsコマンドプロセスと同様に読み込みに対するセマフォ待ちであることがわかります。また、このセマフォ操作はページフォルト例外の延長で起きているように見えます。

例外処理中にデッドロックするパターンとして、ある排他オブジェクトを確保したまま、再度、同じ排他オブジェクトを確保しようとするパターンがあります。

ページフォルト例外を起こしている処理フローを確認します。

```
mm/mincore.c:
asmlinkage long sys_mincore(unsigned long start, size_t len,
        unsigned char __user * vec)
{
        int index = 0;
        unsigned long end;
```

```
        struct vm_area_struct * vma;
        int unmapped_error = 0;
        long error = -EINVAL;

        down_read(&current->mm->mmap_sem);
...
                /* Here vma->vm_start <= start < vma->vm_end. */
                if (end <= vma->vm_end) {
                        if (start < end) {
                                error = mincore_vma(vma, start, end,
                                                        &vec[index]);
                                if (error)
                                        goto out;
                        }
                        error = unmapped_error;
                        goto out;
                }
...
out:
        up_read(&current->mm->mmap_sem);
        return error;
}
```

ソースを確認したところ、sys_mincore() にてメモリ構造体のセマフォを取得した後で、mincore_vma() が呼び出されていることがわかりました。バックトレースからこの mincore_vma() にてページフォルト例外が発生していることがわかります。

どうやら、これがこの問題の原因と考えられます。すなわち、セマフォを取得したまま例外処理で再度同じセマフォを取得しようとしていることでデッドロックになる状況が考えられます。

しかし、読み込みに対するセマフォ操作だけでは、デッドロックは起きません。どこかに書き込みに対するセマフォ操作が存在すると考えられます。

プロセスの詳細を見てみましょう。再度 crash にて ps コマンドを実行します。ここで、ユーザプロセス MYAPL に注目すると、MYAPL がスレッドであることがわかります。

```
crash> ps
...
   2639   2554   3    1022e9920f0    UN   0.0   31248    696  MYAPL
   2640   2554   3    1022e045810    UN   0.0   31248    696  MYAPL
   2641   2554   1    1022e9b4170    UN   0.0       0      0  MYAPL
   2642   2554   0    1022e959850    UN   0.0   31248    696  MYAPL
...
```

それぞれのスレッドについてバックトレースを表示してみます。

```
crash> bt 2640
PID: 2640   TASK: 1022e045810       CPU: 3   COMMAND: "MYAPL"
 #0 [10228c13cb8] schedule at ffffffff805537d7
 #1 [10228c13de0] __down_read at ffffffff80554bbf
 #2 [10228c13e20] do_page_fault at ffffffff80121fcc
 #3 [10228c13f50] error_exit at ffffffff8010f0dd
    RIP: 0000002a957e8793  RSP: 00000000407ff780  RFLAGS: 00010206
    RAX: 0000002a9599b540  RBX: 0000002a95a00020  RCX: 000000000000000c
    RDX: 0000000000002001  RSI: 0000002a95a00768  RDI: 0000000000001000
    RBP: 0000002a95a000b8   R8: 0000000000001e711   R9: 0000002a95a008e0
    R10: 0000000000000003  R11: 0000000000000100  R12: 0000002a95a028f0
    R13: 0000000000002015  R14: 0000000000002011  R15: 0000000000000000
    ORIG_RAX: ffffffffffffffff  CS: 0033  SS: 002b
crash> bt 2641
PID: 2641   TASK: 1022e9b4170       CPU: 1   COMMAND: "MYAPL"
 #0 [1022924bab8] schedule at ffffffff805537d7
 #1 [1022924bbe0] __down_read at ffffffff80554bbf
 #2 [1022924bc20] do_futex at ffffffff8014fd91
 #3 [1022924bd30] sys_futex at ffffffff8015010f
 #4 [1022924bd90] do_exit at ffffffff8013b879
 #5 [1022924be00] get_signal_to_deliver at ffffffff80145512
 #6 [1022924be50] do_signal at ffffffff8010d958
 #7 [1022924bf50] retint_signal at ffffffff8010eb76
    RIP: 0000002a957ea2a4  RSP: 00000000040fff810  RFLAGS: 00000246
    RAX: 0000000000004011  RBX: 0000002a9599b540  RCX: 0000000000004010
    RDX: 0000002a9599b5d8  RSI: 00000000005031d0  RDI: 00000000005031c0
    RBP: 0000000000000000   R8: 00000000005091e0   R9: 00000000005061d0
    R10: 0000000000001001  R11: 0000000000001000  R12: 0000002a9567300b
    R13: 0000002a956799c0  R14: 0000000000001000  R15: 0000000000000003
    ORIG_RAX: ffffffffffffffff  CS: 0033  SS: 002b
crash> bt 2642
PID: 2642   TASK: 1022e959850       CPU: 0   COMMAND: "MYAPL"
 #0 [10228c15d78] schedule at ffffffff805537d7
 #1 [10228c15ea0] __down_write at ffffffff80554b1b
 #2 [10228c15ee0] sys_mprotect at ffffffff8017aa8b
 #3 [10228c15f80] no_syscall_entry_trace at ffffffff8010e539
    RIP: 0000002a95832309  RSP: 00000000417ff780  RFLAGS: 00000297
    RAX: 000000000000000a  RBX: ffffffff8010e539  RCX: 0000000000000004
```

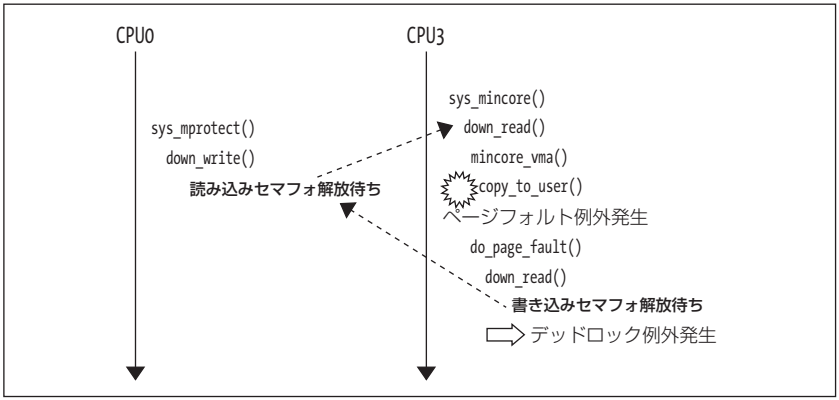

図 5-14　問題のシナリオ

```
RDX: 0000000000000003  RSI: 0000000000024000  RDI: 0000002a95d00000
RBP: 0000000000000003   R8: 00000000ffffffff   R9: 0000000000000000
R10: 0000000000004022  R11: 0000000000000217  R12: 0000000000001000
R13: 0000002a956799c0  R14: 0000000000024000  R15: 0000002a95d00000
ORIG_RAX: 000000000000000a  CS: 0033  SS: 002b
```

pid=2642 のスレッドが down_write() をコールし、書き込みに対するセマフォを要求しています。この down_write() との組み合わせでデッドロックに陥っていると見てよいでしょう。

以上より、この問題のシナリオは**図 5-14** のように推測できます。

この状態に陥ると、問題のプロセスのメモリ構造体にアクセスするプロセスがすべてセマフォ待ちでスリープするようになります。ただし、他のプロセスには影響ありません。

このデッドロック問題の原因は mincore() システムコールの処理で、sys_minicore() にて読み込みセマフォを保持した状態でページフォルト例外を起こす可能性のある処理、すなわち、copy_to_user() が実行されていることです。

ここまでわかったことで問題の再現ができると考えられます。

再現試験

再現用のプログラムを用意します。

カーネルソースを確認し、mmap_sem に対して down_write() を行う処理を探します。その結果、mmap() の処理で down_write() を行っていることがわかりました。

今回の問題は mincore() システムコールと mmap() システムコールをぶつけることで、競合条件を満たせます。

作成したプログラムのソースはこれになります。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <pthread.h>

#define PAGESIZE      (4096)

void *th(void *p)
{
        unsigned char *ptr;

        for (;;) {
                ptr = mmap(NULL, PAGESIZE, PROT_READ|PROT_WRITE,
                                MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
                *ptr = 0;
                munmap(ptr, PAGESIZE);
        }

        return NULL;
}

void do_mincore(void)
{
        unsigned char *vec;

        vec = mmap(NULL, PAGESIZE, PROT_READ|PROT_WRITE,
                        MAP_SHARED|MAP_ANONYMOUS, -1, 0);

        for (;;) {
                if (mincore(vec, PAGESIZE, vec) < 0)
                        perror("mincore");
        }

        munmap(vec, PAGESIZE);
}

int main(int argc, char **argv)
{
        pthread_t tid;
```

```
        pthread_create(&tid, NULL, th, NULL);

        do_mincore();

        return 0;
}
```

この再現プログラムを実行することで、今回の問題現象が発生することが確認できました。

問題対処

まずは、コミュニティに問題対処が存在しないか確認します。

再現プログラムを利用することで、コミュニティのカーネル 2.6.9 で再現することが確認できました。また、最新のカーネルではこの問題を再現させられませんでした。したがって、コミュニティカーネルにおいてこの問題について修正が行われていると考えられます。

対象ファイル mm/mincore.c の履歴を確認することでこの問題に対する修正パッチを発見することができました。

```
commit 2f77d107050abc14bc393b34bdb7b91cf670c250
Author: Linus Torvalds <torvalds@woody.osdl.org>
Date:   Sat Dec 16 09:44:32 2006 -0800

    Fix incorrect user space access locking in mincore()

    Doug Chapman noticed that mincore() will doa "copy_to_user()" of the
    result while holding the mmap semaphore for reading, which is a big
    no-no.  While a recursive read-lock on a semaphore in the case of a page
    fault happens to work, we don't actually allow them due to deadlock
    schenarios with writers due to fairness issues.

    Doug and Marcel sent in a patch to fix it, but I decided to just rewrite
    the mess instead - not just fixing the locking problem, but making the
    code smaller and (imho) much easier to understand.

    Cc: Doug Chapman <dchapman@redhat.com>
    Cc: Marcel Holtmann <holtmann@redhat.com>
    Cc: Hugh Dickins <hugh@veritas.com>
    Cc: Andrew Morton <akpm@osdl.org>
    Signed-off-by: Linus Torvalds <torvalds@osdl.org>
```

このパッチを評価中のカーネルにバックポートして適用することで、作成した再現プログラムにおいて問題現象が発生しないことが確認できました。

まとめ

本 Hack ではカーネルセマフォのデッドロック事例とその解析について説明しました。今回の解析におけるポイントは下記です。

- 問題現象の詳細状況確認
- クラッシュダンプによる解析データの収集
- 再現プログラムによる問題の再現

参考文献

- AMD64 Application Binary Interface
  http://www.x86-64.org/documentation/abi.pdf

—— Hiroshi Shimamoto

HACK #40 リアルタイムプロセスのストール

リアルタイムのユーザプログラムがストールしたことをダンプから証明します。

通常のプロセスがストールしたときのデバッグは「**アプリケーションのストール（無限ループ編）**」**[HACK #32]** で紹介しました。本 Hack ではリアルタイムプロセスがストールしたときのデバッグを紹介します。リアルタイムプロセスがストールすると CPU が占有され、システム全体がストールし重大な障害となります。本 Hack では簡単な再現 TP を使いますが、実際の問い合わせであったものです。カーネルは 2.6.9 になります。

リアルタイムプロセスとは通常のプロセスよりも優先度が高いプロセスです。同じかより高い優先度のリアルタイムプロセスがいなければ、自らスリープ（プリエンプション）しない限り CPU を使用し続けます。

まずは再現

まずは再現をさせてみます。以下は無限ループするだけのシェルスクリプト loop.sh です。chrt コマンドは引数に与えたプロセスのスケジューリングポリシーをリアルタイムに設定します。99 は優先度です。リアルタイムで一番高い数字になります。

ダンプを採取するため事前に IPMI watchdog を有効にしてください（詳細は**「IPMI watchdog timer により、フリーズ時にクラッシュダンプを取得する」[HACK #22]** を参照してください）。今回の現象はユーザアプリケーションのストールのため、NMI watchdog によるダンプの採取はできません。

```
# cat loop.sh
#!/bin/bash

while [ 0 ] ; do
    :
done
# chrt 99 ./loop.sh &
# ps aux
USER   PID %CPU %MEM    VSZ   RSS TTY      STAT START TIME COMMAND
...
root 28333  100  0.0 64624 1028 pts/0  R    23:10 4:48 /bin/bash ./loop.sh
         /* ^^^ CPUの使用率が100% */
...
```

無限ループで CPU を消費し続けているため CPU 使用率が 100% になっています。これで CPU の 1 つを占有したことになります。今回使用したマシンは CPU が 2 つあるため、もう 1 つ loop.sh を実行します。これで 2 つの CPU が loop.sh に占有されます。

```
# chrt 99 ./loop.sh
```

これで loop.sh が 2 つの CPU を占有してシステムがストールします。キー入力もできなくなります。そのため IPMI watchdog のタイムアウトでダンプ採取になります。

最初はバックトレースの確認

実際にバグが検出されたときはここから解析が始まります。ダンプが取れましたので、最初のメッセージを見てみます。最初に注目するのは COMMAND（プロセス）です。

```
# crash vmlinux vmcore
...
      PANIC: ""
        PID: 4223
    COMMAND: "loop.sh"       /* パニックしたときに動作していたプロセス名 */
       TASK: 1007e3087f0  [THREAD_INFO: 10036144000]
        CPU: 0
...
```

パニックしたときに動作していたのは loop.sh だとわかります。

バックトレースの確認（e1000 編）

次に bt コマンドでバックトレースを見てみます。

```
crash> bt
PID: 4223   TASK: 1007e3087f0       CPU: 0   COMMAND: "loop.sh"
...
 #8 [ffffffff8045e3b0] error_exit at ffffffff80110d91
    [exception RIP: panic+211]
    RIP: ffffffff8013797a  RSP: ffffffff8045e468  RFLAGS: 00010086
...
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0000
 #9 [ffffffff8045e460] panic at ffffffff80137966
#10 [ffffffff8045e480] e1000_alloc_rx_buffers_ps at ffffffffa009bb77
#11 [ffffffff8045e4f0] sock_def_write_space at ffffffff802ad1d4
#12 [ffffffff8045e540] ipmi_wdog_pretimeout_handler at ffffffffa01c4892
#13 [ffffffff8045e550] ipmi_smi_watchdog_pretimeout at ffffffffa01ad927
#14 [ffffffff8045e570] handle_flags at ffffffffa01b51a0
#15 [ffffffff8045e590] smi_event_handler at ffffffffa01b54ca
...
```

panic() の直前呼ばれた関数は e1000 ドライバの関数のように見えます。これについて少し調べてみます。まずは e1000_alloc_rx_buffers_ps at ffffffffa009bb77 を dis コマンドで確認します。

```
crash> dis e1000_alloc_rx_buffers_ps
...
0xffffffffa009baf4 <e1000_alloc_rx_buffers_ps+381>:     add    $0x12,%edi
0xffffffffa009baf7 <e1000_alloc_rx_buffers_ps+384>:     callq  0xffffffff802adb53 <alloc_skb> ————❶
0xffffffffa009bafc <e1000_alloc_rx_buffers_ps+389>:     test   %rax,%rax
...
0xffffffffa009bb70 <e1000_alloc_rx_buffers_ps+505>:     inc    %ebx
0xffffffffa009bb72 <e1000_alloc_rx_buffers_ps+507>:     callq  0xffffffff80120294 <dma_map_single> ——❷
0xffffffffa009bb77 <e1000_alloc_rx_buffers_ps+512>:     mov    0x18(%rsp),%rdx
...
```

逆アセンブルでは e1000_alloc_rx_buffers_ps+512 とありますが、これはレジスタに次の命令が入るため、パニックしたときは 1 つ前の命令が実行されていることになります。その

ため 1 行上の e1000_alloc_rx_buffers_ps+507(2) の call 命令をたどります。dma_map_single() を呼んでいるのでソースコードと照らし合わせて見てみます。逆アセンブラの❶とソースコードの①が対応していますので、e1000_alloc_rx_buffers_ps+507 ❷は以下の②になります。

```
[drivers/net/e1000/e1000_main.c]
  e1000_alloc_rx_buffers_ps()
  …
        skb = netdev_alloc_skb(netdev,
                               adapter->rx_ps_bsize0 + NET_IP_ALIGN); ————①

        if (unlikely(!skb)) {
  …
        buffer_info->skb = skb;
        buffer_info->length = adapter->rx_ps_bsize0;
        buffer_info->dma = pci_map_single(pdev, skb->data, ————————②
                                          adapter->rx_ps_bsize0,
                                          PCI_DMA_FROMDEVICE);
  …
```

```
[include/asm-generic/pci-dma-compat.h]
  static inline dma_addr_t
  pci_map_single(struct pci_dev *hwdev, void *ptr, size_t size, int direction)
  {
         return dma_map_single(hwdev == NULL ? NULL : &hwdev->dev, ptr, size, (enum dma_data_direction)
  direction);
  }
```

pci_map_single() は inline で定義されているためダンプのバックトレースで call 0xffff… <pci_map_single> のようには表示されません。pci_map_single() は dma_map_single() を呼んでいます。

```
[arch/x86_64/kernel/pci-gart.c]
  dma_addr_t dma_map_single(struct device *dev, void *addr, size_t size, int dir)
  {
        unsigned long phys_mem, bus;

        BUG_ON(dir == DMA_NONE);

        if (swiotlb)
  …
```

確かにe1000_alloc_rx_buffers_ps() -> BUG() -> panic()のルートがありました。このような場合であればバックトレースにdma_map_single+16(3)と残っているはずですが、ありませんのでe1000_alloc_rx_buffers_ps()でパニックしたとは考えにくいです。

```
crash> dis dma_map_single
0xffffffff80120294 <dma_map_single>:    push   %rbp
0xffffffff80120295 <dma_map_single+1>:  cmp    $0x3,%ecx
0xffffffff80120298 <dma_map_single+4>:  mov    %rdi,%rbp
0xffffffff8012029b <dma_map_single+7>:  mov    %rdx,%rdi
0xffffffff8012029e <dma_map_single+10>: push   %rbx
0xffffffff8012029f <dma_map_single+11>: push   %rax
0xffffffff801202a0 <dma_map_single+12>: jne    0xffffffff801202ae <dma_map_single+26>
0xffffffff801202a2 <dma_map_single+14>: ud2a
0xffffffff801202a4 <dma_map_single+16>: mov    $0x70,%dh ──────────────────────❸
0xffffffff801202a6 <dma_map_single+18>: xor    0xffffffffffffffff(%rax),%al
…
```

バックトレースにはsock_def_write_space()もありますが、panic()を呼ぶルートはありませんでした。

バックトレースの確認（IPMI編）

バックトレースの#12にipmi_wdog_pretimeout_handler()があります。これについて調べます。

```
crash> dis ipmi_wdog_pretimeout_handler
0xffffffffa01c486f <ipmi_wdog_pretimeout_handler>:      push   %rax
0xffffffffa01c4870 <ipmi_wdog_pretimeout_handler+1>:    cmpb   $0x0,19226(%rip)        # 0xffffffffa
01c9391
0xffffffffa01c4877 <ipmi_wdog_pretimeout_handler+8>:    je     0xffffffffa01c48e3
0xffffffffa01c4879 <ipmi_wdog_pretimeout_handler+10>:   movzbl 19216(%rip),%eax       # 0xffffffffa
01c9390
0xffffffffa01c4880 <ipmi_wdog_pretimeout_handler+17>:   cmp    $0x1,%al
0xffffffffa01c4882 <ipmi_wdog_pretimeout_handler+19>:   jne    0xffffffffa01c4892
0xffffffffa01c4884 <ipmi_wdog_pretimeout_handler+21>:   mov    $0xffffffffa01c4e6f,%rdi
0xffffffffa01c488b <ipmi_wdog_pretimeout_handler+28>:   xor    %eax,%eax
0xffffffffa01c488d <ipmi_wdog_pretimeout_handler+30>:   callq  0xffffffff801378a7 <panic> ──────❹
0xffffffffa01c4892 <ipmi_wdog_pretimeout_handler+35>:   cmp    $0x2,%al
…
```

❹でpanic()を呼んでいます。ソースでは以下の④になります。

```
[drivers/char/ipmi/ipmi_watchdog.c]
static void ipmi_wdog_pretimeout_handler(void *handler_data)
{
        if (preaction_val != WDOG_PRETIMEOUT_NONE) {
                if (preop_val == WDOG_PREOP_PANIC)
                        panic("Watchdog pre-timeout"); ―――――――――――――④-1
                else if (preop_val == WDOG_PREOP_GIVE_DATA) {
                        spin_lock(&ipmi_read_lock);
                        data_to_read = 1;
…
```

ipmi_wdog_pretimeout_handler() で panic()（④-1）を呼んでいます。panic() の引数には文字列「Watchdog pre-timeout」を入れています。そこで log コマンドによりカーネル内のログバッファを確認します。

```
crash> log
…
Kernel panic - not syncing: Watchdog pre-timeout ―――――――――――――――――④-2
----------- [cut here ] --------- [please bite here ] ---------
Kernel BUG at panic:75
invalid operand: 0000 [1] SMP
…
Call Trace:<IRQ> <ffffffffa009bb77>{:e1000:e1000_alloc_rx_buffers_ps+512}
    <ffffffff802ad1d4>{sock_def_write_space+18} <ffffffffa01c4892>{:ipmi_watchdog:ipmi_wdog_pretimeout_
handler+35}
   <ffffffffa01c486f>{:ipmi_watchdog:ipmi_wdog_pretimeout_handler+0}
   <ffffffffa01ad927>{:ipmi_msghandler:ipmi_smi_watchdog_pretimeout+53}
   <ffffffffa01b51a0>{:ipmi_si:handle_flags+87} <ffffffffa01b54ca>{:ipmi_si:smi_event_handler+490}
   <ffffffffa01b58ad>{:ipmi_si:smi_timeout+72} <ffffffffa01b5865>{:ipmi_si:smi_timeout+0}
   <ffffffff80140115>{run_timer_softirq+356} <ffffffff8013c7c8>{__do_softirq+88}
   <ffffffff8013c871>{do_softirq+49} <ffffffff80110bf5>{apic_timer_interrupt+133}
    <EOI>
…
```

ソースコードにある文字列 Watchdog pre-timeout（④-2）が表示されています。このダンプは IPMI watchdog でパニックしたようです。

この時点でわかったことをまとめます。

1. 今回の現象は IPMI watchdog のタイムアウトでパニックしている。またそのことからカーネルかユーザアプリケーションでストールしている可能性が高い

2. IPMI watchdog であることから割り込み禁止状態ではない

実行中プロセスの確認

ストールの場合、原因は大きく分けて4つです。カーネルとユーザアプリケーションでそれぞれデッドロックと無限ループがあります。

e1000の関数を再度調査しましたが、無限ループやストールするようなコードはありませんでした。そこでアプリケーションのストールに着目して見てみます。まずはpsコマンドで実行中のプロセスを確認します。

```
crash> ps
   PID    PPID  CPU       TASK        ST  %MEM     VSZ    RSS  COMM
...
>  4223   4046   0     1007e3087f0    RU   0.1   53532   1168  loop.sh
>  4224   4046   1     10057b147f0    RU   0.1   53532   1168  loop.sh
crash>
```

実行中のプロセスはloop.shです。これらのプロセスについて情報を集めます。taskコマンドで優先度とスケジューリングポリシーを確認します。

```
crash> task 4223 | grep prio
  prio = 0,
  static_prio = 120,
  rt_priority = 99,
crash> task 4224 | grep prio
  prio = 0,
  static_prio = 120,
  rt_priority = 99,
crash> task 4223 | grep policy -w
  policy = 2,
crash> task 4224 | grep policy -w
  policy = 2,
```

loop.shは2つとも優先度が99でラウンドロビンのリアルタイムプロセスであることがわかります。どのぐらいのCPU時間を消費していたか見てみます。アプリケーションなのでutimeを見ます。utimeとはユーザ空間で消費しているCPU時間になります（time(1)コマンドの"user"の値になります）。スケジューラがこの値を更新しています（カーネル空間はstimeになります）。

```
crash> task 4223 | grep utime
  utime = 50303,
crash> task 4224 | grep utime
  utime = 50252,
```

utime を見るには ps コマンドが便利です。他のものと比較するため ps コマンドで CPU 消費時間を見てみます。

```
crash> ps -t
...
PID: 4044   TASK: 10037d25030      CPU: 1   COMMAND: "sshd"
    RUN TIME: 00:27:31
  START TIME: 266
   USER TIME: 203
 SYSTEM TIME: 283
PID: 4046   TASK: 100793aa7f0      CPU: 1   COMMAND: "bash"
    RUN TIME: 00:27:30
  START TIME: 267
   USER TIME: 53
 SYSTEM TIME: 30
...
PID: 4223   TASK: 1007e3087f0      CPU: 0   COMMAND: "loop.sh"
    RUN TIME: 00:00:55
  START TIME: 1862
   USER TIME: 50303
 SYSTEM TIME: 2628
PID: 4224   TASK: 10057b147f0      CPU: 1   COMMAND: "loop.sh"
    RUN TIME: 00:00:54
  START TIME: 1863
   USER TIME: 50252
 SYSTEM TIME: 2651
...
```

utime は USER TIME になり、単位はミリ秒です。今回 IPMI watchdog は pretimeout が 30、timeout を 90 にしたのでストールが 60 秒続くとパニックしますが、watchdog デーモンの interval が 10 秒なので実際にストールしてから 50 ～ 60 秒の間でパニックします。sshd、bash は 1 秒も動作していないのに対して、loop.sh はユーザ空間だけでも 50 秒動作しています。そのためスケジュールせずに動作し続けている、つまり無限ループしていると判断できます。

カーネルダンプからアプリケーションの解析はここまでになります。アプリケーションの原因を解析するには、strace（「**strace を使って、不具合原因の手がかりを見つける**」**[HACK #43]** 参照）、gdb（「**デバッガ（GDB）の基本的な使い方（その 1）**」**[HACK #5]** 参照）を使って再現させるなど別の手段が必要になります。また loop.sh は実行されてすぐにストールしたため 55 秒で watchdog が動作していますが、長時間動作し続けていたプロセスがストールした場合は他のプロセスと比較するなど別途検討する必要はありますが、無限ループであれば utime が不自然な値になるはずです。ただしループ中にスケジュールする場合、この数値だけを過信できないことに注意が必要です。

まとめ

カーネルダンプから utime を見ることでリアルタイムプロセスの無限ループを見つけた例を紹介しました。

実際に同じような現象が発生したときは e1000 の関数がバックトレースに複数出力され、e1000 でパニックしたと問い合わせがありました。今回はそれを再現するためネットワーク負荷をかけて、意図的に e1000 の関数が表示されるようにしました。今回のようにスタックに以前の情報が残ることがあり、その状態でダンプが採取されるとバックトレースに関係のないシンボルが表示されることがあります。

—— Naohiro Ooiwa

HACK #41 動作がスローダウンする不具合

カーネルのバージョンアップの際に発生した MTD デバイスへの書き込み速度低下をデバッグした事例について紹介します。

カーネルバージョンアップ後の異変

ここで紹介するのは、ある MTD デバイス（Flash メモリ）が搭載されているシステムのカーネルを 2.6.9 から 2.6.18 にバージョンアップした際の不具合です。以下のコマンドで MTD デバイスに 128KB の書き込みを行ったところ、以前なら、数秒で終了していた操作が、数分待っても終了しなくなりました。また、エラーメッセージも出力されません。

```
# dd if=a.dat of=/dev/mtd0 bs=131072 count=1
```

問題の原因リストアップと絞り込み

まず、何が起こったのか理解するため、次の 2 点を確認しました。問題が発生した時、これらを確認するだけも、その問題に関する手がかりが得られる場合が多くあるからです。

- 画面にエラーメッセージが表示されていないか
- /var/log/messeges や /var/log/syslog にプロセス、デーモン、カーネルのエラーや警告等のメッセージがないか

しかし、筆者が確認したところ、それらの情報はありませんでした。とは言え、何もメッセージがないというのも、ひとつの情報です。つまり、プロセスやカーネルは、実行している処理をエラーとは認識していないと言えます。次に、筆者は、次のようなことが発生していないか考えてみました。

a. どこかで無限ループ（ビジーループ）している
b. 何らかのエラーが発生してリトライを繰り返している
c. プロセスが何らかのイベント待ちになっている（シグナル待ち、スリープしている等）
d. デッドロックしている
e. SIGSTOP などによりプロセスが停止している

これらは、次の表のように CPU 使用率と、プロセスの状態を見ることで、大まかに、切り分けることができます。

表 5-2　CPU 使用率とプロセスの状態からの問題の特定

可能性	CPU 使用率	プロセス状態†
a.	高い（ほぼ 100%）	R
b.	さまざまなケースあり	R または S
c.	低い（ほぼ 0%）	S または D
d.	低い（ほぼ 0%）	S または D
e.	0%	T

†　R：Run 状態、S：割り込み可能なスリープ、D：割り込み不可能なスリープ、T：停止状態

top コマンドを使い、CPU 使用率を調べました。その結果、CPU 使用率はほぼ 0% でした。次に、ps コマンドでプロセス（dd）の状態を確認しました。すると、下記のように割り込み不可能なスリープ状態になっています（左から 3 番目の項目がプロセスの状態です）。

```
# ps ax | grep dd
25921 pts/3    D+     0:00 dd if /dev/mtd0 of a.dat bs 131072 count 1
```

何回かこのコマンドを繰り返しても、ずっとDのままでした。つまり、上記のc.やd.が疑われます。これらの問題（例えば、デッドロック）は、プロセスのコードでもカーネルのコードでも発生します。そこで、次にこの問題を引き起こしている原因は、プロセス（dd）のコードか、カーネルのコードかを調べました。

これには、GDBを使ってプロセスにアタッチし、バックトレースを取りました。

```
# gdb -p `pidof dd`
...
(gdb) bt
#0  0x000000309eec0e60 in __write_nocancel () from /lib64/libc.so.6
#1  0x0000000000401fa7 in iwrite (fd=1, buf=0x10169000 "", size=512) at dd.c:782
#2  0x000000000040200b in write_output () at dd.c:808
#3  0x0000000000403425 in main (argc=<value optimized out>, argv=<value optimized out>) at dd.c:1294
```

本来は、上記に表示されている各関数の詳細な動作を確認しないと何とも言えないのですが、このバックトレースには、デッドロックやスリープを行うような関数（pthread_mutex_lock()、wait()、sleep()のような関数）は、含まれていません。また、ddの場合は、比較的自明ですが、下記のようにシングルスレッドプロセスなので、デッドロックが起きる可能性は比較的低いと考えられます。

```
(gdb) i thr
  1 Thread 46912496307920 (LWP 10070)  0x000000309eec0e60 in __write_nocancel () from /lib64/libc.so.6
```

また、問題がプロセスかカーネルかを調べる、もうひとつの調査方法としてstraceを用いました。straceは、プロセスのシステムコールをトレースするツールです。straceは、実行中のプロセスに対しても下記のようにトレースできます。straceの詳細については、**「straceを使って、不具合原因の手がかりを見つける」[HACK #43]**を参照してください。

```
# strace -t -p `pidof dd`
Process 10070 attached - interrupt to quit
14:49:14 read(0, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 512) = 512
14:49:14 write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 512) = 512
<< この状態で13秒停止 >>
14:49:27 read(0, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 512) = 512
14:49:27 write(1, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 512
<< この状態で13秒停止 >>
) = 512
...
<< 以後、read()とwrite()が繰り返される >>
```

上記のトレース結果からわかることは、write システムコールでの 512B の書き込みに約 13 秒を要しているということです。直感的にこれは、非常に長い時間です。HDD では、書き込みスループットは、おおよそ数 MB/s から数十 MB/s ですし、いま書き込みを行っている MTD デバイスでも数 100KB/s 程度です。そのため、本来 512B の書き込みは、ミリ秒のオーダーで完了するはずです。また、注目すべきことに、これほど長い時間を要しているにも関わらず、システムコールそのものは成功しています。

上記の GDB と strace を用いた調査からはプロセスのコードよりも、カーネル内部、もっと言えば、write システムコールの延長上で問題が起こっている可能性が高いと考えられます。

プロセスが D 状態（割り込み不可能なスリープ状態）の時、上記のように GDB や strace を使ってアタッチすると、GDB や strace がフリーズしたようになることがあります。これらのコマンドは起動直後、アタッチ対象のプロセスがトレース状態になることを待つのですが、そのプロセスがカーネル内でのデッドロック等により、動作できない状態になっていると、いつまでたってもトレース状態にならないために発生します。このような場合は、原因がカーネル内部にある可能性が高いので、GDB や strace での調査はスキップし、次の crash を使った調査に進んだほうがいいでしょう。

そこで、crash コマンドでカーネル内部の様子を確認しました。strace での調査どおり、write システムコール (c) が呼ばれており、そこから MTD デバイス用の書き込み関数 mtd_write(B) がコールされています。

```
crash> bt -t 4313
PID: 4313   TASK: ffff81022760d040  CPU: 1   COMMAND: "dd"
              START: thread_return (schedule) at ffffffff80061f29
  [ffff81021b073cf0] schedule_timeout at ffffffff80062839
  [ffff81021b073d10] process_timeout at ffffffff8009409f
  [ffff81021b073d28] inval_cache_and_wait_for_operation at ffffffff883d31d6
  [ffff81021b073d40] msleep at ffffffff80094768                    ——————————(A)
  [ffff81021b073d50] inval_cache_and_wait_for_operation at ffffffff883d3332
  [ffff81021b073d88] default_wake_function at ffffffff8008986e
  [ffff81021b073da0] __wake_up at ffffffff8002e04d
  [ffff81021b073de0] do_write_oneword at ffffffff883d5ad8
  [ffff81021b073e30] cfi_intelext_write_words at ffffffff883d6e05
  [ffff81021b073e80] mtd_write at ffffffff883b34ef                 ——————————(B)
  [ffff81021b073eb0] do_mmap_pgoff at ffffffff8000dc60
  [ffff81021b073f10] vfs_write at ffffffff80016233
```

```
[ffff81021b073f40] sys_write at ffffffff80016b00 ————————————(C)
[ffff81021b073f80] system_call at ffffffff8005c116
…
```

その先を見ていくと、(A) の msleep() によってスリープしていることがわかります。なぜ、ここでスリープしているのかを知るためには、ソースを読む必要があります。そこで、ソースを読んだ結果、次のようなことがわかりました。この msleep() は、inval_cache_and_wait_for_operation() という関数から呼ばれています。inval_cache_and_wait_for_operation() は、do_write_oneword() という関数の中で、1 ワード（このチップの場合 2B）を書き込んだ直後に呼び出され、その書き込みが完了するのを待つ関数です。また、2.6.9 カーネルの相当する部分を調べると、この関数は存在せず、別のアルゴリズムで、書き込みの完了を待っているようです。

ソースコードの調査

```
for (;;) {
    status = map_read(map, cmd_adr); ————————————①
    if (map_word_andequal(map, status, status_OK, status_OK)) ———②
        break;

    if (!timeo) { ————————————————————————③
        map_write(map, CMD(0x70), cmd_adr);
        chip->state = FL_STATUS;
        return -ETIME;
    }

    /* OK Still waiting. Drop the lock, wait a while and retry. */
    spin_unlock(chip->mutex);
    if (sleep_time >= 1000000/HZ) { ————————————④
        /*
         * Half of the normal delay still remaining
         * can be performed with a sleeping delay instead
         * of busy waiting.
         */
        msleep(sleep_time/1000); ————————————⑤
        timeo -= sleep_time;
        sleep_time = 1000000/HZ;
    } else { ————————————————————————⑥
```

```
            udelay(1);
            cond_resched();
            timeo--;
        }
        spin_lock(chip->mutex);

        if (chip->state != chip_state) {
            /* Someone's suspended the operation: sleep */
            <<省略>>
        }
    }
```

①の部分で、Flash メモリデバイスから書き込みが完了したかどうかを読み取り、②で書き込み完了か否かの判定をしています。もし、書き込みが完了していれば、for ループを出て、この関数から返ります。③は、タイムアウトの判定です。いつまでも書き込みが完了しないと、エラーとしてこの関数を終了します。ソースを確認すると、この関数を呼び出している do_write_oneword() は、ここでエラーが発生した場合、その旨のカーネルメッセージを表示します。dd コマンド実行中に、そのようなカーネルメッセージは表示されていないので、ここでタイムアウトしている可能性はないと考えられます。

④と⑥のブロックは、書き込み完了を一定時間待つためのルーチンです。④のブロックが実行されるか、⑥のブロックが実行されるかは、sleep_time という変数の値によって決まります。sleep_time は、⑤で 1000 で割って、msleep() の引数として使われているので、sleep_time は、マイクロ秒で指定されるスリープ時間と考えられます。このシステムでは、HZ は 1000 なので④は、sleep_time が、1000 マイクロ秒以上かどうかを判断しています。スリープ時間によって、処理を分岐している理由は、Linux のスリープ（タイマ）の精度が、1000 マイクロ秒（1 ミリ秒）であるためと思われます。つまり、スリープ時間が 1 ミリ秒以上であれば、msleep() などのスリープ関数で、ほぼ指定した時間の経過後に、プロセスを起床させることができます。しかし、それ以下であれば、ビジーループ関数の udelay() などを使って、時間を調整せざるをえません。

では、いま sleep_time は、いくつに設定されているのでしょうか。これは、上記ソースの先頭に記述されているように、chip_op_time という値を 1/2 にしたものです。chip_op_time は、ソースを調べていくと、フラッシュメモリのチップの情報を保持している構造体 struct flchip の word_write_time メンバからコピーされたものであることがわかりました。さらにソースを確認すると、この word_write_time メンバは、初期値として、50000（単位はマイクロ秒）が与えられていました。つまり、sleep_time の値、すなわちスリープ時間は、その 1/2 の 25 ミリ秒ということになります。

仮説の立案と検証

ここで考えられる可能性は、1ワード（2B）のデータが書き込まれた直後、①でまだ書き込み中であることがデバイスから返されたので、25ミリ秒のスリープ⑤が、実行されているということです。1回あたりのスリープ時間は、ごく短いですが、128KBの書き込みでは、64 × 1024回のスリープすることになります。時間にすると約30分であり、数分待っても書き込みが終了しなかったこととつじつまが合います。

だとすれば、本当にここで25ミリ秒も待つ必要があるのでしょうか。というのは、2.6.9カーネルでは、はるかに短時間で書き込みが終了していたので、待ち時間は、もっと短くてもよいはずです。そこで、このFlashメモリデバイスのデータシートを見て、1ワードの書き込みに要する時間を調べました。その結果、1ワードの書き込み時間は平均で10マイクロ秒、最大で200マイクロ秒であることがわかりました。そのため、1ワードの書き込みは、10マイクロ秒程度で完了しているのに、その後延々と25ミリ秒（250000マイクロ秒）も待ってから、完了したかどうかを確かめていると考えられます。

なお、Flashメモリデバイスの型番は、MTDドライバの初期化時に表示される下記のカーネルメッセージから、M50FW080ということがわかります。

```
kernel: Found: ST M50FW080
```

また、このデバイスを含め、多くのFlashメモリデバイスは、ベンダのWebページでデータシートが公開されています。

この仮説を検証するため、word_write_timeの初期値を最大書き込み時間よりも少し長い256マイクロ秒（sleep_timeはその1/2の128マイクロ秒）に設定して、カーネルをビルドしました。ビルド後のカーネルで、ddコマンドによる書き込みを行うと2.6.9カーネルの時と同じように数秒で終了しました。そのため、やはり、待ち時間の初期値が長すぎたことが原因であることがわかりました。

まとめ

カーネルのバージョンアップの際に発生したMTDデバイスへの書き込み速度低下をデバッグした事例を紹介しました。

—— Kazuhiro Yamato

HACK #42 CPU 負荷が高くなる不具合

VLAN を使ったネットワークで TCP 通信を行うとハードウェアによるチェックサム計算機能が動作しない問題がありました。この問題についてのデバッグを紹介します。

Intel のネットワークデバイスは TCP パケットのチェックサムを計算する機能を持っています。これにより CPU 負荷を軽減できます。Linux カーネルはこの機能がネットワークデバイスにあるか判断し、あればハードウェアで計算するので、カーネル（ソフトウェア）でチェックサムの計算は行いません。通常のネットワーク通信であればハードウェアのチェックサム計算機能が使われますが、VLAN デバイスで TCP の通信を行うと、カーネルでチェックサムの計算をしていることがわかりました。これは実際の問い合わせでわかった問題です。

本 Hack では発見方法から修正までの説明を行います。カーネルは 2.6.18 の RedHat 系ディストリビューションです。

再現の準備

まず VLAN デバイスを作成します。今回は eth3.510 を作成します。vconfig コマンドでも作成できますが、起動時に作成されるように設定ファイルを編集します。

```
# vi /etc/sysconfig/network-scripts/ifcfg-eth3.510
DEVICE=eth3.510
…
VLAN=yes
…
```

必ず eth3 とネットワークのセグメントを分けます。

```
# ifconfig
…
eth3      Link encap:Ethernet  HWaddr 00:15:17:3A:61:09
          inet addr:192.168.1.200  Bcast:192.168.1.255  Mask:255.255.255.0
…
eth3.510  Link encap:Ethernet  HWaddr 00:15:17:3A:61:09
          inet addr:192.168.0.200  Bcast:192.168.0.255  Mask:255.255.255.0
…
#
```

ネットワークの性能を計測するため、マシンは TCP パケットを送信する sender、受信する receiver の 2 台を用意しました。環境は**図 5-15** の構成になります。

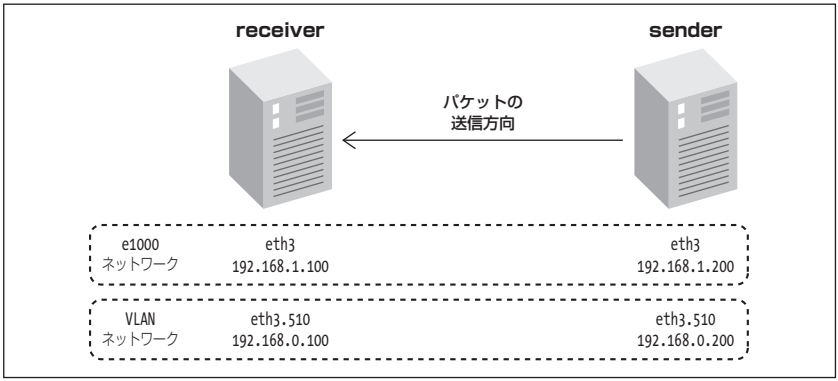

図 5-15 ネットワーク性能測定の環境

eth3 と eth3.510 の設定をしたら、receiver と sender で通信できるか ping コマンドなどで確認します。

nuttcp でスループットの計測

通常のネットワークデバイスと VLAN デバイスでスループットを計測します。今回は nuttcp を使います。nuttcp は TCP/UDP のネットワークテストツールでスループットを測定できます。

```
# wget -t0 -c http://www.lcp.nrl.navy.mil/nuttcp/nuttcp-5.5.5.tar.bz2
# tar jxvf nuttcp-5.5.5.tar.bz2
# cd nuttcp-5.5.5
# gcc -O2 -o nuttcp nuttcp-5.5.5.c
```

receiver では以下のオプションで nuttcp を起動します。nuttcp のサーバが起動します。

```
[receiver]# ./nuttcp -S
```

sender でも nuttcp を実行し、パケットを送信します。-n オプションで合計の送信データサイズを 1GB にし、フラグメントをするように -l オプションでデータを 1500 バイトを書き込むように設定します。

```
[sender]# ./nuttcp -n1G -l1500 192.168.1.100          /* 通常のネットワーク */
 1023.9987 MB /   9.12 sec =  941.3975 Mbps 12 %TX 18 %RX
[sender]# ./nuttcp -n1G -l1500 192.168.0.100          /* VLAN ネットワーク */
 1023.9987 MB /   9.15 sec =  938.5309 Mbps 22 %TX 19 %RX
```

最初の数値は送信したデータサイズです。オプションで設定しているので 1GB 送信しています。その次の値は送信が終わるまでの時間です。その次にスループット（Mbps）の値が表示されます。VLAN ネットワークのほうがスループットは少し低いです。また %TX と %RX は送信プロセス（sender）と受信プロセス（receiver）の CPU 使用率です。ps コマンドなど通常の CPU 使用率はプロセスの生存期間中に実行に利用された CPU 時間のパーセンテージですが、nuttcp は送受信直前から送受信終了直後の時間と、そのときに使用された CPU（ユーザ＋カーネル）時間でパーセンテージを割り出しています。送信プロセスの CPU 使用率（%TX）は通常のネットワークで 12%、VLAN ネットワークは 22% です。スループットはあまり差がなかったのに対し、CPU 使用率は 10% 増えています。

そこで oprofile を使い、どこにオーバーヘッドがあるのか比較してみます。

oprofile によるオーバーヘッドの確認

oprofile を使用しますが、シンボルを解決するために vmlinux が必要です。kernel-debuginfo の RPM パッケージを展開すると /usr/lib/debug/lib/modules/2.6.18/vmlinux がありますのでこれを opcontrol コマンドで指定します。

nuttcp で送信している状態にして、opcontrol コマンドを実行します（実際にはスクリプトで実行しています）。

```
[sender]# opcontrol --init
[sender]# opcontrol --start --vmlinux=/boot/vmlinux-2.6.18
[sender]# ./nuttcp -T20s -l1500 192.168.0.100 &       /* VLAN ネットワーク */
[sender]# sleep 10
[sender]# opcontrol --stop
```

まずは VLAN ネットワークのプロファイルを行いました。opreport コマンドで簡単な結果がわかります。以下ではカーネルが一番多く動作しており、次に多いのは e1000e です。

```
[sender]# opreport
...
  samples|      %|
------------------
    59056 81.6808 vmlinux-2.6.18
    10712 14.8158 e1000e
     1053  1.4564 nuttcp
      869  1.2019 libc-2.5.so
      243  0.3361 oprofiled
      119  0.1646 bash
```

```
       87  0.1203 8021q
       50  0.0692 oprofile
…
```

opreport コマンドでもう少し詳しい情報を見てみます。

```
[sender]# opreport -l
…
samples  %        app name        symbol name
11049    15.2819  vmlinux-2.6.18  csum_partial_copy_generic
10712    14.8158  e1000e          (no symbols) ———①
3093      4.2779  vmlinux-2.6.18  tcp_sendmsg
2800      3.8727  vmlinux-2.6.18  kfree
2627      3.6334  vmlinux-2.6.18  tcp_init_tso_segs
2140      2.9598  vmlinux-2.6.18  skb_clone
2133      2.9502  vmlinux-2.6.18  kmem_cache_free
1866      2.5809  vmlinux-2.6.18  tcp_ack
1805      2.4965  vmlinux-2.6.18  cache_grow
1554      2.1493  vmlinux-2.6.18  mwait_idle
1539      2.1286  vmlinux-2.6.18  tcp_v4_rcv
1473      2.0373  vmlinux-2.6.18  tcp_transmit_skb
1383      1.9128  vmlinux-2.6.18  __kfree_skb
1382      1.9115  vmlinux-2.6.18  eth_header
1166      1.6127  vmlinux-2.6.18  __alloc_skb
1139      1.5754  vmlinux-2.6.18  dev_queue_xmit
1072      1.4827  vmlinux-2.6.18  cache_alloc_refill
1047      1.4481  vmlinux-2.6.18  ip_queue_xmit
1004      1.3886  vmlinux-2.6.18  tcp_v4_send_check
949       1.3126  vmlinux-2.6.18  system_call

…
```

①を見るとシンボル名は (no symbols) となっています。そのため -p オプションを使い、シンボル解決します。以下のコマンドを実行するとウォーニングが出力されますが、この場合は特に問題ありません。

```
[sender]# opreport -l -p /lib/modules/2.6.18/kernel/
…
samples  %        image name      app name        symbol name
11049    15.2819  vmlinux-2.6.18  vmlinux-2.6.18  csum_partial_copy_generic
```

```
3093      4.2779  vmlinux-2.6.18  vmlinux-2.6.18  tcp_sendmsg
2800      3.8727  vmlinux-2.6.18  vmlinux-2.6.18  kfree
2627      3.6334  vmlinux-2.6.18  vmlinux-2.6.18  tcp_init_tso_segs
2170      3.0013  e1000e.ko       e1000e          e1000_clean_tx_irq  ――②
2155      2.9806  e1000e.ko       e1000e          e1000_xmit_frame    ――③
2140      2.9598  vmlinux-2.6.18  vmlinux-2.6.18  skb_clone
2133      2.9502  vmlinux-2.6.18  vmlinux-2.6.18  kmem_cache_free
1866      2.5809  vmlinux-2.6.18  vmlinux-2.6.18  tcp_ack
1830      2.5311  e1000e.ko       e1000e          e1000_irq_enable    ――④
1805      2.4965  vmlinux-2.6.18  vmlinux-2.6.18  cache_grow
1694      2.3430  e1000e.ko       e1000e          e1000_intr_msi      ――⑤
1642      2.2711  e1000e.ko       e1000e          e1000_clean_rx_irq  ――⑥
1554      2.1493  vmlinux-2.6.18  vmlinux-2.6.18  mwait_idle
1539      2.1286  vmlinux-2.6.18  vmlinux-2.6.18  tcp_v4_rcv
1473      2.0373  vmlinux-2.6.18  vmlinux-2.6.18  tcp_transmit_skb
1383      1.9128  vmlinux-2.6.18  vmlinux-2.6.18  __kfree_skb
1382      1.9115  vmlinux-2.6.18  vmlinux-2.6.18  eth_header
1166      1.6127  vmlinux-2.6.18  vmlinux-2.6.18  __alloc_skb
1139      1.5754  vmlinux-2.6.18  vmlinux-2.6.18  dev_queue_xmit
1072      1.4827  vmlinux-2.6.18  vmlinux-2.6.18  cache_alloc_refill
1047      1.4481  vmlinux-2.6.18  vmlinux-2.6.18  ip_queue_xmit
1004      1.3886  vmlinux-2.6.18  vmlinux-2.6.18  tcp_v4_send_check
949       1.3126  vmlinux-2.6.18  vmlinux-2.6.18  system_call
...
```

シンボルを解決すると e1000e も関数ごとにサンプリングの割合がわかります。①の 14.8% は②、③、④ ... ⑥ ... の合計です。続いて同じように通常のネットワークの場合のプロファイルを取ります。以下は通常のネットワークの場合です。

```
[sender]# ./nuttcp -T20s -l1500 192.168.1.100 &
...
[sender]# opreport -l -p /lib/modules/2.6.18/kernel/
...
samples  %        image name      app name        symbol name
8658     16.7839  vmlinux-2.6.18  vmlinux-2.6.18  copy_user_generic   ――⑦
2509      4.8638  vmlinux-2.6.18  vmlinux-2.6.18  tcp_sendmsg
2048      3.9701  e1000e.ko       e1000e          e1000_irq_enable
1941      3.7627  e1000e.ko       e1000e          e1000_xmit_frame
1803      3.4952  vmlinux-2.6.18  vmlinux-2.6.18  mwait_idle
1738      3.3692  e1000e.ko       e1000e          e1000_intr_msi
```

```
1566     3.0358  e1000e.ko       e1000e          e1000_clean_rx_irq
1424     2.7605  vmlinux-2.6.18  vmlinux-2.6.18  tcp_v4_rcv
1342     2.6015  vmlinux-2.6.18  vmlinux-2.6.18  ip_output
1295     2.5104  vmlinux-2.6.18  vmlinux-2.6.18  kfree
1098     2.1285  vmlinux-2.6.18  vmlinux-2.6.18  __tcp_push_pending_frames
1072     2.0781  vmlinux-2.6.18  vmlinux-2.6.18  kmem_cache_free
983      1.9056  vmlinux-2.6.18  vmlinux-2.6.18  system_call
974      1.8881  vmlinux-2.6.18  vmlinux-2.6.18  tcp_ack
800      1.5508  vmlinux-2.6.18  vmlinux-2.6.18  tcp_transmit_skb
772      1.4966  vmlinux-2.6.18  vmlinux-2.6.18  skb_split
768      1.4888  e1000e.ko       e1000e          e1000_clean_tx_irq
724      1.4035  vmlinux-2.6.18  vmlinux-2.6.18  skb_clone
721      1.3977  vmlinux-2.6.18  vmlinux-2.6.18  ip_queue_xmit
694      1.3454  nuttcp          nuttcp          Nwrite
691      1.3395  libc-2.5.so     libc-2.5.so     __write_nocancel
690      1.3376  vmlinux-2.6.18  vmlinux-2.6.18  dnotify_parent
678      1.3143  vmlinux-2.6.18  vmlinux-2.6.18  put_page
…
31       0.0601  vmlinux-2.6.18  vmlinux-2.6.18  copy_from_user   ――⑨
```

VLAN ネットワークと比較すると 1 番目の⑦が違います。VLAN ネットワークの場合は csum_partial_copy_generic() です。この関数について調査すると、コールシーケンスは図 5-16 になることがわかりました。

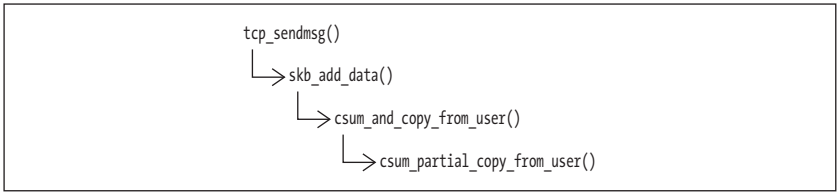

図 5-16 VLAN ネットワークにおける csum_partial_copy_generic() 関数コールシーケンス

```
[include/linux/skbuff.h]
  static inline int skb_add_data(struct sk_buff *skb,
                                 char __user *from, int copy)
  {
          const int off = skb->len;

          if (skb->ip_summed == CHECKSUM_NONE) {   /* 送信に使う NIC に
                  int err = 0;                        チェックサム計算機能がない */
```

```
                unsigned int csum = csum_and_copy_from_user(from,
                                                    skb_put(skb, copy),
                                                    copy, 0, &err);
                }
        } else if (!copy_from_user(skb_put(skb, copy), from, copy))    ――⑧
                return 0;
…
```

skb_add_data() はユーザから要求された送信データをカーネル（ソケットバッファ）に転送する関数です。CHECKSUM_NONE の場合、つまり NIC にチェックサム計算機能がない場合に csum_partial_copy_from_user() を実行します。csum_partial_copy_from_user() はユーザ空間からデータを転送するのと同時にチェックサムを計算します。この関数が呼ばれているということはソフトウェア（カーネル）でチェックサムを計算していることになります。

通常のネットワークはハードウェアのチェックサム計算機能を使っているとして、ソースコードの⑧がどのぐらいの割合なのか opreport コマンドを見ると⑨になります。ほとんど負荷がないようです。

ハードウェアのチェックサム計算機能を設定する仕組み

skb->ip_summed が CHECKSUM_NONE である理由をこれから見つけますが、e1000e ドライバがデバイスの情報を取得するのは初期化のときです。そのため e1000e ドライバをカーネルに組み込むときの処理から、ハードウェアのチェックサム計算機能を認識する仕組みを確認し

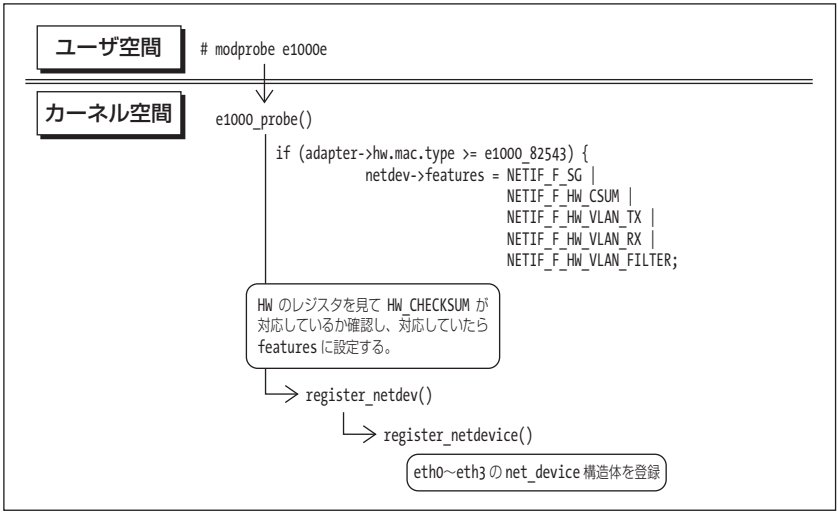

図 5-17　e1000e デバイス作成のシーケンス

ます。

features はネットワークデバイスがサポートしている機能を示します。機能とはチェックサム計算や VLAN などになります。features は sysfs で確認できます。

```
[sender]# cat /sys/class/net/eth3/features
0x1113a9
[sender]# cat /sys/class/net/eth3.510/features
0x0
```

features はビットになっており、値はカーネルの include/linux/netdevice.h に定義されています。

[include/linux/netdevice.h]

```
…
        unsigned long           features;
#define NETIF_F_SG              1       /* Scatter/gather IO. */
#define NETIF_F_IP_CSUM         2       /* Can checksum only TCP/UDP over IPv4. */
#define NETIF_F_NO_CSUM         4       /* Does not require checksum. F.e. loopack. */
#define NETIF_F_HW_CSUM         8       /* Can checksum all the packets. */
#define NETIF_F_HIGHDMA         32      /* Can DMA to high memory. */
#define NETIF_F_FRAGLIST        64      /* Scatter/gather IO. */
…
```

eth3（通常のネットワーク）は NETIF_F_HW_CSUM が有効になっています。しかし VLAN ネットワークの eth3.510 は 0x0 になっており、すべて無効です。そのため次は VLAN デバイス

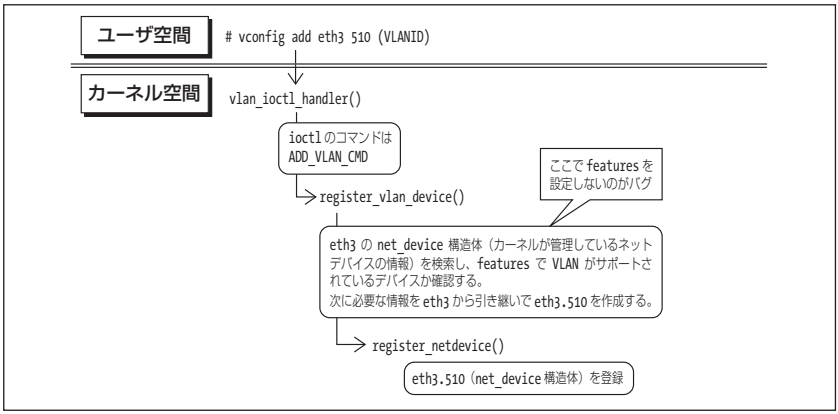

図 5-18 VLAN デバイス作成のシーケンス

の初期化を確認します。

VLAN デバイスが物理デバイス eth3 の features を引き継いでいないため、eth3.510 の features が 0x0 になっています。

ここまでは features についてです。skb->ip_summed ==CHECKSUM_NONE で csum_partial_copy_

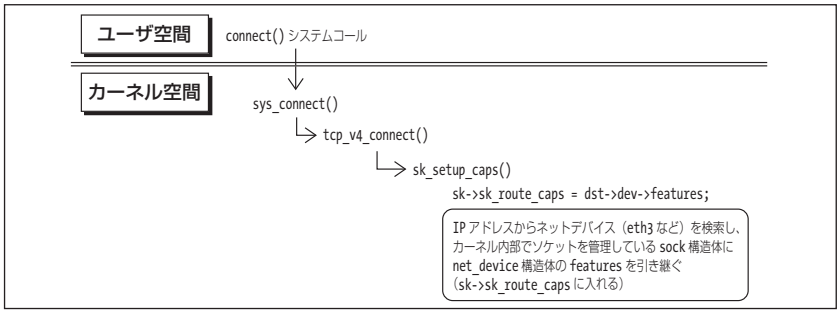

図 5-19　connect() システムコール処理

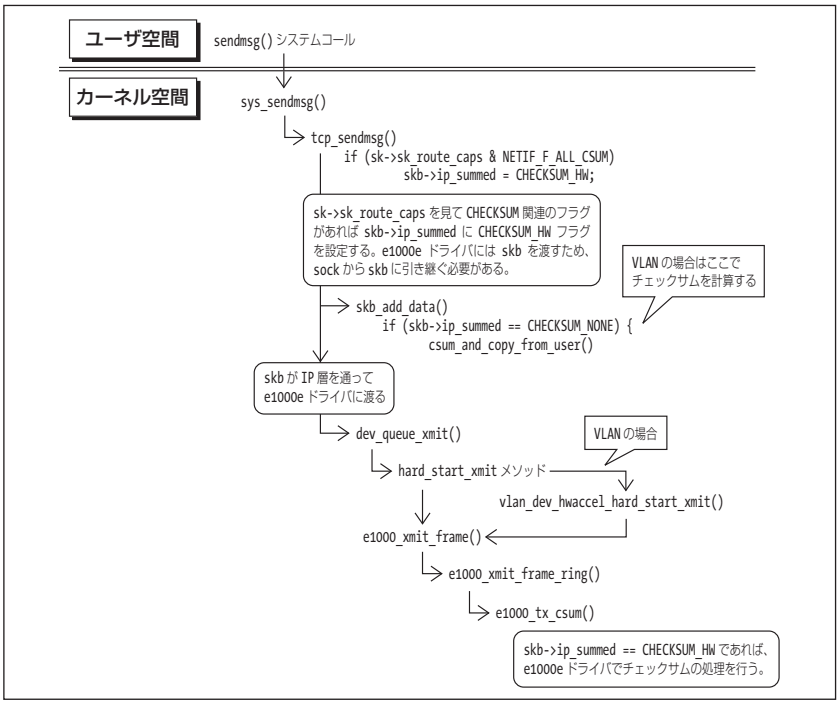

図 5-20　sendmsg() システムコール処理

generic() が実行される仕組みをシステムコールの処理で確認します。図 5-19、図 5-20 にシステムコールの処理をまとめます。

今までの調査をまとめると e1000e ドライバを組み込むとハードウェアを見て、net_device 構造体の features に NETIF_F_HW_CSUM フラグを設定します。connect() で net_device 構造体の features を sk->sk_route_caps に設定します。sendmsg() で sk->sk_route_caps を見て、skb->ip_summed に CHECKSUM_HW を設定します。

VLAN デバイスから TCP パケットを送信したときに、カーネルでチェックサムを計算してしまうのは、eth3 の features が eth3.510 のソケットバッファに引き継がれていないためです。

コミュニティによる修正

原因がわかりましたので、このことをネットワーク（netdev）のコミュニティにメールをすると、すぐに修正されました。

以下のパッチで修正されます。register_vlan_device() の中で features を設定するようになります。

```
commit 5fb13570543f4ae022996c9d7c0c099c8abf22dd
Author: Patrick McHardy <kaber@trash.net>
Date:   Tue May 20 14:54:50 2008 -0700

    [VLAN]: Propagate selected feature bits to VLAN devices

commit 289c79a4bd350e8a25065102563ad1a183d1b402
Author: Patrick McHardy <kaber@trash.net>
Date:   Fri May 23 00:22:04 2008 -0700

    vlan: Use bitmask of feature flags instead of seperate feature bits
```

e1000e ドライバで VLAN デバイスの features を設定するというアイデア（以下のパッチ）も出ましたが、これでは e1000e ドライバしか対応されないのと、ethtool コマンドの関係で上のパッチがメインラインにマージされました (Intel のサイトにある e1000 ドライバにはこのパッチが入っています)。

```
[PATCH 4/4] e1000e: Allow TSO to trickle down to VLAN device
http://www.spinics.net/lists/netdev/msg63716.html
```

修正後の測定

まずはパッチを適応したカーネルで features の確認をします。

```
# cat /sys/class/net/eth3.510/features
0x10009
```

NETIF_F_HW_CSUM フラグが有効になっています。

次に nuttcp で通信を行い oprofile の結果を見てみます。期待としては csum_partial_copy_generic() が現れないことです。

```
# opreport -l -p /lib/modules/2.6.18vlan/kernel/
...
samples  %        image name               app name                 symbol name
26955    23.3575  vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       copy_user_generic
6554      5.6793  vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       mwait_idle
5375      4.6576  e1000e.ko                e1000e                   e1000_irq_enable
4681      4.0563  e1000e.ko                e1000e                   e1000_intr_msi
4148      3.5944  e1000e.ko                e1000e                   e1000_clean_rx_irq
3770      3.2668  vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       tcp_v4_rcv
3494      3.0277  vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       tcp_sendmsg
3436      2.9774  e1000e.ko                e1000e                   e1000_xmit_frame
2523      2.1863  vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       tcp_ack
2478      2.1473  vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       kfree
2079      1.8015  vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       kmem_cache_free
1715      1.4861  vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       put_page
1554      1.3466  vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       __tcp_push_pending_frames
1494      1.2946  vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       IRQ0x42_interrupt
1462      1.2669  e1000e.ko                e1000e                   e1000_clean_tx_irq
1322      1.1456  vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       __kfree_skb
1206      1.0450  vmlinux-2.6.18vlan       vmlinux-2.6.18vlan       ip_rcv
...
```

期待通り csum_partial_copy_generic() がサンプリングにありませんでした。

次は nuttcp を同じオプションで再実行しました。パッチ適応前と適応後のカーネルで 10 回実行し、平均値を**表 5-3** にまとめました。Mbps は nuttcp のスループットの値、sec は nuttcp の時間、%TX は送信プロセス（sender）の CPU 使用率です。

表 5-3 2.6.18 カーネルとパッチを適応したカーネルの結果

	2.6.18			パッチ適応後		
	Mbps	sec	%TX	Mbps	sec	%TX
e1000e（eth3）	941.38	9.12	14.6	941.41	9.12	14.1
VLAN（eth3.510）	938.57	9.15	22.0	938.98	9.15	14.0

スループットはあまり変化がありませんが、CPU 使用率は 22% から 14% に減少し、通常のネットワークとほぼ同じになりました。

パッチを適応していない 2.6.18 カーネルでもスループットが出ていた理由としては、計測時に nuttcp だけ実行しており、CPU に余裕があったため、通常のネットワークとほぼ同じスループットを出せたのだと思います。

これでネットワークデバイスのチェックサム計算機能が、VLAN デバイスであっても使用されるようになりました。これで CPU 負荷が軽減されます。

まとめ

本 Hack は CPU の使用率を削減した例です。スループットだけを見るとわからない問題も、oprofile で解析すると今回のような問題が見えてきます。

参考文献

- Phil Dykstra's nuttcp quick start guide
  http://www.wcisd.hpc.mil/nuttcp/Nuttcp-HOWTO.html
- Linux Man Page TCP(7)
- ギガビット PCI ベースのネットワーク・コネクション（Linux*）用ネットワーク・アダプター・ドライバー
  http://downloadcenter.intel.com/Detail_Desc.aspx?strState=LIVE&ProductID=2776&DwnldID=16563&lang=jpn

—— Naohiro Ooiwa

6章 差がつくデバッグテクニック
Hack #43-66

この章では、デバッグするにあたってのさまざまなツールの紹介やちょっとしたノウハウなど幅広いものを集めました。紹介しているツールやテクニックも strace、objdump、Valgrind、kprobes、jprobes、KAHO、systemtap、proc ファイルシステム、oprofile、VMware vprobe、フォルト・インジェクション、Xen など多岐にわたります。その他、OOM Killer の動作と仕組み、GOT/PLT を経由した関数コールの仕組みと理解、initramfs、RT Watchdog を使ってリアルタイムプロセスのストールを検知する方法、手元の x86 マシンが 64 ビット対応かどうか調べる方法まで記しています。

HACK #43 strace を使って、不具合原因の手がかりを見つける

システムコールをトレースする strace コマンドを使って不具合が発生した際、その原因の手がかりを見つける方法を説明します。

strace

strace は、プロセスが呼び出すシステムコールをトレースし、その内容を表示します。そのため、原因がよくわからない不具合をデバッグする時、まず、strace を使って、システムコールがエラーになる箇所を探すと、不具合の手がかりが得られることがあります。とりわけ、ファイルに関するエラーや不正なパラメータは、この方法で比較的簡単に見つけられます。

この方法は、システムコールが失敗する不具合の原因を探るには有効ですが、ユーザが記述したプログラムや共有ライブラリの中で発生しているエラーは見つけられません。

strace の使用例

アクセスしようとしたファイルが見つからない、あるいは、ファイルに対するアクセス

の権限がない場合などは、システムコールから、それらのエラー内容がほぼそのまま返されます。実際に、その様子を次のプログラム st1.c を使って確認しましょう。

[st1.c]

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp;
    fp = fopen("/etc/shadow", "r");
    if (fp == NULL) {
        printf("Error!\n");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

このプログラムを実行すると、次のように表示されます。

```
$ ./st1
Error!
```

これは、一般ユーザには読み取り権限のない /etc/shadow ファイルをオープンしようとしたためです。しかし、エラーメッセージから、そのことを理解するのは、まず不可能です。実際のプログラムでも、エラーメッセージの内容が不明確だったり、いたるところで同じエラーメッセージが表示されたり、あるいは、何も表示されない場合があります。これはそのようなプログラムの一例です。このような場合、エラーがソースのどこで出力されているのかを特定するのが困難なため、GDB を使っても、ブレークポイントを設定できません。そこで、次のように strace を実行してみます。

```
$ strace st1
execve("./st1", ["st1"], [/* 44 vars */]) = 0
brk(0)                                  = 0x486b000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x2aaaaaaab000
uname({sys="Linux", node="cglsv1.cgl", ...}) = 0
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/usr/local/X11R7/lib/tls/x86_64/libc.so.6", O_RDONLY) = -1 ENOENT (No such file or directory)
...
```

```
(省略)
…
open("/lib64/libc.so.6", O_RDONLY)      = 3 ―――――――――――――――――――――――――――――――――①
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\240\331"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1687632, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x2aaaaaad7000
mmap(0x359de00000, 3461272, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x359de00000
mprotect(0x359df44000, 2097152, PROT_NONE) = 0
mmap(0x359e144000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x144000) =
0x359e144000
mmap(0x359e149000, 16536, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) =
0x359e149000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x2aaaaaad8000
arch_prctl(ARCH_SET_FS, 0x2aaaaaad8210) = 0
mprotect(0x359e144000, 16384, PROT_READ) = 0
mprotect(0x359dc19000, 4096, PROT_READ) = 0
munmap(0x2aaaaaaac000, 173557)          = 0
brk(0)                                  = 0x486b000
brk(0x488c000)                          = 0x488c000
open("/etc/shadow", O_RDONLY)           = -1 EACCES (Permission denied) ―――――――――――――――②
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x2aaaaaaac000
write(1, "Error!\n", 7)                 = 7 ―――――――――――――――――――――――――――――――――③
exit_group(1)                           = ?
Process 18399 detached
```

エラー終了するような場合、strace の出力を最後から見ていくのが解決への近道です。上記の例でも、最後の方の③が画面にエラーメッセージを出力するシステムコールです。さらに、その前を見ると、②で open() システムコールが失敗しているのがわかります。しかも、/etc/shadow をオープンしようとして Permission denied となっていることまですぐにわかります。

たくさんのメッセージが表示されますが、最初の方に表示されるのは、プロセスが起動するための処理です。エラーが多く表示されていますが、これは、プロセスが使用する共有ライブラリをいろいろなパスからロードしようとして、失敗しているためです。①からの 10 数行で、st1 が使用するライブラリをプロセスにリンクすることに成功していることがわかります。この辺りまではランタイムローダが行っている処理なので、無視しても構いません。

GDBでの詳細な調査

先ほどの例では、straceの出力だけでも十分状況が理解できましたが、プログラムによっては、エラーを引き起こす真の原因は、別の場所にあることもあります。そのような場合、GDBを使って、さらなる調査を進めるの定石ですが、そのためには、ブレークポイントに設定するアドレスが必要です。straceでは、-iを使うことでシステムコールを呼び出したアドレスが表示されるので、その値をブレークポイントして使用することができます。

```
$ strace -i st1
…
[      359debf310] open("/etc/shadow", O_RDONLY) = -1 EACCES (Permission denied)
…
```

各行先頭[]の中の数値が、システムコールを呼び出しているコードのアドレスです。GDBで、そのアドレスを指定して、バックトレースを表示させてみます。

```
$ gdb st1
…
(gdb) start
Breakpoint 1 at 0x4004c0: file st1.c, line 7.
Starting program: /home/kyamato/DebugHacks/kyamato/chapter5/strace.work/st1g
main () at st1.c:7
7           fp = fopen("/etc/shadow", "r");
(gdb) b *0x359debf310  ―――――――――――――――――――――――――――straceで表示されたアドレス
Breakpoint 2 at 0x359debf310
(gdb) c
Continuing.

Breakpoint 2, 0x000000359debf310 in __open_nocancel () from /lib64/libc.so.6
(gdb) bt
#0  0x000000359debf310 in __open_nocancel () from /lib64/libc.so.6
#1  0x000000359de68f23 in __GI__IO_file_open () from /lib64/libc.so.6
#2  0x000000359de6906c in _IO_new_file_fopen () from /lib64/libc.so.6
#3  0x000000359de5eba4 in __fopen_internal () from /lib64/libc.so.6
#4  0x00000000004004cf in main () at st1.c:7
gdb st1
```

プロセスへのアタッチ

ここまでは、straceからプロセスを起動させて、その動作を確認しました。次に、デーモンのようにすでに実行されているプロセスの動作をstraceで確認する方法を、以下の

プログラムを使って説明します。

[st2.c]

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    while(1) {
        FILE *fp;
        fp = fopen("/etc/shadow", "r");
        if (fp == NULL)
            printf("Error!\n");
        else
            close(fp);
        sleep(3);
    }
    return EXIT_SUCCESS;
}
```

まず、上記のプログラムst2を実行させます。このプログラムは、終了することなく、動作し続けます。このプログラムのシステムコールをトレースするには、次のように-p <PID>オプションを使います。その結果を下記に示します。st2のシステムコールがトレースされ、④や⑤で発生しているエラーが表示されています。なお、終了するときにはCtrl-Cを入力します。

```
$ strace -p `pidof st2`
Process 23030 attached - interrupt to quit
restart_syscall(<... resuming interrupted call ...>) = 0
open("/etc/shadow", O_RDONLY)           = -1 EACCES (Permission denied) ——④
write(1, "Error!\n", 7)                 = 7
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
nanosleep({3, 0}, {3, 0})               = 0
open("/etc/shadow", O_RDONLY)           = -1 EACCES (Permission denied) ——⑤
write(1, "Error!\n", 7)                 = 7
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
nanosleep({3, 0},  <unfinished ...>
```

その他の便利な使い方

表示内容をファイルへ出力するには、次のように -o オプションを用います。

```
$ strace -o output.log command
```

strace の出力は、標準エラー出力です。そのため、表示内容を標準出力し、grep や less に渡すには、次のようにします（bash の場合）。

```
$ strace command 2>&1 | grep mmap
$ strace command 2>&1 | less
```

プロセスが fork() した際、fork() 後のプロセスもトレースする場合、-f オプションを使用します。

```
$ strace -f command
```

システムコールが発行された時刻を -t、または、-tt オプションで表示させることができます。違いは、-t は秒単位で、-tt はマイクロ秒単位で表示することです。

```
$ strace -t command
$ strace -tt command
```

まとめ

システムコールの呼び出しをトレースする strace コマンドを使って、不具合が発生した際、その原因の手がかりを見つける方法を説明しました。

—— Kazuhiro Yamato

HACK #44

objdump の便利なオプション

objdump でデバッグ情報付きのバイナリを扱うときに便利なオプションを紹介します。

objdump には便利な -S、-l オプションがあります。デバッグ情報のあるバイナリであれば、以下のようにソースコード、ファイル名とその行数がアセンブリ言語と対応して表示されます。以下の例では「**アセンブリ言語の勉強法**」**[HACK #13]** で使用したテストプログラム（TP）を使います。

```
$ gcc -Wall -O0 -g assemble.c -o assemble
$ objdump -Sl --no-show-raw-insn assemble
…
```

```
080483cc <main>:
...
/home/user/assemble.c:18
        unsigned int i = 0;
 80483dd:       movl   $0x0,-0x10(%ebp)
...
/home/user/assemble.c:24

        i = 0xabcd;
 804840a:       movl   $0xabcd,-0x10(%ebp)
/home/user/assemble.c:26

        if (i != 0x1234)
 8048411:       cmpl   $0x1234,-0x10(%ebp)
 8048418:       je     8048427 <main+0x5b>
/home/user/assemble.c:27
                i = 0;
 804841a:       movl   $0x0,-0x10(%ebp)
/home/user/assemble.c:29

        while (i == 0)
 8048421:       jmp    8048427 <main+0x5b>
/home/user/assemble.c:30
                i++;
 8048423:       addl   $0x1,-0x10(%ebp)
/home/user/assemble.c:29
        i = 0xabcd;

        if (i != 0x1234)
                i = 0;

        while (i == 0)
 8048427:       cmpl   $0x0,-0x10(%ebp)
 804842b:       je     8048423 <main+0x57>
/home/user/assemble.c:32
                i++;

        func();
 804842d:       call   804839e <func>
/home/user/assemble.c:33
        i = func_pointer();
```

```
 8048432:       mov    -0x8(%ebp),%eax
 8048435:       call   *%eax
 8048437:       mov    %eax,-0x10(%ebp)
/home/user/assemble.c:35

        for (i=0; i<MAX_WORD-1 ; i++)
 804843a:       movl   $0x0,-0x10(%ebp)
 8048441:       jmp    8048452 <main+0x86>
/home/user/assemble.c:36
                word = words[i];
 8048443:       mov    -0x10(%ebp),%eax
 8048446:       movzbl -0x20(%ebp,%eax,1),%eax
 804844b:       mov    %al,-0x9(%ebp)
/home/user/assemble.c:35
                i++;

        func();
        i = func_pointer();

        for (i=0; i<MAX_WORD-1 ; i++)
 804844e:       addl   $0x1,-0x10(%ebp)
 8048452:       cmpl   $0xe,-0x10(%ebp)
 8048456:       jbe    8048443 <main+0x77>
/home/user/assemble.c:38
                word = words[i];

        return 0;
 8048458:       mov    $0x0,%eax
/home/user/assemble.c:39
}
 804845d:       add    $0x24,%esp
 8048460:       pop    %ecx
 8048461:       pop    %ebp
 8048462:       lea    -0x4(%ecx),%esp
 8048465:       ret
…
```

これはデバッグ情報を使って表示しているため、GCC でコンパイルするときは -g を付けなければなりません。またアセンブリ言語とソースコードではどうしてもずれが生じる

ため上のように while 文が 2 箇所に現れたりします。完全に正確なものでもありませんが、便利です。

まとめ

objdump の便利なオプションを紹介しました。ただしこのオプションですべてがわかるわけではありません。最適化されているバイナリの場合は特にアテにできません。目安として使うのがよいでしょう。objdump の -l オプションと同じような情報を出力する addr2line コマンドもあります。

—— Naohiro Ooiwa

HACK #45 Valgrind の使い方（基本編）

プログラムの動的解析ツールである Valgrind の基本的な使い方を説明します。

Valgrind とは

Valgrind は、メモリの不正使用の検出、キャッシュやヒープのプロファイル、POSIX スレッドの競合検出などを行うことができます。Valgrind の特徴のひとつは、検査対象のプログラムのビルド時に特別なオプションの指定や、ライブラリのリンクが不要なことです。本 Hack では、Valgrind のもっとも典型的な用途であるメモリの不正使用の検出方法を説明します。基本的な使い方は、次のとおりです。ここで、program はチェックするプログラムのファイル名です。

```
$ valgrind --tool=memcheck --leak-check=yes program
```

また、メモリの不正使用の検出（memcheck）はデフォルト動作なので、単に以下のように入力することもできます。

```
$ valgrind --leak-check=yes program
```

メモリリークの検出

ここでは、ソースが以下の test1.c であるプログラム test1 を使って説明します。test1 は、malloc() の後に、free() を実行しない典型的なメモリリーク動作をします。なお、ビルド時には、Valgrind にソース行数等を表示されるために -g オプションを指定します。

[test1.c]

```
int main(void)
{
```

```
    char *p = malloc(10);    /* メモリ確保 */
    return EXIT_SUCCESS;     /* メモリを解放せず終了 */
}
```

実行結果は、以下のようになります。

```
$ valgrind test1
...
==3125== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 1)
==3125== malloc/free: in use at exit: 10 bytes in 1 blocks.
==3125== malloc/free: 1 allocs, 0 frees, 10 bytes allocated. ————————(A)
==3125== For counts of detected errors, rerun with: -v
==3125== searching for pointers to 1 not-freed blocks.
==3125== checked 65,416 bytes.
==3125==
==3125== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1 ——(B)
==3125==    at 0x4A05809: malloc (vg_replace_malloc.c:149)
==3125==    by 0x400489: main (test1.c:6)
==3125==
==3125== LEAK SUMMARY: ————————————————————————————————————(C)
==3125==    definitely lost: 10 bytes in 1 blocks.
==3125==      possibly lost: 0 bytes in 0 blocks.
==3125==    still reachable: 0 bytes in 0 blocks.
==3125==         suppressed: 0 bytes in 0 blocks.
==3125== Reachable blocks (those to which a pointer was found) are not shown.
==3125== To see them, rerun with: --show-reachable=yes
```

上記の出力の ==3125== の数字部分 3125 は、実行された Valgrind のプロセス ID です。そのため、実行ごとに異なる値が表示されます。すべての行に表示されるため、やや目立ちますが、それほど重要な情報ではありません。

Valgrind の出力で、まず注目すべきは、malloc/free の回数 (A) です。malloc の実行 1 回に対して、free は 0 回と表示されています。LEAK SUMMARY(C) でも、definitely lost として表示されています。より詳細な情報は、(B) に表示されており、これによるとアドレス 0x400489（test1.c の 6 行目）で実行された malloc() により確保されたメモリがリークしているとなっています。ソースと照らし合わせると、正しくメモリリークが検出されていることがわかります。

不正なメモリ位置へのアクセス検出

次に、以下の test2.c のように確保した領域外へアクセスするバグを検出する方法を示します。

[test2.c]

```
int main(void)
{
    char *p = malloc(10);    /* 10B の確保 */
    p[10] = 1;               /* 確保した領域外 (11B 目 ) への書き込み */
    free(p);
    return EXIT_SUCCESS;
}
```

このプログラムを Valgrind でチェックすると、次の出力が得られ、確保した領域の外のアドレスに書き込みを行っていることがわかります。

```
…
==3438== Invalid write of size 1
==3438==    at 0x4004D6: main (test2.c:7)
==3438==  Address 0x4C3603A is 0 bytes after a block of size 10 alloc'd
==3438==    at 0x4A05809: malloc (vg_replace_malloc.c:149)
==3438==    by 0x4004C9: main (test2.c:6)
…
```

初期化されていない領域の読み出し

初期化されていない領域の読み出しも検査できます。ここでは、test3.c を使って説明します。

[test3.c]

```
int main(void)
{
    int *x = malloc(sizeof(int));  /* int サイズのメモリ確保 */
    int a = *x + 1;                /* 確保したメモリを初期化せず使用 */
    free(x);
    return a;
}
```

上記のコードをビルドし、Valgrind で検査すると、以下のような出力結果が得られます。

```
...
==3941== Syscall param exit_group(exit_code) contains uninitialised byte(s)
==3941==    at 0x359DE948CF: _Exit (in /lib64/libc-2.5.so)
==3941==    by 0x359DE32D04: exit (in /lib64/libc-2.5.so)
==3941==    by 0x359DE1D8AA: (below main) (in /lib64/libc-2.5.so)
...
```

解放後の領域に対するアクセス

解放後の領域へのアクセスも検出できます。例として test4.c を使用します。

[test4.c]

```
int main(void)
{
    int *x = malloc(sizeof(int));   /* メモリ確保 */
    free(x);                        /* メモリ解放 */
    int a = *x + 1;                 /* 解放されたメモリ領域を参照 */
    return a;
}
```

以下のようにアクセス箇所の指摘が行われます。

```
...
==4134== Invalid read of size 1
==4134==    at 0x4004DB: main (test4.c:8)
==4134==  Address 0x4C36030 is 0 bytes inside a block of size 4 free'd
==4134==    at 0x4A0541E: free (vg_replace_malloc.c:233)
==4134==    by 0x4004D6: main (test4.c:7)
...
```

メモリの二重解放

test5.c のようなメモリの二重解放の検出もできます。

[test5.c]

```
int main(void)
{
    char *x = malloc(sizeof(int));  /* メモリ確保 */
    free(x);                        /* メモリ解放 */
    free(x);                        /* 同じアドレスの二重解放 */
    return EXIT_SUCCESS;
}
```

上記のコードをビルドし、Valgrind でチェックすると、これも問題が指摘されます。

```
…
==4236== Invalid free() / delete / delete[]
==4236==    at 0x4A0541E: free (vg_replace_malloc.c:233)
==4236==    by 0x4004DF: main (test5.c:8)
==4236==  Address 0x4C36030 is 0 bytes inside a block of size 4 free'd
==4236==    at 0x4A0541E: free (vg_replace_malloc.c:233)
==4236==    by 0x4004D6: main (test5.c:7)
…
```

不正なスタック領域の操作

以下の test6.c は、スタックポインタより低位アドレスのメモリにデータを書き込みます。

[test6.c]

```
int main(void)
{
    int a;          /* スタック上に変数 ( メモリ ) を確保 */
    int* p = &a;    /* a のアドレスを指すポインタを作成 */
    p -= 0x20;      /* 誤ったポインタ操作 */
    *p = 1;         /* 誤ったメモリ位置への書き込み */
    return EXIT_SUCCESS;
}
```

GCC を使ってビルドしたプログラムを Valgrind で検査すると以下の出力が表示されます。ただし、具体的な動作はコンパイラ依存なので、異なる環境では出力内容に違いがあるかもしれません。

```
…
928==
==5928== Invalid write of size 4
==5928==    at 0x40043D: main (test6.c:9)
==5928==  Address 0x7FF000024 is just below the stack ptr.  To suppress, use:
--workaround-gcc296-bugs=yes
…
```

test6 では、次のことが起こっています。変数 a は、スタックに生成されます。main 関数では a と p しか変数を使っていないのでスタックの下限（スタックポインタの指す値）は、

aのアドレスそのものか、それより数バイトから数十バイト小さいものと期待されます。そのため、p -= 0x20で得られるアドレス（aのアドレスより0x80バイト = 0x20*sizeof(int)バイト小さいアドレス）は、多くの場合、スタックポインタの値より小さくなります。そのアドレスに1を書き込んだため、Valgrindは、そのことを検出し、メッセージを出力しました。

検出できないエラー

メモリの不正使用の検出に効果を発揮するValgrindですが、万能というわけでもありません。例えば、次のようなスタックに生成されたメモリ領域への不正なアクセスは検出できません。

```
test_a.c
int main(void)
{
    char p[10];
    p[100] = 1;
    return EXIT_SUCCESS;
}
```

上記のコードをビルドし、検査しても、次のようにエラー等は何も表示されません。

```
…
==6284== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 1)
==6284== malloc/free: in use at exit: 0 bytes in 0 blocks.
==6284== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==6284== For counts of detected errors, rerun with: -v
==6284== All heap blocks were freed -- no leaks are possible.
```

まとめ

Valgrindを用いて、メモリリーク、不正なメモリ位置へのアクセス、初期化されていない領域の読み出し、解放後の領域に対するアクセス、メモリの二重解放、不正なスタック領域の操作を検出する方法を説明しました。

—— Kazuhiro Yamato

HACK #46 Valgrind の使い方（実践編）

Valgrind を使って、実際に検出した発見が困難なメモリリークの事例について説明します。

メモリリークの検出

メモリリークのやっかいな点のひとつは、それが発生しても、多くの場合、その瞬間には、何も問題が起こっていないように振る舞うことです。それでも、メモリリークの程度が大きい場合は、top コマンド等でプロセスの使用メモリ量を長時間監視することで、その兆候を検出することができます。例えば、いま foo というプログラムがメモリリークをしている場合、プロセスの仮想メモリの使用量（VIRT と RES）は、時間と共に以下のように増加します。VIRT と RES の違いは、VIRT には、スワップされている領域も含まれますが、RES にはそれが含まれない点です。

```
[開始直後]
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2512 root      25   0 12268 6880  284 R  100  1.8   0:18.66 foo

[10 分経過後]
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2512 root      25   0 31084  23m  284 R  100  6.2   1:08.65 foo

[40 分経過後]
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2512 root      25   0  121m 114m  284 R  100 30.9   5:38.63 foo
```

VIRT や RES が少しだけ増加するケースの中には、メモリリークでないこともあります。というのは、ライブラリの中にはメモリを確保するとき、効率のために多めに領域を確保することや、確保した領域が不要になっても、キャッシュとしてしばらく、その領域を保持することがあるからです。例えば、ユーザプログラムが数バイトの領域を malloc() で確保する場合、要求を受けた glibc は、内部的には、それ以上のメモリをカーネルから確保します。また、free() を実行しても、すぐさま、その領域をカーネルに返却するわけでありません。しかし、このように余分に確保されているメモリは、ライブラリやそのバージョン等にもよりますが、せいぜい数メガバイトです。

検出の困難なメモリリーク

一方、メモリリーク量が小さい場合、前説の方法での検出は困難です。しかし、数ヶ月や数年に渡って長期間稼働するようなプログラムであれば、徐々にメモリ資源が消費され、ある日、OOM Killer の発動（「OOM Killer の動作と仕組み」[HACK #56] 参照）など思わ

ぬ事態を招きます。

そのような長期間動作するプログラムでは、一度 Valgrind を使ってチェックすることをお奨めします。以下は、実際にあったメモリリークを発見するきっかけとなったプログラムの、そのメモリリークに関連する部分のみを抜粋したコードです。dlopen() は、共有ライブラリを実行時にロードする関数で、dlclose() は、逆に dlopen() でロードされた共有ライブラリをクローズする関数です。以下のコードは抜粋されているため、オープンの後、すぐにクローズを行っていますが、実際には、オープンした後、その共有ライブラリに含まれる関数を使用し、その関数が不要になった時、クローズします。

```
#include <stdio.h>
#include <dlfcn.h>

int main(){
    void *p = NULL;

    while(1){
        p = dlopen("./lib1.so",RTLD_LAZY);
        if (NULL == p){
            printf("Error: dlopen()\n");
            return 1;
        }
        dlclose(p);
        sleep(100);
    }
    return 0;
}
```

コードを見る限り、一見、問題箇所はないように思われます。実際、このコードに問題はありません。ところが、Valgrind を使うと、以下のように表示されました。

```
$ valgrind --leak-check=full prog
(実行後、Ctrl-C を入力)
...
==2370== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 5 from 1)
==2370== malloc/free: in use at exit: 8 bytes in 1 blocks.
==2370== malloc/free: 6 allocs, 5 frees, 1,454 bytes allocated.
==2370== For counts of detected errors, rerun with: -v
==2370== searching for pointers to 1 not-freed blocks.
==2370== checked 68,368 bytes.
```

```
==2370==
==2370== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2370==    at 0x4A05809: malloc (vg_replace_malloc.c:149)
==2370==    by 0x37D7E067EF: _dl_map_object_from_fd (dl-load.c:1473)
==2370==    by 0x37D7E07CAB: _dl_map_object (dl-load.c:2232)
==2370==    by 0x37D7E1088E: dl_open_worker (dl-open.c:252)
==2370==    by 0x37D7E0CC35: _dl_catch_error (dl-error.c:178)
==2370==    by 0x37D7E1036B: _dl_open (dl-open.c:551)
==2370==    by 0x37D8A00F79: dlopen_doit (dlopen.c:66)
==2370==    by 0x37D7E0CC35: _dl_catch_error (dl-error.c:178)
==2370==    by 0x37D8A014EC: _dlerror_run (dlerror.c:164)
==2370==    by 0x37D8A00EF0: dlopen@@GLIBC_2.2.5 (dlopen.c:87)
==2370==    by 0x4005C6: main (main.c:8) ―――――――――――――――――――――――――――――――――――①
==2370==
==2370== LEAK SUMMARY:
==2370==    definitely lost: 8 bytes in 1 blocks.
==2370==      possibly lost: 0 bytes in 0 blocks.
==2370==    still reachable: 0 bytes in 0 blocks.
==2370==         suppressed: 0 bytes in 0 blocks.
==2370== Reachable blocks (those to which a pointer was found) are not shown.
==2370== To see them, rerun with: --show-reachable=yes
```

なお、このプログラムは、デーモンのように終了しないプログラムなので、実行して少し経過したところで、Ctrl - C を入力して強制終了させました。出力内容の詳細な見方は、**「Valgrind の使い方（基本編）」[HACK #45]** を参照してください。ここでは、①で main.c の 8 行目から dlopen() が呼ばれ、さらにそこから呼ばれた malloc() で確保されたメモリが、リークしていることが報告されています。実際にソースコードの dl-load.c:1473（実際には改行されているので 1474 行目）を見てみると、次のように malloc() が使用されています。また、ここで確保されたメモリを解放するコードはありませんでした。

```
1471      /* Create an appropriate searchlist.  It contains only this map.
1472         This is the definition of DT_SYMBOLIC in SysVr4.  */
1473      l->l_symbolic_searchlist.r_list =
1474        (struct link_map **) malloc (sizeof (struct link_map *));
```

この問題の修正は参考文献にあります。

dlopen() は、glibc が提供する関数ですので、この例ではユーザプログラムではなく、OS のライブラリのバグであることがわかりました。このようにライブラリに問題がある場合、当然ながら、ユーザのコードをいくらレビューしても問題は発見されません。また、dlopen() のコールごとのリーク量は数バイトなので、本 Hack の前半で紹介した top を使う方法でメモリ量を監視しても、数時間程度では顕著な増加を見ることはできません。そのような状況でも、Valgrind を使うことで問題を発見することができます。

参考文献

- glibc/efl/dl-load.c の CVS
  http://sourceware.org/cgi-bin/cvsweb.cgi/libc/elf/dl-load.c.diff?r1=1.249.2.31&r2=1.290&cvsroot=glibc&f=h

まとめ

Valgrind を使って、OS が提供するライブラリに不具合があり、かつその発見が困難なメモリリークを検出する方法を説明しました。

—— Kazuhiro Yamato

HACK #47 kprobes を使って、カーネル内部の情報を取得する

カーネルのデバッグ機能の 1 つである kprobes の使い、動的にプローブを挿入し、カーネル内部の情報を取得する方法を説明します。

kprobes

printk を用いて変数などを表示させることは、有効なカーネルデバッグ方法のひとつです。しかし、この方法は、カーネルの再構築、再起動を必要とするので、デバッグの効率がよくありません。kprobes をカーネルモジュールから使用することで、printk 等を含むデバッグのためのプローブを任意のアドレスに、カーネルの再構築、再起動なく挿入することができます。

簡単な例

以下に do_execve() 関数の先頭をプローブするサンプルを示します。プローブを登録するには、まず、kprobes の動作に必要な kprobe 構造体型の変数を確保します。次に、kprobe 構造体のメンバ addr にプローブを挿入するアドレスを設定します。このアドレスの取得方法は複数ありますが、ひとつの方法は、kallsyms_lookup_name() を使う方法です。ただし、この関数は、カーネルバージョン 2.6.19 から EXPORT されなくなり、モジュールから使

用することができなくなりました。その代わり、カーネルバージョンが 2.6.19 以上であれば、struct kprobes の symbol_name メンバに、プローブ対象関数のシンボル名を指定することができます（下記ソースコード参照）。symbol_name を設定した場合、addr メンバには何も設定しなくても構いません。

RHEL5など、ディストリビューションによっては、2.6.18以前のカーネルでも、symbol_name が使用できる場合があります。

また、次のように /proc/kallsyms を使って直接アドレスを求めることもできます。

```
# cat /proc/kallsyms | grep "\ do_execve$"
ffffffff8003e1b4 T do_execve
```

続いて、pre_handler メンバにプローブ関数を設定し、register_kprobe 関数を呼び出します。これで、指定したアドレスが実行される時（厳密には実行される直前に）、プローブが呼び出されます。また、プローブの解除は、unregister_kprobe() 関数を用いて行います。

```
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>

struct kprobe kp; /* krpobe 構造体型の変数の確保 */

int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
    printk(KERN_INFO "pt_regs: %p, pid: %d, jiffies: %ld\n",
                        regs, current->tgid, jiffies);
    return 0;
}

static __init int init_kprobe_sample(void)
{
    /* do_execve() のアドレスを設定 */
    kp.addr = (kprobe_opcode_t *)0xffffffff8003e1b4;

    /* addr メンバに直接、アドレスを設定するかわりに次のように
       シンボル名を使うことも可能 */
    /* kernel version が 2.6.18 以下 */
    /* kp.addr = (kprobe_opcode_t *)kallsyms_lookup_name("do_execve"); */
    /* kernel version が 2.6.19 以上 */
```

```
    /* kp.symbol_name = "do_execve"; */

    /* 設定されたアドレスの命令が実行される直前のプローブ設定 */
    kp.pre_handler = handler_pre;

    /* プローブの登録 */
    register_kprobe(&kp);

    return 0;
}

module_init(init_kprobe_sample);

static __exit void cleanup_kprobe_sample(void)
{
    /* プローブの登録解除 */
    unregister_kprobe(&kp);
}
module_exit(cleanup_kprobe_sample);

MODULE_LICENSE("GPL");
```

上記のサンプルをビルドしてinsmodすると、do_execve()が実行される度に以下のようなカーネルメッセージが出力されます。この関数は、プロセス生成時に使用されるので、ls等のコマンドを実行する度に1つ表示されるはずです。

```
pt_regs: ffffffff80414f58, pid: 6899, jiffies: 4405656851
pt_regs: ffff810009189f58, pid: 6902, jiffies: 4405656857
pt_regs: ffffffff80414f58, pid: 6903, jiffies: 4405656864
…
```

引数の調査

上記のサンプルの方法を使うと、指定した関数が実行された時の、グローバルな変数currentやjiffiesは簡単に表示することができます。しかし、実際のデバッグでは、その関数で使われている変数の値を調べたい場合があります。kprobesのプローブ内である関数のローカル変数の値を表示させるためにはいくらかの工夫が必要です。なぜなら、printkの引数に直接変数名を指定することができないからです。その代わり、プローブ関数には、そのアドレスの命令実行時のレジスタ情報が格納された下記のような構造体pt_regsが与えられます。もちろん、この構造体のメンバは、アーキテクチャによって異なりますが、これを用いることで、変数などより詳しい情報を表示させることができます。

```
struct pt_regs {
    unsigned long r15;
    unsigned long r14;
    unsigned long r13;
    unsigned long r12;
    unsigned long rbp;
    unsigned long rbx;
    unsigned long r11;
    unsigned long r10;
    unsigned long r9;
    unsigned long r8;
    unsigned long rax;
    unsigned long rcx;
    unsigned long rdx;
    unsigned long rsi;
    unsigned long rdi;
    unsigned long orig_rax;
    unsigned long rip;
    unsigned long cs;
    unsigned long eflags;
    unsigned long rsp;
    unsigned long ss;
 };
```

まず、引数の情報を表示させてみましょう。do_execve() は、次のような引数を取ります。

```
int do_execve(char * filename, char __user *__user *argv,
              char __user *__user *envp, struct pt_regs * regs)
```

例えば、filename と argv を表示させるプローブのコードは次のようになります。

```
int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
    int cnt = 0;
    char __user *__user *argv;

    printk(KERN_INFO "filename: %s\n", (char*)regs->rdi);
    for (argv = (char __user *__user *)regs->rsi; *argv != NULL; argv++, cnt++)
        printk(KERN_INFO "argv[%d]: %s\n", cnt, *argv);
        return 0;
}
```

x86_64 アーキテクチャでは、関数の引数は、左から順に rdi、rsi、rdx、rcx、r8、r9 に格納されるため、rdi と rsi を参照すれば、第 1 引数と第 2 引数の値を得ることができます（**「関数コール時の引数の渡され方（x86_64 編）」[HACK #10]**、**「関数コール時の引数の渡され方（i386 編）」[HACK #11]** 参照）。

引数の値を調べるだけであれば、**「jprobes を使って、カーネル内部の情報を取得する」[HACK #48]** で説明する jprobes を使うとより簡単にできます。

スタックトレースの表示

kprobes を用いたもうひとつの有効なデバッグ方法は、スタックトレースの表示です。

```
int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
    dump_stack();
    return 0;
}
```

上記のプローブを挿入すると、下記のようなカーネルメッセージが出力されます。

```
Call Trace:
 <#DB>  [<ffffffff8003e1b4>] do_execve+0x0/0x243
 [<ffffffff883e3009>] :kpro3:handler_pre+0x9/0x10
 [<ffffffff8006512c>] kprobe_handler+0x198/0x1c8
 [<ffffffff80065197>] kprobe_exceptions_notify+0x3b/0x75
 [<ffffffff80065def>] notifier_call_chain+0x20/0x32
 [<ffffffff800649b4>] do_int3+0x42/0x83
 [<ffffffff8006413b>] int3+0x93/0xa4
 [<ffffffff8003e1b5>] do_execve+0x1/0x243
 <<EOE>>  [<ffffffff80052a64>] sys_execve+0x36/0x4c
 [<ffffffff8005c4d3>] stub_execve+0x67/0xb0
```

参考文献

カーネルソースツリー /Docuementation/kprobes.txt

まとめ

kprobes を用いて、カーネルのグローバル変数と引数およびスタックトレースを表示させる方法を示しました。

—— Kazuhiro Yamato

HACK #48 jprobes を使って、カーネル内部の情報を取得する

カーネルのデバッグ機能のひとつである jprobes を使い、カーネル関数の先頭にプローブを挿入し、カーネル内部の情報を取得する方法を説明します。

jprobes

「kprobes を使って、カーネル内部の情報を取得する」**[HACK #47]** で、kprobes の使い方について説明しました。kprobes は、カーネル内のほぼすべての箇所にプローブを挿入できます。一方、jprobes は、関数先頭のプローブに特化されています。そのため、関数に渡される引数を、kprobes を使う場合よりも容易に取得することができます。

簡単な例

ここでは、kprobes の使い方を説明した **[HACK #47]** と同じく、do_execve() 関数の先頭をプローブするサンプルを示します。

```
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>

struct jprobe jp;

int jp_do_execve(char *filename, char __user *__user *argv,
                 char __user *__user *envp, struct pt_regs *regs)
{
    int cnt = 0;

    printk(KERN_INFO "filename: %s\n", filename);
    for (; *argv != NULL; argv++, cnt++)
        printk(KERN_INFO "argv[%d]: %s\n", cnt, *argv);

    jprobe_return();
    return 0;
}

static __init int init_jprobe_sample(void)
{
    jp.kp.symbol_name = "do_execve";
    jp.entry = JPROBE_ENTRY(jp_do_execve);
    register_jprobe(&jp);
```

```
    return 0;
}
module_init(init_jprobe_sample);

static __exit void cleanup_jprobe_sample(void)
{
    unregister_jprobe(&jp);
}
module_exit(cleanup_jprobe_sample);

MODULE_LICENSE("GPL");
```

おおまかなコードは、kprobesを使う場合のそれとほとんど同じですが、異なる点が3つあります。1つ目は、プローブのためのデータ構造としてjprobe構造体を使用し、そのポインタをregister_jprobe()、unregister_jprobe()に渡すことです。jprobe構造体のメンバは、以下のようにkprobe構造体とentryの2つだけです。

```
struct jprobe {
        struct kprobe kp;
        kprobe_opcode_t *entry; /* probe handling code to jump to */
};
```

struct kprobe kpのメンバで設定するのは、プローブ対象関数（上記の例ではdo_execve()）のシンボル（symbol_name）か、アドレス（addr）です。entryには、JPROBE_ENTRY()というマクロで処理されたプローブハンドラ（上記の例では、jp_do_execve()）を代入します。

2つ目は、プローブハンドラの引数を、プローブ対象の関数（上記の例ではdo_execve()）と同じにすることです。これは、printk()などで、変数の内容を取得する際、変数名をそのまま使えるため、簡潔にプローブハンドラを記述できます。kprobesを使う場合、レジスタやスタックから、引数の値を算出しなければなりません。また、その算出方法は、アーキテクチャに依存します。jprobesならば、アーキテクチャに関する詳細な知識がなくても、簡単に引数の値を調べることができます。

3つ目は、プローブハンドラの最後に、単なるreturn文ではなく、jprobe_return()を記述することです。この関数により、プローブ対象関数に復帰するので、その次のreturn文は実際には実行されません。このreturn文は、コンパイラの警告やエラーを回避するために記載されます。

まとめ

jprobesを用いて、関数の先頭をプローブにする方法を説明しました。引数の値を、

kprobes を使う場合より簡単に取得できます。

—— Kazuhiro Yamato

HACK #49 kprobes を使って、カーネル内部の任意箇所の情報を取得する

kprobes を使って、カーネル関数の任意箇所にプローブを挿入し、情報を取得する方法を説明します。

kprobes の強力な機能

[HACK #47] では、kprobes で関数の先頭にプローブを挿入し、カーネル内のグローバル変数、関数の引数、スタックトレースを表示する例を紹介しました。ただし、関数の先頭にプローブを挿入するだけなら、**[HACK #48]** で紹介した jprobes を使った方が便利な場合が多いです。それでも kprobes には jprobes にない強力な機能があります。それは、カーネルの任意アドレスにプローブを挿入できることです。また、そのプローブを任意アドレスにある命令が実行される前と後の一方、あるいは、両方で実行することができます。

任意アドレスへのプローブ挿入

カーネルは、その大部分が C 言語で記述されていますが、残念ながら kprobes では、プローブの挿入箇所として、ソース内の任意の行を指定することはできません。指定できるのは、あくまで任意のアドレスです。そのため、アセンブリコードを実際に見ながら、ソースコードにおける調べたい箇所が、ビルドされたバイナリのどのアドレスに対応するのか、また、表示すべき変数が、どのレジスタやメモリアドレスに格納されているのかをあらかじめ調べなければなりません。

以下では、do_execve() の①の部分にプローブを挿入し、kzalloc の戻り値 bprm を表示する例を示します。

```
int do_execve(char * filename,
        char __user *__user *argv,
        char __user *__user *envp,
        struct pt_regs * regs)
{
        struct linux_binprm *bprm;
        struct file *file;
        int retval;
        int i;

        retval = -ENOMEM;                              ————————⓪
```

```
bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);  ―――――①
if (!bprm)                                  ―――――②
        goto out_ret;

file = open_exec(filename);
(以下省略)
```

まず、①のアドレスを調べます。ここでアドレスとは、調査対象マシンにおける絶対論理アドレスではなく、do_execve() の先頭からの相対アドレスを意味します。以下に、crash コマンドを用いてアドレスを調べる例を示します。

crash コマンドが利用できない環境では、vmlinux ファイルやモジュール（*.ko）を objdump で逆アセンブルすることで、アドレスを調べることができます。

```
crash> dis do_execve
0xffffffff8003e290 <do_execve>: push   %r15
0xffffffff8003e292 <do_execve+2>:       mov    %rsi,%r15
0xffffffff8003e295 <do_execve+5>:       mov    $0xd0,%esi
0xffffffff8003e29a <do_execve+10>:      push   %r14
0xffffffff8003e29c <do_execve+12>:      mov    %rdx,%r14
0xffffffff8003e29f <do_execve+15>:      push   %r13
0xffffffff8003e2a1 <do_execve+17>:      mov    %rdi,%r13
0xffffffff8003e2a4 <do_execve+20>:      push   %r12
0xffffffff8003e2a6 <do_execve+22>:      mov    $0xfffffff4,%r12d
0xffffffff8003e2ac <do_execve+28>:      push   %rbp
0xffffffff8003e2ad <do_execve+29>:      push   %rbx
0xffffffff8003e2ae <do_execve+30>:      sub    $0x8,%rsp
0xffffffff8003e2b2 <do_execve+34>:      mov    2811247(%rip),%rdi        # 0xffff
ffff802ec828 <malloc_sizes+104>
0xffffffff8003e2b9 <do_execve+41>:      mov    %rcx,(%rsp)
0xffffffff8003e2bd <do_execve+45>:      callq  0xffffffff800d2528 <kmem_cache_zalloc>
0xffffffff8003e2c2 <do_execve+50>:      test   %rax,%rax
0xffffffff8003e2c5 <do_execve+53>:      mov    %rax,%rbp
0xffffffff8003e2c8 <do_execve+56>:      je     0xffffffff8003e4c3 <do_execve+563>
0xffffffff8003e2ce <do_execve+62>:      mov    %r13,%rdi
0xffffffff8003e2d1 <do_execve+65>:      callq  0xffffffff8003b769 <open_exec>
…
```

ソース上の kzalloc() はインライン関数なので、do_execve() の中にその処理が展開されて

います。アセンブリコードとソースコードは厳密に一対一対応するわけではないので、正確にその範囲を対応させるのは難しいですが、kzalloc が展開されているのは、おおよそ do_execve+28 から do_execve+45 だと思われます。これは、①の前後のソースコードをアセンブリコードと比べていけば、わかります。do_execve+2 から do_execve+17 は、引数を別のレジスタに格納しているだけなので、コンパイラが自動的に生成したこの関数の初期化部のようなものです。その次の do_execve+20 と do_execve+22 は、-12($0xfffffff4) をあるレジスタ r12d に代入しています。ENOMEM は 12 なのでここは、ソース上の⓪に対応しそうです。一方、do_execve+50 の test 命令と do_execve+56 の je 命令は、C 言語の if 文の対応する典型的なコードですので、ここは②に対応すると考えられます。

というわけで、kzalloc() の返り値、すなわち bprm の値は、do_execve+45 の kmem_cache_zalloc() の戻り値と考えて良さそうです。プローブを挿入するアドレスは、kmem_cache_zalloc() を呼び出す do_execve+45 より後ろで、かつ、関数からの戻り値が格納されている rax が変更される前であれば、どのアドレスでも構いません。ただし、命令の先頭アドレスでなければなりません。これは、上記逆アセンブリコードでいうと、do_execve+50/+53/+56/+62/+65 などです。

プローブの作成

do_execve+50 をプローブするコードの例を以下に示します。ポイントは、カーネルバージョン 2.6.19（または RHEL5）以上なら、struct kp の symbol_name メンバと offset メンバを設定し、カーネルバージョンがそれ以下なら、kallsyms_lookup_name() の戻り値に関数先頭からのアドレスを加えた値を addr メンバに設定することです。

```
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>

struct kprobe kp;

int handler(struct kprobe *p, struct pt_regs *regs) {
    printk(KERN_INFO "rax: %016lx, eflags: %08x, %rip: %016lx\n",
                    regs->rax, regs->eflags, regs->rip);
    return 0;
}

static __init int init_kprobe_sample(void)
{
    kp.symbol_name = "do_execve";
```

```
    kp.offset = 50;
    /* kernel version が 2.6.18 以下なら */
    /* kp.addr = (kprobe_opcode_t *)kallsyms_lookup_name("do_execve") + 50; */

   kp.pre_handler = handler;
   register_kprobe(&kp);

    return 0;
}
module_init(init_kprobe_sample);

static __exit void cleanup_kprobe_sample(void)
{
    unregister_kprobe(&kp);
}
module_exit(cleanup_kprobe_sample);

MODULE_LICENSE("GPL");
```

このモジュールをロードし、何かコマンドを実行して、do_execve() をコールさせた結果、筆者の環境では、以下のようなカーネルメッセージが得られました。したがって、この rax の値がソースコード上の bprm の値です。

```
kernel: rax: ffff81000b9a3800, eflags: 00000246, %rip: ffffffff8003e2c3
```

kprobes を使ったときの注意点として、ハンドラでの rip の値は、プローブが挿入されているアドレスではありません。筆者の環境では、以下のように do_execve() の絶対アドレスは、ffffffff8003e290 です。

```
# cat /proc/kallsyms | grep do_execve
ffffffff8003e290 T do_execve
```

プローブを挿入したアドレスは、関数の先頭から 50 バイト目なので、このアドレスに 50（0x32）を加えた ffffffff8003e2c2 です。一方、kprobes のハンドラで、表示させた rip の値は、それよりも 1 バイト大きい値です。これは、x86_64 や i386 アーキテクチャでは、プローブを呼び出すために、int 3 という 1 バイト長の命令がプローブ対象のアドレスに埋め込まれているためです。プローブ関数の引数 regs には、この int 3 が実行された直後のレジスタが格納されているため、プログラムカウンタ rip は、プローブの挿入アドレスよりも 1 バイト大きくなっています。

命令の実行後のプローブを挿入

kprobes の強力な点として、命令を実行した直後にもプローブを挿入できることを本 Hack の冒頭で述べました。その方法は、struct kprobes の post_handler を以下のように指定することです。post_handler メンバには pre_handler と同じ関数を指定することも、異なる関数を指定することもできます。

```
kp.post_handler = handler;
```

上記のコードを先ほどのサンプルのソースコードに追加して、実行させました。その結果が下記です。1 行目が、do_execve+50 の test 命令実行直前のプローブ結果、2 行目が test 命令実行直後のプローブ結果です。

```
kernel: rax: ffff81001e4b9a00, eflags: 00000246, %rip: ffffffff8003e2c3
kernel: rax: ffff81001e4b9a00, eflags: 00000186, %rip: ffffffff88000012
```

test 命令は 2 つのオペランドの論理積を計算し、その結果をフラグに反映します。この命令では、レジスタは変化しないので、rax の値は実行前と後で同じになっています。eflags に注目すると、いま rax と rax の論理積は 0 でないので、test 命令の実行後のゼロフラグ（6 ビット目）が落ちており、また、論理積の MSB（MostSignificant Bit：最上位ビット）が 1 であるためサインフラグ（7 ビット目）が立っていることがわかります（「**Intel アーキテクチャの基本**」**[HACK #8]** を参照）。

他に変化している eflags のビットは、割り込みフラグ（9 ビット目）とトラップフラグ（8 ビット目）です。これは、kprobes が、命令の実行直後にプローブを挿入するために CPU の TRAP 機能（1 命令の実行ごとに例外を発生させる機能）を割り込み禁止状態で使用しているためです。
また、命令実行後の rip の値は、プローブ対象アドレスとは大きく異なっています。これは、do_execve+50 の test 命令は、実行前のプローブを呼び出すための int 3 命令を上書きするときに破壊されるので、実は、そこにあった test 命令はカーネル内の他の領域に退避され、実行されているためです。rip として表示されているアドレスは、その退避された領域のアドレスです。

まとめ

kprobes を任意のアドレスに挿入し、カーネル関数のソースコード途中での変数の値を取得する方法を説明しました。

—— Kazuhiro Yamato

HACK #50 kprobesを使って、カーネル内部の任意箇所で変数名を指定して情報を取得する

kprobesを使って、カーネル関数をまるごと置き換えることにより、任意箇所で変数名を指定して情報を取得する方法を説明します。

関数の置き換え

[HACK #49] ではkprobesを使い、関数の任意の箇所にプローブを挿入する方法を説明しました。しかし、この方法では、変数の値を調べるために、アセンブリコードを解析し、レジスタやスタックを参照しなければなりません。できることなら、任意の箇所でソース上の変数名を指定してその値を表示させたいと思う場合も多いでしょう。ここではそのような方法を紹介します。

ただし、この方法はソースを変更するため、生成されるコードも大きく変更される可能性があります。そのため、調査している問題が再現しなくなる可能性があることを理解しておいてください。

プローブの作成

以下に、**[HACK #49]** と同じく、do_execve() の冒頭でポインタ bprm の値を表示させるプローブの例を示します。まず、内容を調査したいカーネル関数（この場合、do_execve()）をまるごとコピーし、シンボルのコンフリクトを避けるため、その関数名を変更します④。そして、調査のための printk() 文⑤を追加します。プローブのハンドラの中では、⑥で、regs->rip を設定することにより、プローブ終了後の復帰アドレスを my_do_execve() に変更します。⑦から⑨は実行中のプローブを中断する処理です。①から③は、④の関数をビルドするために必要な宣言等です。これらは、fs/exec.c に記載されているため、このファイルに直接記載しないとビルドできません。

```
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/binfmts.h>
#include <linux/security.h>
#include <linux/err.h>
#include <linux/kallsyms.h>
#include <linux/acct.h>
#include <linux/file.h>
#include <asm/mmu_context.h>
#include <asm/percpu.h>
```

```
#define free_arg_pages(bprm) do { } while (0)                    ①
int count(char __user * __user * argv, int max);                 ②
int copy_strings(int argc, char __user * __user * argv,
                 struct linux_binprm *bprm);                     ③

struct kprobe kp;

int my_do_execve(char * filename, char __user *__user *argv, char __user *__user *envp,
        struct pt_regs * regs)                                   ④
{
        struct linux_binprm *bprm;
        struct file *file;
        int retval;
        int i;

        retval = -ENOMEM;
        bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
        if (!bprm)
                goto out_ret;
        printk("bprm: %p, filename: %s\n", bprm, filename);      ⑤

        file = open_exec(filename);
        retval = PTR_ERR(file);
        <<以下、オリジナルのソースと同じ>>
}

int handler(struct kprobe *p, struct pt_regs *regs) {
    regs->rip = (unsigned long)my_do_execve;                     ⑥
    reset_current_kprobe();                                      ⑦
    preempt_enable_no_resched();                                 ⑧
    return 1;                                                    ⑨
}

static __init int init_kprobe_sample(void)
{
    kp.symbol_name = "do_execve";
    /* kernel version が2.6.18以下なら */
    /* kp.addr = (kprobe_opcode_t *)kallsyms_lookup_name("do_execve"); */

   kp.pre_handler = handler;
```

```
    register_kprobe(&kp);

    return 0;
}
module_init(init_kprobe_sample);

static __exit void cleanup_kprobe_sample(void)
{
    unregister_kprobe(&kp);
}
module_exit(cleanup_kprobe_sample);

MODULE_LICENSE("GPL");
```

プローブ用モジュールのインストールと問題の回避

上記のソースは、カーネル標準の方法でモジュール（ここでは、kpro3.koという名前とします）としてビルドできます。しかし、insmodすると、筆者の環境では、下記のエラーが出力されました。一般的に、この問題はよく発生します。（コピーする調査対象のカーネル関数によっては、このようなエラーが出力されない場合ももちろんありますが）これらは、モジュールからの使用が許可されていない関数やデータです。上記ソースでは、（モジュールではなく）カーネルに組み込まれて使われるdo_execve()をコピーしたため、このような問題が発生しました。

```
kernel: kpro3: Unknown symbol per_cpu__current_kprobe
kernel: kpro3: Unknown symbol init_new_context
kernel: kpro3: Unknown symbol mm_alloc
kernel: kpro3: Unknown symbol acct_update_integrals
kernel: kpro3: Unknown symbol sched_exec
kernel: kpro3: Unknown symbol __mmdrop
kernel: kpro3: Unknown symbol count
kernel: kpro3: Unknown symbol copy_strings
```

この制限は、そもそもLinuxのポリシーなので一般的な解決方法はありません。ただし、特定の環境のためだけのデバッグであれば、次のようにモジュールの未解決シンボルにアドレスを与えることで回避できます。

```
# cat addrs.dat
SECTIONS
{
  per_cpu__current_kprobe = 0xffffffff804052a0;
  init_new_context        = 0xffffffff8006d7f7;
  acct_update_integrals   = 0xffffffff8004ecd8;
  sched_exec              = 0xffffffff800457be;
  __mmdrop                = 0xffffffff8008d5e2;
  count                   = 0xffffffff80039821;
  copy_strings            = 0xffffffff80017381;
  mm_alloc                = 0xffffffff8004c098;
}

# ld -r -o kpro3a.ko kpro3.ko -R addrs.dat
```

addrs.dat は、シンボルとそのアドレスを記載したファイルです。各シンボルの値は、/proc/kallsyms や vmlinux ファイルから、以下のように取得可能です。

```
# nm vmlinux | grep per_cpu__current_kprobe
ffffffff804052a0 D per_cpu__current_kprobe
# cat /proc/kallsyms | grep init_new_context
ffffffff8006d7f7 T init_new_context
```

このようにして、アドレスを強引に解決したモジュールを insmod し、何かしらのコマンドを実行して、do_execve() をコールすると、次のようなカーネルメッセージが得られます。

```
kernel: bprm: ffff81001ff1a800, filename: /bin/ls
```

まとめ

kprobes を使って、カーネル関数をまるごと置き換えることによって、任意箇所で変数名を指定して情報を取得する方法を説明しました。

—— Kazuhiro Yamato

HACK #51 KAHOを使い、コンパイラによって Optimized outされた変数の値を取得する

KAHOを使って、プロセスの関数をまるごと置き換えることにより、最適化され、GDBで検査できなくなった変数の値を取得する方法を説明します。

最適化と変数の表示

本書のいたるところで説明されているように、GDBを使うと、任意の箇所でプロセスを停止して、その時のレジスタやメモリの値を得ることができます。デバッグ対象プロセスの実行ファイルにデバッグ情報があれば（-gオプション付きでビルドされていれば）、変数名を指定して、その値を表示することもできます。しかしながら、最適化によって、変数の値を取得できない場合があります。例えば、次のようなソースコードを最適化オプション（-O2）を指定してビルドしてみましょう。

[calc.c]

```
#include <stdio.h>
#include <stdlib.h>

int func(int x)
{
    int a, a0, a1, a2;
    a0 = x * x * 2 + 1;
    a1 = x + 2 - a0;
    a2 = x / 2 + a1;
    a = a0 + a1 + a2;
    printf("a: %d\n", a);
    return a;
}

int main(void)
{
    int i = 1;
    while(1) {
        i = func(i);
        sleep(3);
    }
    return EXIT_SUCCESS;
}
```

このプログラムの func() 中の変数 a0 や a1 を表示させようとしたところ、筆者の環境では、次のように <value optimized out> と表示され、変数の値を表示することができませんでした。

```
(gdb) b func
Breakpoint 1 at 0x4004c0: file opt.c, line 7.
(gdb) run
Starting program: /home/kyamato/DebugHacks/kyamato/chapter5/kaho.work/opt2

Breakpoint 1, func (x=1) at opt.c:7
7           a0 = x * x * 2 + 1;
(gdb) n
10          a = a0 + a1 + a2;
(gdb) n
8           a1 = x + 2 - a0;
(gdb) n
7           a0 = x * x * 2 + 1;
(gdb) n
10          a = a0 + a1 + a2;
(gdb) n
5       {
(gdb) n
10          a = a0 + a1 + a2;
(gdb) n
11          printf("a: %d\n", a);
(gdb) p a0
$1 = <value optimized out>
(gdb) p a1
$2 = <value optimized out>
```

多くのプログラムは最適化を有効にしてビルドされているため、このような事態は、比較的よく発生します。GDB でステップ実行を行う際、Optimized out されている変数があると、動作の確認が妨げられ、効率よくデバッグを進められません。簡単な解決方法のひとつは、ソースを変更して、a0 や a1 なども printf() で表示させることです。ただし、この方法は、起動や初期化に時間のかかるプログラム（X11 や多数のプロセスからなる大規模アプリケーションなど）の場合、再ビルドし、実行させた後、実際に値が表示されるまでの待ち時間が長く、スムーズにデバッグを進められないことがあります。

プロセスの関数の置き換えによるデバッグ

上記のような状況でのデバッグには、ライブパッチ、またはランタイム・バイナリ・パッ

チャとよばれるプログラムを使ってみる価値があります。この種のプログラムは、元々高可用システムで、バグやセキュリティホールをプロセスの再起動を行うことなく、修正するために使用されます。ここでは、このようなプログラムのひとつである KAHO を使って、稼働中のプロセスの変数を取得する方法を説明します。KAHO は、関数単位でプログラムを置き換えることができます。そのため、表示させたい変数を含む関数を、printf() などのデバッグ出力を含む関数にまるごと置き換えることで、プロセスを再起動させることなく、すなわち、初期化等の時間を待つことなく、変数の値を出力させることができます。

[HACK #50] では、カーネル関数をまるごと置き換えました。この方法は、そのユーザプロセス版と言えます。そのため、**[HACK #50]** と同様に、printf() 文を挿入したことによって、生成されるコードも変更される可能性があり、調査している問題が再現しなくなることがあることを理解しておいてください。

KAHO のインストール（Fedora10）

ここでは、KAHO を Fedora10 にインストールする方法を説明します。KAHO は、カーネルパッチと、コマンドラインツールから構成されます。それは、以下のサイトからダウンロードできます。

```
http://sourceforge.net/projects/kaho-01/
```

まず、カーネルのソースをダウンロードして、下記のようにパッチを適用します。

```
# yumdownloader --source kernel
# rpm -ihv kernel-2.6.27.9-159.fc10.src.rpm
```

spec ファイルを編集し、KAHO のカーネルパッチが適用されるようにします。spec ファイルは ~/rpmbuild/SPECS/kernel.spec に展開されています。KAHO 機能を組み込んだカーネルであることがわかるように spec ファイルの 15 行目あたりにある buildid の定義をコメントアウトし、.local を .kaho に変更します。

```
%define buildid .kaho
```

1360 行目あたりの # END OF PATCH APPLICATIONS 行の前に次の行を挿入してください。

```
ApplyPatch kaho_kernel_fedora10.patch
```

次に、KAHO のカーネルパッチを SOURCES/ ディレクトリに配置し、カーネルをビルド、

インストールします。

```
# cp ~/kaho_kernel_fedora10.patch ~/rpmbuild/SOURCES
# rpmbuild -bb rpmbuild/SPECS/kernel.spec
# rpm -ihv rpmbuild/RPMS/kernel-2.6.27.9.x86_64.rpm
```

インストールできたら、新しいカーネルで再起動してください。

KAHO は、UTRACE という機能を使用しています。この機能は、RHEL、Fedora、Cent OS や Asianux には取り込まれていますが、まだメインラインにはマージされていません。そのため、メインラインカーネルで KAHO を使う場合は、UTRACE パッチと、KAHO のパッチの両方を適用する必要があります。
UTRACE パッチは、下記の Web ページから入手できます。
http://people.redhat.com/roland/utrace/
また、Fedora10 のカーネルビルドは、下記のサイトが参考になります。
http://www.atmarkit.co.jp/flinux/rensai/linuxtips/a113rebuild.html

コマンドラインツールに関しては、上記サイトに Fedora10 用のバイナリ RPM が用意されているので、それをダウンロードして、インストールすれば完了です。

KAHO を使ったデバッグ

次のような 3 つのファイルを用意します。

```
/home/kaho_dbg/calc
/home/kaho_dbg/1/debug.so
/home/kaho_dbg/1/debug.cmd
```

calc は、冒頭で紹介した calc.c を最適化オプション（-O2）を指定してビルドした実行ファイルです。debug.so は、次のソースコード debug.c を、calc と同じビルドオプションでビルドした共有ライブラリです。debug.c では、デバッグ対象の関数 func() を、calc.c からまるごとコピーし、関数名を func_debug() に変更しています。そして、変数の値を出力するための printf() 文①を追加しています。以下にそのソースをビルド例と共に示します。

```
# cat debug.c
#include <stdio.h>

int func_debug(int x)
{
```

```
    int a, a0, a1, a2;
    a0 = x * x * 2 + 1;
    a1 = x + 2 - a0;
    a2 = x / 2 + a1;
    a = a0 + a1 + a2;
    printf("[Debug]a0: %d, a1: %d, a2: %d\n", a0, a1, a2); ———①
    printf("a: %d\n", a);
    return a
}
```

```
# gcc -o debug.so debug.c -fPIC -shared -O2 -g
```

debug.cmd は、KAHO の動作を記述したファイルで以下のような書式を持ちます。

```
patch-file  置き換える関数のある共有ライブラリ名
J func 置き換える関数の名前（シンボル名）
```

KAHO は、実は関数以外にも、データを書き換える機能も持っています。2行目の J func というのは、書き換え対象がデータでなく関数であることを指定するためのキーワードです。

以降の実行例では、次の内容のファイルを使用します。

```
# cat debug.cmd
patch-file debug.so
J func func_debug
```

では、実際に動作させてみましょう。表示の便宜上、2つの端末（端末A、Bとします）を用意して、まず、端末Aで calc を起動させます。

[端末 A での操作]

```
# ./calc
a: 3
a: -8
a: -145
…
```

次に端末Bで以下のように KAHO コマンドを2回入力します。1回目の KAHO コマンドは、置き換えるべき関数 func_debug() を calc のプロセスのメモリ空間にロードします。ただし、

この状態では、まだ置き換えは行われていません。2 回目の KAHO コマンドで関数を実際に置き換えます（この操作をアクティベーションと呼びます）。なお、KAHO コマンドの引数の最後に指定している「1」は、$KAHO_HOME/1 という名称のディレクトリにあるコマンドファイル（.cmd 拡張子を持つファイル）を読み込めという意味です。

[端末 B での操作]

```
# cd /home/kaho_dbg
# export KAHO_HOME=`pwd`
# kaho -l `pidof calc` 1
3693,1,"loaded
# kaho -a `pidof calc` 1
3693,1,"activated"
```

上記の操作を行うと、確認したい変数の値を端末 A に表示させることができました。

[アクティベーション後の端末 A]

```
…
[Debug]a0: -795409887, a1: -1135769211, a2: -2101358761
a: 262429437
[Debug]a0: -1258439661, a1: 1520869100, a2: 1652083818
a: 1914513257
…
```

次のコマンドにより、置き換えられた関数を元に戻すことができます。

```
# kaho -d `pidof calc` 1
3693,1,"deactivated"
# kaho -u `pidof calc` 1
3693,1,"unloaded
```

参考文献

- A Runtime Code Modification Method for Application Programs
  http://ols.fedoraproject.org/OLS/Reprints-2008/yamato-reprint.pdf
- ランタイム・バイナリ・パッチャ（KAHO）の開発
  http://blog.miraclelinux.com/yume/files/YLUG-2008-0225update.pdf

まとめ

ランタイム・バイナリ・パッチャのひとつである KAHO を使うと、プロセスの関数を

まるごと置き換えることができます。これを応用して、最適化により Optimized out された変数の値を取得する方法を説明しました。

—— Kazuhiro Yamato

HACK #52 systemtap を使って動作中のカーネルをデバッグする（その 1）

タイミング計測をするサンプルを例に systemtap の使用方法を説明します。

はじめに

systemtap は kprobes を利用して作られたツールです。C 言語に似た独自のスクリプト言語を用いてプローブハンドラを作成します。スクリプトで書かれたプローブを専用のパーサーが C 言語に変換し、自動的にカーネルモジュールを作成してくれます。その際、作成したコードの安全性チェックをしてくれたり、tapset と呼ばれるスクリプト群の中に便利な関数があらかじめ用意されていたりと、kprobes より比較的使いやすくなっています。

準備

systemtap を使用するにはデバッグ情報付きでコンパイルされたカーネルが必要です。また、stap コマンドが自動的にカーネルモジュールをビルドしますので、カーネルヘッダもインストールしておく必要があります。筆者は Fedora9 を使用しました。カーネルバージョンは 2.6.27.7-53.fc9、systemtap のバージョンは 0.8-1.fc9 です。Fedora9 では、次のコマンドにて必要なパッケージをインストールします。

```
# yum install kernel-devel kernel-headers
# debuginfo-install kernel
```

サンプルスクリプト

ここでは次のようなスクリプトを使います。これは systemtap に付属していた sleeptime.stp というサンプルスクリプトをベースに、少し手を加えたものです。nanosleep() システムコールが実際にどれくらいの時間スリープしたのかを計測するスクリプトです。

```
#!/usr/bin/stap -v

 /*
  * Format is:
  * 12799538 3389 (xchat) nanosleep: 9547
```

```
  * 12846944 2805 (NetworkManager) nanosleep: 100964
  * 12947924 2805 (NetworkManager) nanosleep: 100946
  * 13002925 4757 (sleep) nanosleep: 13000717
  */

global start
global entry_nanosleep
global entry_nanosleep_restart

function timestamp:long() {
  return gettimeofday_us() - start
}

function proc:string() {
  return sprintf("%d (%s)", pid(), execname())
}

probe begin {
 start = gettimeofday_us()
}

probe syscall.nanosleep {
  if (uid() != 500) next;
  t = gettimeofday_us(); p = pid()
  entry_nanosleep[p] = t
}

probe syscall.nanosleep.return {
  if (uid() != 500) next;
  t = gettimeofday_us(); p = pid()
  elapsed_time = t - entry_nanosleep[p]
  printf("%d %s nanosleep: %d\n",  timestamp(), proc(), elapsed_time)
  delete entry_nanosleep[p]
}

probe kernel.statement("hrtimer_nanosleep@kernel/hrtimer.c:1551") {
  if (uid() != 500) next;
  printf("%d %s nanosleep is interrupted.\n", timestamp(), proc());
  p = pid();
  entry_nanosleep_restart[p] = entry_nanosleep[p];
}

probe kernel.function("hrtimer_nanosleep_restart").return {
```

```
    if (uid() != 500) next;
    t = gettimeofday_us(); p = pid()
    elapsed_time = t - entry_nanosleep_restart[p]
    printf("%d %s nanosleep_restart: %d\n",  timestamp(), proc(), elapsed_time)
    delete entry_nanosleep_restart[p];
  }
```

時間を計測する

systemtap を使ったライブデバッグでわりと便利な使い方として、ある処理に要する時間の測定があります。ただし、数マイクロ秒以下の微妙なタイミング計測には不向きであることに注意してください。スクリプトの作りやマシン性能にもよりますが、systemtap 自体のオーバーヘッドによる影響が無視できなくなるからです。

カーネル内部処理には非同期のイベントをトリガーにして処理を進めるものがたくさんあります。本 Hack で取り上げる nanosleep() は、タイマの割り込みイベントなどをトリガーにしてスリープ状態から起床します。そしてタイマの割り込みイベントは非同期のイベントで、システムの負荷状況によってそのタイミングが影響を受けます。また、スリープしているタスクへのシグナル配信のような、タイマの割り込み以外の非同期イベントにも nanosleep() は影響を受けます。その結果、通常は要求したスリープ時間より若干遅いタイミングで nanosleep() コールから復帰します。今回はこの実際にスリープする時間を計測します。

方法は単純で、nanosleep() を実行した直後の時刻と nanosleep() から復帰する時刻を保存しておき、復帰時にユーザコンソールへその差分を経過時間として表示させるだけです。保存する時刻は tapset で提供されている関数 gettimeofday_us() を使用します。

プローブポイントの定義

プローブを挿入する場所をプローブポイントと言います。プローブポイントの定義方法はたくさんありますが、ここではその中からよく使う定義をいくつか紹介します。

```
probe begin
probe end
```

スクリプト起動時、終了時に動作させるハンドラを定義します。スクリプト内のグローバル変数の初期化や、終了時に収集したログを整形して表示させる時などに使います。

```
probe kernel.funcion(" 関数名 ")
probe kernel.funcion(" 関数名 ").return
```

それぞれ、"関数名" で指定したカーネル内関数がコールされた時、リターンする時に実行されるプローブを定義する時に使用します。"関数名" の部分には次のようにワイルドカード（*）を使用することもできます。

```
probe kernel.function("*init*")
probe kernel.function("*init*@kernel/sched.c")
```

1つ目の例ではカーネルの初期化関数すべてにプローブを挿入します。2つ目の例では、kernel/sched.c に定義されている初期化関数すべてにプローブを挿入します。このように関数名に続けてファイル名を指定することで対象範囲を限定する記述も可能です。異なるファイルに同じ名前の関数が存在する場合、この表記を使うことで対象とする関数を1つに限定できます。またファイル名の部分にワイルドカードを使用することもできます。

カーネルモジュールにプローブを定義する場合は次のように記述します。

```
probe module("モジュール名").function("関数名")
probe module("モジュール名").function("関数名").return
```

これまで kernel と書いていた箇所を module("モジュール名") と変更するだけで、あとは全く同じです。

```
probe syscall.システムコール名
probe syscall.システムコール名.return
```

プローブ対象がシステムコールの時はこのように記述できます。前述の kernel.function() と動作は同じで、システムコールの場合はどちらの記述を使っても構いません。

```
probe kernel.statement("関数名@ファイル名:行番号")
probe kernel.statement(アドレス)
```

これらは関数の途中にプローブを入れる場合に使用します。関数内で、ある if 文の条件にマッチした場合だけをデバッグしたい時に便利です。ただし、行番号で指定する書き方では、カーネルソースコードの作りによっては、実際のプローブ挿入位置が少しずれてしまう場合があるので注意してください。これは stap コマンドがカーネルのデバッグ情報から対応するアドレスを割り出してプローブ位置を決めるからです。アドレスを直接指定する書き方では、それが正しく命令境界にアラインされているアドレスであれば、意図どおりの場所に挿入されます。

カーネルモジュールを対象とする場合も同様に次のように記述します。

```
probe module("モジュール名").statement("関数名@ファイル名:行番号")
probe module("モジュール名").statement(アドレス)
```

本Hackの例ではこの表記を使って、nanosleep()がシグナルに割り込まれた場合に動くコードにプローブを挿入しています。

```
probe kernel.statement("hrtimer_nanosleep@kernel/hrtimer.c:1551")
```

これによりプローブが挿入される部分のカーネルソースを次に引用します。このような手法を使うと、関数内にあるif文の判定結果が真であったか、偽であったかがわかるようなスクリプトを作成することも可能です。

[kernel/hrtimer.c]

```
1527 long hrtimer_nanosleep(struct timespec *rqtp, struct timespec __user *rmtp,
1528                        const enum hrtimer_mode mode, const clockid_t clockid)
1529 {
…
             /* シグナルに割り込まれるとこのif文が偽になる */
1536         if (do_nanosleep(&t, mode))
1537                 goto out;
…
1551         restart = &current_thread_info()->restart_block;
1552         restart->fn = hrtimer_nanosleep_restart;
1553         restart->nanosleep.index = t.timer.base->index;
1554         restart->nanosleep.rmtp = rmtp;
1555         restart->nanosleep.expires = t.timer.expires.tv64;
1556
1557         ret = -ERESTART_RESTARTBLOCK;
1558 out:
1559         destroy_hrtimer_on_stack(&t.timer);
…
```

最後に、このままではシステム上で実行されるすべてのnanosleep()がフックされてしまうため少々使いにくいです。そこで次の記述を各プローブハンドラの冒頭に追加し、テスト用のユーザ（UID=500）が発行するnanosleep()だけを計測対象とすることにします。

```
  if (uid() != 500) next;
```

実行してみる

スクリプトを実行するコマンドは stap です。デフォルトではほとんどメッセージを出さないので、プローブが有効化されたかどうかがわかりにくいのです。そのため筆者はかならず verbose オプション（-v）をつけて実行するようにしています。

```
# stap -v sleeptime.stp
Pass 1: parsed user script and 45 library script(s) in 230usr/10sys/243real ms.
Pass 2: analyzed script: 6 probe(s), 8 function(s), 15 embed(s), 3 global(s) in 450usr/320sys/778real
ms.
Pass 3: translated to C into "/tmp/stap7JYrBT/stap_2aa0c877c1a26e0304cc924aa0dcfbb3_13404.c" in
370usr/620sys/988real ms.
Pass 4: compiled C into "stap_2aa0c877c1a26e0304cc924aa0dcfbb3_13404.ko" in 5290usr/900sys/6210real ms.
Pass 5: starting run.
```

上記のメッセージが表示されればすでにプローブは有効化されています。別ターミナルからテスト用のアカウントで nanosleep() を発行し、10 秒スリープしてみます。

```
$ usleep 10000000
```

すると stap コマンドを実行したターミナル上に次のようなメッセージが表示されます。

```
13304586 19724 (usleep) nanosleep: 10000008
```

メッセージの表示形式は次のようになっています。表示される経過時間はすべてマイクロ秒単位です。

```
プローブ有効化からの経過時間　PID （コマンド名） nanosleep: 実経過時間
```

ということで遅延は 8usec なのでたいした誤差もなく、優秀なシステムだとわかりました。

では、わざとシグナルで中断させた場合にどうなるか確認してみます。先ほどと同じように usleep コマンドで 10 秒間スリープさせます。ただし、途中で SIGSTOP を送り nanosleep() を中断させて、そのままおおよそ 10 秒以上待ってから SIGCONT を送りました。

```
11920140 26702 (usleep) nanosleep is interrupted.
11920148 26702 (usleep) nanosleep: 1233996
23249021 26702 (usleep) nanosleep_restart: 12562868
```

nanosleep() が中断されたことを示すメッセージが出ています。nanosleep() 発行後約 1.2 秒後に SIGSTOP を受信したことが 2 番目のメッセージからわかります。3 番目のメッセージから、nanosleep() からリターンするのに約 12.5 秒かかったことがわかります。つまり筆者の体内時計は実際より 2.5 秒も遅れていたということです。

確認ができたら systemtap を終了します。Ctrl + c を押すとロードしたプローブモジュールがアンロードされ、systemtap を終了させることができます。

まとめ

nanosleep() システムコールの実経過時間の計測を例に、systemtap の使い方について説明しました。systemtap は kprobes を使用しているため、kprobes でできることのほとんどは systemtap でも実現できます。

参考

man ページに加え、systemtap 付属のサンプルスクリプトおよび tapset が参考になります。またプロジェクトのページには tutorial などのドキュメント類が豊富に揃っています。

- systemtap 付属のサンプルスクリプト
  /usr/share/doc/systemtap-<version>/examples/
- systemtap 付属の tapset
  /usr/share/systemtap/tapset/
- SystemTap プロジェクトページ
  http://sourceware.org/systemtap/documentation.html

—— Toyo Abe

HACK #53 systemtap を使って動作中のカーネルをデバッグする（その 2）

systemtap を使ってコールトレースや構造体データの内容を調べる方法について説明します。

カーネルコードを読んでいると、注目している関数がどこからコールされるのか、注目しているデータにどんな値が入っているのかを調べたくなります。本 Hack では、「systemtap を使って動作中のカーネルをデバッグする（その 1）」**[HACK #52]** で紹介した systemtap スクリプト（sleeptime.stp）を拡張して、systemtap を使ったコールトレースの調べ方とカーネル内データの参照方法を説明します。実行環境は Fedora9 でカーネルバー

ジョンは 2.6.27.7-53.fc9、systemtap のバージョンは 0.8-1.fc9 です。

sleeptime.stp の拡張

sleeptime.stp に以下のコードを追加しました。その他は **[HACK #52]** のまま、変更はありません。

```
#include <linux/thread_info.h>
function res_expires:long(res:long) %{
  struct restart_block *restart = (struct restart_block *)(THIS->res);
  THIS->__retvalue = restart->nanosleep.expires;
%}

probe kernel.function("hrtimer_nanosleep_restart") {
  if (uid() != 500) next;
  printf("%d %s Call trace:\n", timestamp(), proc());
  print_backtrace();
  printf("restart->nanosleep.expires = %u\n", res_expires($restart));
}
```

コールトレースを調べる

カーネルコードを読んでいると、複雑すぎて注目している関数がどのようにコールされているのか、ソースだけではわかりにくい、あるいは調べるのに時間がかかりすぎてしまうといったことがよくあります。そんな時はカーネルコード中に WARN_ON(1) などを書き込んで、カーネルを再コンパイルすることがあると思います。ただカーネル再コンパイルというのは結構面倒くさいものです。ところが systemtap を使えばプローブハンドラを書くだけで、そのような手間から解放されます。ハンドラに記述するのは次の 1 行だけです。

```
print_backtrace();
```

本 Hack の例では、nanosleep() がシグナルに割り込まれた後、SIGCONT で再開する場所（hrtimer_nanosleep_restart()）にこれを記述しています。

カーネルの内部データを参照する

関数内で使用されている変数は $変数の記述形式で参照できます。ただし、これもカーネルコードの作りやプローブポイントの位置によっては参照できない場合もあるので注意してください。どうしても「$変数」の形式で参照できない場合は、アドレスを直接指定するか、レジスタを直接参照しなければならなくなります。その場合、どのアドレス、ど

のレジスタを参照するのかはカーネルバイナリを逆アセンブルして自分で探す必要があります。

本 Hack の例では $restart として、hrtimer_nanosleep_restart() の引数 restart の値を参照しています。

[kernel/hrtimer.c]

```
1500 long __sched hrtimer_nanosleep_restart(struct restart_block *restart)
1501 {
1502         struct hrtimer_sleeper t;
1503         struct timespec __user  *rmtp;
1504         int ret = 0;
1505
1506         hrtimer_init_on_stack(&t.timer, restart->nanosleep.index,
1507                               HRTIMER_MODE_ABS);
1508         t.timer.expires.tv64 = restart->nanosleep.expires;
1509
1510         if (do_nanosleep(&t, HRTIMER_MODE_ABS))
1511                 goto out;
```

中断された nanosleep() を再開する際には 1508 行目にあるとおり、restart->nanosleep.expires に保存された値を起床時刻として do_nanosleep() をコールします。この値をプローブハンドラで表示することにします。こういったカーネル内部の構造体メンバを参照する時はスクリプト内で C 言語を使います。

スクリプト内で C 言語を使う

systemtap スクリプトから C 言語を使うには 2 つポイントがあります。

C 言語で書かれた関数を定義する

次のような書式で関数を定義します。

```
function 関数名:返り値の型 ( 引数:引数の型, ... ) %{ C言語で書かれた処理 %}
```

本 Hack の例では、restart_block 構造体が定義されているカーネルヘッダ <linux/thread_info.h> をインクルードし、引数で与えられたポインタから restart->nanosleep.expires を返す res_expires() 関数を定義しています。

systemtap を guru モードで実行する

スクリプトに C 言語が含まれる場合、guru モードにしなければ実行できなくな

ります。guru モードにするには stap コマンド実行時に -g オプションを付加します。ただし、guru モードにすると systemtap が行う安全性チェック機能が無効化されてしまいます。ロックで保護されるべきデータにアクセスする場合などは、ユーザがロックを記述しなければなりません。なお、スクリプト内で排他処理を記述する場合などは注意が必要です。プローブハンドラがスピンロック獲得待ちに陥ってしまったり、スリープしたりしないよう注意して設計しなければなりません。したがって一般的には、ロックが必要な場合は trylock を使用し、獲得できなければエラー終了させるようにハンドラを記述すべきです。

実行してみる

前述のとおり、stap コマンドに -g オプションを付加して実行します。**[HACK #52]** と同様に、別ターミナルからテスト用アカウントで usleep コマンドで 10 秒スリープさせています。さらに SIGSTOP と SIGCONT を送り、本 Hack で追加したプローブハンドラを動作させています。

```
# stap -vg sleeptime.stp
Pass 1: parsed user script and 45 library script(s) in 230usr/10sys/244real ms.
Pass 2: analyzed script: 6 probe(s), 10 function(s), 15 embed(s), 3 global(s) in 490usr/320sys/818real
ms.
Pass 3: translated to C into "/tmp/stapvOIXZ1/stap_5879bfa558535efa4dced96a1adff5e3_13896.c" in
370usr/640sys/1018real ms.
Pass 4: compiled C into "stap_5879bfa558535efa4dced96a1adff5e3_13896.ko" in 5320usr/870sys/6219real ms.
Pass 5: starting run.
9144183 27784 (usleep) nanosleep is interrupted.
9144192 27784 (usleep) nanosleep: 1162593
11000541 27784 (usleep) Call trace:
 0xffffffff812bff5c : hrtimer_nanosleep_restart+0x1/0x62 [kernel]
 0xffffffff812c2870 : kretprobe_trampoline_holder+0x4/0x50 [kernel] (inexact)
 0xffffffff8101024a : sys_rt_sigreturn+0x558/0x189e [kernel] (inexact)
 0xffffffffff000000 : packet_exit+0x7d9e988a/0x7dfe988a [kernel] (inexact)
 0xfffffffffff00000 : vgetcpu+0x9ef800/0x0 [kernel] (inexact)
 0xffffffffffffff00 : vgetcpu+0x9ff700/0x0 [kernel] (inexact)
 0xffffffffffffffff : vgetcpu+0x9ff7ff/0x0 [kernel] (inexact)
restart->nanosleep.expires = 990839225309246
17981629 27784 (usleep) nanosleep_restart: 10000029
```

hrtimer_nanosleep_restart() に挿入したプローブハンドラによってコールトレースが表示されています。コールトレース内に kretprobe_trampoline_holder() が表示されていますが無

```
sys_rt_sigreturn()
  └→ hrtimer_nanosleep_restart()
```

図 6-1 SIGCONT によって nonsleep() が再開する際のコールフロー

視して構いません。これは関数リターン時に起動するプローブも挿入しているために表示されています。以上からコールフローは図 6-1 のようになっていることがわかります。

また、restart->nanosleep.expires の値も参照できていることがわかります。

まとめ

systemtap を使って、コールトレースやカーネル内部のデータにアクセスする方法を紹介しました。これらはカーネルコード内に printk() や WARN_ON() を追加するデバッグ手法の一部を代替できる方法です。カーネルを再コンパイルすることなしにこうしたデバッグコードを仕込むことができるため、慣れてしまえば作業効率がアップするはずです。

参考

- systemtap 付属のサンプルスクリプト
  /usr/share/doc/systemtap-<version>/examples/
- systemtap 付属の tapset
  /usr/share/systemtap/tapset/
- SystemTap プロジェクトページ
  http://sourceware.org/systemtap/documentation.html

—— Toyo Abe

HACK #54

/proc/meminfo でわかること

/proc/meminfo に含まれるシステムのメモリに関する情報とメモリリーク時に変化する項目を説明します。

/proc/meminfo

/proc/meminfo から、カーネルとプロセスを含めたシステム全体のメモリ使用状況を取得することができます。以下に主要な項目の説明をします。

表 6-1　/proc/meminfo の表示項目

項目	説明
MemFree	空きメモリの合計サイズ。
Buffers	バッファ（ブロックデバイスのデータのキャシュ）の合計サイズ。デバイスファイル（/dev/sda1 など）に対して読み書きを行った場合、この値は、その読み書きサイズと同程度増加します。通常のファイルを読み書きした場合も、ファイルシステムドライバが、デバイスのスーパーブロックや i-node ブロックをアクセスするため、少し増加します。
Cached	ページキャシュ（通常ファイルのキャシュ）の合計サイズ。ファイルを読み書きすると増加します。空きメモリが不足するまで通常、保持されます。また、このサイズには、Buffers と SwapCached は含まれません。
SwapCached	ページアウトされていたデータが、ページインした後も、スワップデバイスに残っているページの合計サイズ。空きメモリが足りなくなった場合に、I/O を省略して、そのまま解放できるサイズの合計を意味します。
Active	Active な LRU リストにつながっているページの合計サイズ。
Inactive	Inactive な LRU リストにつながっているページの合計サイズ。
Mapped	ファイルをマップしているページの合計サイズ。起動しているプロセスの種類とコード量に比例して増加します。また、MAP_SHARED フラグを指定してファイルを mmap() を実行した場合も増加します。
Slab	スラブアロケータのメモリ使用量。スラブアロケータは、比較的少量（数十バイトから数メガバイト）のメモリをカーネルやドライバの要求に応じて、メモリ割り当てや、解放を行います。
PageTables	ページテーブルに使用されているメモリの合計サイズ。プロセスの使用するアドレス空間の合計が大きいほど、この値も大きくなります。
Committed_AS	プロセスにコミットされているメモリサイズの合計。まだ、実ページがアサインされていないエリアも含みます。
AnonPages	無名リージョンに属するページの合計サイズ。プロセスのヒープ領域として主に使用されます。

メモリリークの目安

上記では、それらの項目が増加するおおよそのタイミングも述べましたが、実際のところ、増加されたメモリが減少するタイミングは複雑であり、一概に記述することは困難です。そのため、特定の項目だけに注目すれば、すぐメモリリークと断定できるものはありません。それでも、Committed_AS は、プロセスのメモリリークの目安になります。Committed_AS の値が、想定される値よりも多い場合、メモリリークが疑われます。

実際に、次のプログラムを実行させ、その時の Comitted_AS、MemFree、SwapFree、AnonPages、Cached の値を測定しました。このプログラムは、最大 4MB までのランダムなサイズのメ

モリアロケーションを5つのスレッドが行います。その際、一定の割合（10%）でメモリをリークします。

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include <string.h>

const float Pleak = 0.1;
const float Paccess = 0.5;
const float MinAlloc = 1;
const float MaxAlloc = 4*1024*1024;

double frand(void)
{
    return ((double)rand())/RAND_MAX;
}

size_t calc_size(void)
{
    double r = frand();
    double a = pow(MaxAlloc/MinAlloc, r) * MinAlloc;
    return (size_t)a;
}

int leak(void)
{
    if (frand() < Pleak)
        return 1;
    return 0;
}

int access(void)
{
    if (frand() < Paccess)
        return 1;
    return 0;
}
```

```
void *thr_func(void *arg)
{
    while (1) {
        size_t s = calc_size();
        void *p = malloc(s);
        if (p == NULL) {
            printf("Failed to malloc: %d\n", s);
            return NULL;
        }

        if (!access())
            memset(p, 0xaa, s);

        if (!leak())
            free(p);

        sleep(1);
    }
}

int main(void)
{
    pthread_t t1, t2, t3, t4;
    pthread_create(&t1, NULL, thr_func, NULL);
    pthread_create(&t2, NULL, thr_func, NULL);
    pthread_create(&t3, NULL, thr_func, NULL);
    pthread_create(&t4, NULL, thr_func, NULL);
    thr_func(NULL);

    return EXIT_SUCCESS;
}
```

このプログラムを、物理メモリ 1024MB、スワップ領域 1024MB 搭載の x86_64 アーキテクチャのマシン上で、2.6.18 カーネルを使って実行しました。その結果を図 6-2 に示します。時間の経過につれて、Committed_AS の値は、ほぼ単調に増加していきます。これは、メモリリークを起こしているからです。プログラムを実行させた当初は、Committed_AS の増加にしたがって MemFree が減少していきますが、MemFree がほぼ 0 になると、今度は Cached が減少し始めることがわかります。さらに Cached もほぼ 0 になると今度は、スワップが使用されはじめ、SwapFree がどんどん減少します。注目すべきは、スワップが使用されはじ

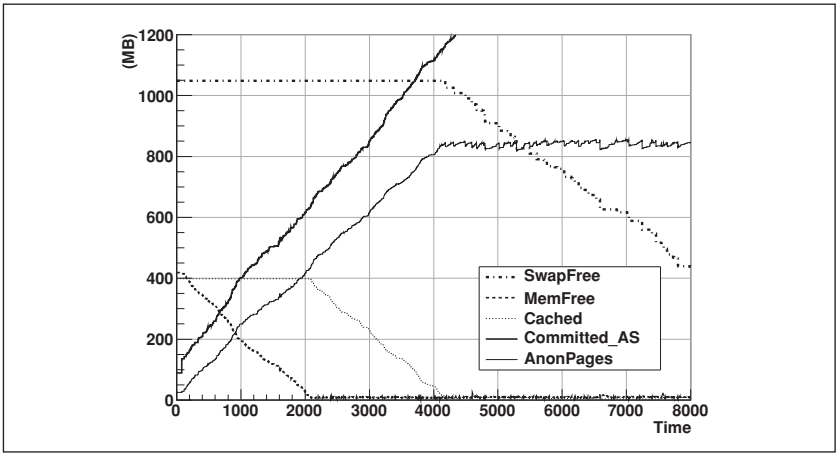

図 6-2 プログラム実行結果

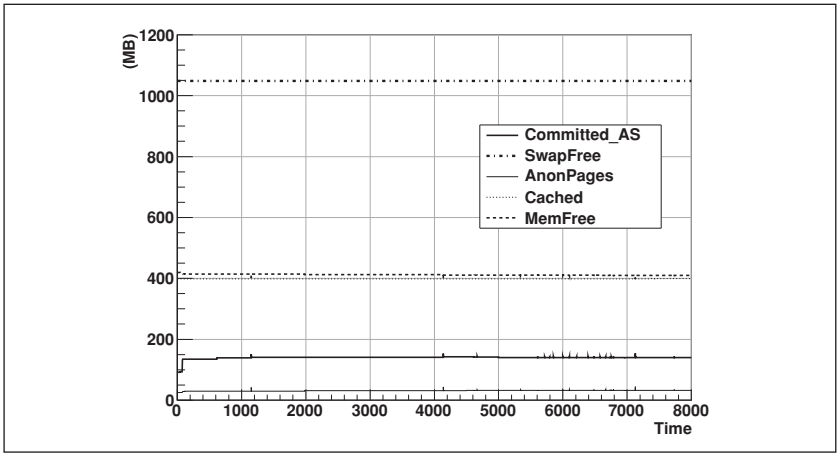

図 6-3 メモリリークをしない設定で実行させた結果

めても Committed_AS は、メモリリークにつれて増加していきますが、割り当て可能な実メモリは上限があるため、AnonPages は、あるところでほぼ一定になります。

なお、先ほどのプログラムで変数 Pleak を 0 にして（すなわち、メモリリークをしない設定にして）、実行させた結果を**図 6-3** に示します。Committed_AS の値は、メモリリークを起こしていないため、**図 6-2** と異なり、全体的には、ほぼ一定です。また、Committed_AS の増減はところどころしかありません（図中で髭のようにはみ出ている部分）。これは、

少量（おおよそ 1MB 以下）のメモリ確保では、glibc が、内部に確保している領域を返すため、プロセス全体として、OS にメモリを要求する頻度が少なく、プロセスとしてのメモリ使用量がたまにしか変化しないためです。

まとめ

/proc/meminfo に含まれるシステムのメモリに関する情報とメモリリーク時に変化する項目を説明しました。

—— Kazuhiro Yamato

HACK #55 /proc/<PID>/mem でプロセスのメモリ内容を高速に読み出す

/proc インタフェースを用いた、ptrace システムコールよりも高速なプロセスのメモリ空間へのアクセス方法を説明します。

/proc/<PID>/mem インタフェース

任意プロセスのメモリ内容を仮想ファイル /proc/‹PID›/mem を通じて、読み出すことができます。‹PID› は、読み出し対象プロセスのプロセス ID です。

同様の処理は、ptrace システムコールの PTRACE_PEEKDATA を用いても行うことができます。しかし、PTRACE_PEEKDATA を用いて一度に読み出すことのできるメモリサイズは、i386 アーキテクチャでは 4 バイト、x86_64 アーキテクチャでも 8 バイトです。そのため、読み出しサイズが大きい場合、何度も ptrace システムコールを呼び出す必要があり、処理の完了までに多くの時間が必要になります。/proc/‹PID›/mem インタフェースは、read システムコールを通じて、任意サイズのメモリを一度のシステムコールで読み出すことができるので、より短い時間で処理を完了することができます。そのため、大きなデータ領域の値を検査する場合などに有用です。

サンプルプログラム

以下のサンプルプログラムは、引数で指定されたプロセスのメモリを /proc/‹PID›/mem を用いて読み出します。第 1 引数に読み出し対象プロセスの PID、第 2 引数に読み出すメモリ領域の先頭アドレス、第 3 引数に読み出しサイズを取ります。

```
#include ‹stdio.h›
#include ‹stdlib.h›
#include ‹unistd.h›
#include ‹fcntl.h›
#include ‹errno.h›
```

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>

int main( int argc, char *argv[] )
{
    const int npath = 32;
    char path[npath]; /* /proc/<PID>/mem */
    pid_t pid;        /* 読み出しプロセスの pid */
    int target_addr;  /* 読み出し先頭アドレス */
    int read_sz;      /* 読み出しサイズ */
    int fd;
    off_t ofs;
    ssize_t sz;
    unsigned char *buf;

    /* 引数の取得 */
    if (argc < 4) {
        printf("Usage:\n");
        printf("  # %s pid target_addr(hex) read_sz(hex)\n\n", argv[0]);
        exit(1);
    }
    pid = atoi(argv[1]);
    sscanf(argv[2], "%x", &target_addr);
    sscanf(argv[3], "%x", &read_sz);

    /* 読み出しバッファの確保 */
    buf = malloc(read_sz);
    if (buf == NULL) {
        fprintf(stderr, "Failed to malloc (size: %d)\n", read_sz);
        exit(1);
    }

    /* 読み出し対象のトレース状態化と /proc/<PID>/mem のオープン */
    if (ptrace(PTRACE_ATTACH, pid, NULL, NULL) != 0) {
        fprintf(stderr, "Failed to attach (pid: %d)\n", pid);
        exit(1);
    }
    if (waitpid(pid, NULL, 0) < 0) { /* ATTACH の完了を待つ */
```

```
        fprintf(stderr, "Failed to waitpid (pid: %d)\n", pid);
        exit(1);
    }

    snprintf(path, npath, "/proc/%d/mem", pid);
    fd = open(path, O_RDONLY);
    if (fd < 0 ) {
        fprintf(stderr, "Failed to open: %s\n", path);
        ptrace(PTRACE_DETACH, pid, NULL, NULL);
        exit(1);
    }

    /* 目的のアドレスまでシークし、読み出す */
    ofs = lseek(fd, target_addr, SEEK_SET);
    if (ofs == (off_t)-1) {
        fprintf(stderr, "Failed to lseek, errno: %d\n", errno);
        ptrace(PTRACE_DETACH, pid, NULL, NULL);
        exit(1);
    }

    sz = read(fd, buf, read_sz);
    if (sz != read_sz) {
        fprintf(stderr, "Failed to read, errno: %d\n", errno);
        ptrace(PTRACE_DETACH, pid, NULL, NULL);
        exit(1);
    }

    /* メモリ内容の表示 */
    for (sz = 0; sz < read_sz; sz++) {
        if (sz%16 == 0) printf("\n");
        printf("%02x ", buf[sz]);
    }
    printf("\n");

    close(fd);
    ptrace(PTRACE_DETACH, pid, NULL, NULL);
    free(buf);
    return EXIT_SUCCESS;
}
```

上記サンプルプログラムの基本的な流れは、読み出し対象プロセスのトレース状態化、/proc/<PID>/mem のオープン、目的アドレスまでのシーク、read() による読み出しです。ptrace(PTRACE_ATTACH, ...) を用いた、読み出し対象プロセスのトレース状態化を行わなければ、その後の処理は失敗します。

ベンチマーク

/proc/<PID>/mem を使用する場合と、ptrace(PTRACE_PEEKDATA,...) を使用する場合で、どのくらいの差があるかベンチマークをしました。Core 2 6400 2.13GHz のマシンを使用し、カーネルは Linux 2.6.22.1 (i386)です。読み出し対象プロセスは、128MB のデータ領域を持ち、その領域はランダムな値が書き込まれています。両者による、データ領域の読み出し時間を表にまとめました。4KB 以下ではほとんど両者に差はありませんが、読み出しサイズが 64KB 以上の場合には、read() を用いる方が高速になっています。特に 16MB の読み出しでは、100 倍近い時間差になりました。

表 6-2 read() と ptrace() の読み出し時間

読み出しサイズ	read()	ptrace()
4KB	0.009s	0.010s
64KB	0.009s	0.023s
1MB	0.011s	0.242s
16MB	0.048s	3.675s

まとめ

/proc/<PID>/mem インタフェースを用いて高速に任意プロセスのメモリを読み出す方法を紹介しました。ベンチマークから、数十キロバイト以上の読み出しを行う場合、ptrace(PTRACE_PEEKDATA, ...) よりも、この方法が高速なことを示しました。

—— Kazuhiro Yamato

HACK #56

OOM Killer の動作と仕組み

OOM Killer の動作と仕組みについて説明します。

Linux では Out Of Memory（OOM）Killer という機能によりシステムのメモリ・スワップを使い尽くすと、メモリを確保する最終手段としてプロセスにシグナルを送信し、強制的に終了させようとします。

この機能のおかげでメモリを解放できないにもかかわらずメモリ確保の処理が繰り返されシステムが止まってしまうのを避けることができます。またメモリを過剰に消費してい

るプロセスを検出できます。本Hackでは2.6カーネルのOOM Killerについて説明します。

動作・ログの確認

システムの検証や負荷試験をしていると、動作しているはずのプロセスが終了していたり、突然 ssh のコネクションが切れ、再度ログインしようとしても接続できないときがあります。

このようなときはログを確認します。以下のようなメッセージが出力されているときがあります。

```
Pid: 4629, comm: stress Not tainted 2.6.26 #3

Call Trace:
 [<ffffffff80265a2c>] oom_kill_process+0x57/0x1dc
 [<ffffffff80238855>] __capable+0x9/0x1c
 [<ffffffff80265d39>] badness+0x16a/0x1a9
 [<ffffffff80265f59>] out_of_memory+0x1e1/0x24b
 [<ffffffff80268967>] __alloc_pages_internal+0x320/0x3c2
 [<ffffffff802726cb>] handle_mm_fault+0x225/0x708
 [<ffffffff8047514b>] do_page_fault+0x3b4/0x76f
 [<ffffffff80473259>] error_exit+0x0/0x51

Node 0 DMA per-cpu:
CPU    0: hi:    0, btch:   1 usd:   0
CPU    1: hi:    0, btch:   1 usd:   0
...
Active:250206 inactive:251609 dirty:0 writeback:0 unstable:0
 free:3397 slab:2889 mapped:1 pagetables:2544 bounce:0
Node 0 DMA free:8024kB min:20kB low:24kB high:28kB active:8kB inactive:180kB present:7448kB pa
ges_scanned:308 all_unreclaimable? yes
lowmem_reserve[]: 0 2003 2003 2003
...
Node 0 DMA: 6*4kB 4*8kB 2*16kB 2*32kB 5*64kB 1*128kB 3*256kB 1*512kB 2*1024kB 2*2048kB 0*4096k
B = 8024kB
Node 0 DMA32: 1*4kB 13*8kB 1*16kB 6*32kB 2*64kB 2*128kB 1*256kB 1*512kB 0*1024kB 0*2048kB 1*40
96kB = 5564kB
29 total pagecache pages
Swap cache: add 1630129, delete 1630129, find 2279/2761
Free swap  = 0kB
Total swap = 2048248kB
```

```
Out of memory: kill process 2875 (sshd) score 94830592 or a child
Killed process 3082 (sshd)
```

最後に Out of memory（メモリ不足）とあります。これは OOM Killer が動作したことを示します。再度接続できないときは sshd が OOM Killer で終了させられているのが原因です。sshd を起動し直さないとログインできません。

OOM Killer はプロセスを終了させることにより空きメモリを確保しますが、どのようにそのプロセスを選定するか次に説明します。

プロセスの選び方

OOM Killer はメモリが枯渇するとプロセスすべてを参照し、プロセスごとに独自のポイントを付けます。このポイントが一番高かったプロセスに対してシグナルを送信します。

ポイントの付け方

OOM Killer はさまざまなことを考慮してポイントを付けます。プロセスごとに、以下の 1 から 9 について確認しポイントを付けます。

1. まずはプロセスの仮想メモリサイズをポイントの基本とします。仮想メモリサイズは ps コマンドの VSZ や /proc/<PID>/status の VmSize [†] で確認できます。メモリを消費しているプロセスほど最初のポイントは高くなります。単位は 1KB を 1 ポイントとしています。1GB のメモリを消費しているプロセスであれば、ポイントはおよそ 1000000 となります。

2. swapoff システムコールを実行しているプロセスであれば、ポイントを最大値（unsigned long の最大値）に設定します。スワップを無効にするという行為がメモリ不足と逆行しており、速やかに OOM Killer の対象とさせるためです。

3. 親プロセスの場合はすべての子プロセスについてメモリサイズの半分をポイントに加算していきます。

4. プロセスの CPU 使用時間と起動時間からポイントを調整します。ここでは長時間起動している、また動作しているプロセスほど重要と見なしポイントを低く保つためです。
   まずは CPU 使用時間（10 秒単位）の平方根でポイントを割ります。CPU 使用時間が 90 秒であれば、10 秒単位なので 9 の平方根「3」でポイントを割ります。

† /proc/<PID>/status の VmSize とポイントの値が多少異なる場合があります。

またプロセスが起動してからの時間でもポイントを調整します。起動時間（1000秒単位）の平方根の平方根でポイントを割ります。16000秒起動し続けているプロセスの場合は、16の平方根「4」、さらにその平方根の「2」でポイントを割ります。どちらも長時間動作しているプロセスは重要と見なしています。

ソースコードのコメントには10秒単位、1000秒単位と書いてあるのですが、実際にはビット演算で8と1024で計算されます。

5. niceコマンドなどで優先度を低く設定されたプロセスはポイントを2倍にします。nice -nで1～19を設定したコマンドはポイントが2倍になります。
6. スーパーユーザプロセスは一般的に重要であることから、ポイントを1/4にします。
7. capset(3)などでケーパビリティCAP_SYS_RAWIO †を設定されているプロセスはポイントが1/4になります。H/Wに直接アクセスできるプロセスは重要であると判断しています。
8. cgroupに関して、OOM Killerが動作するきっかけとなったプロセスの許可されているメモリノードと全く別のメモリノードしか許可されていないプロセスであれば1/8にします。
9. 最後にprocファイルシステムoom_adjによりポイントを調整します。

このようなルールですべてのプロセスにポイントを付け、一番高いポイントのプロセスにシグナルSIGKILLを送ります（2.6.10まではケーパビリティCAP_SYS_RAWIOが設定されている場合はSIGTERMを送信し、設定されていない場合はSIGKILLを送信します）。

各プロセスのポイントは/proc/<PID>/oom_scoreで確認できます。

ただしinit（PIDが1の）プロセスはOOM Killerの対象にはなりません。対象となったプロセスに子プロセスがいる場合は、先にその子プロセスへシグナルが送信されます。

また対象となったプロセスにシグナルを送信したあと、システムの全スレッドを参照し、スレッドグループ（TGID）が違っていても、対象となったプロセスと同じメモリ空間を共有しているプロセスが存在する場合は、それらのプロセスにもシグナルが送られます。

OOM Killerに関するprocファイルシステム

ここからはOOM Killerに関係するprocファイルシステムについて説明をします。

† デフォルトは設定されています。

/proc/<PID>/oom_adj

/proc/<PID>/oom_adj に値を設定するとポイントを調整することができます。調整値の範囲は -16 ～ 15 です。プラスの値は OOM Killer に選ばれやすくなります。マイナスは低くなります。例えば 3 を設定するとポイントが 2^3 倍になり、-5 を指定するとポイントが $1/2^5$ になります。

"-17" は特別な値で、設定すると OOM Killer によるシグナルを禁止します（-17 の設定は 2.6.12 カーネルからできます）。

OOM Killer が動作してもリモートからログインするため sshd を対象外にしたい場合は以下のようにします。

```
# cat /proc/`cat /var/run/sshd.pid`/oom_score
15
# echo -17 >  /proc/`cat /var/run/sshd.pid`/oom_adj
# tail /proc/`cat /var/run/sshd.pid`/oom_*
==> /proc/2278/oom_adj <==
-17
==> /proc/2278/oom_score <==
0                                   /* ポイントが 0 になる */
```

/proc/<PID>/oom_adj についてのドキュメントは 2.6.18 カーネルから Documentation/filesystems/proc.txt に記述がありますが、実際は 2.6.11 カーネルから使用できます。

/proc/sys/vm/panic_on_oom

/proc/sys/vm/panic_on_oom に 1 を設定すると OOM Killer が動作したときにシグナルの送信ではなくパニックさせることができます。

```
# echo 1 > /proc/sys/vm/panic_on_oom
```

/proc/sys/vm/oom_kill_allocating_task

2.6.24 カーネルから proc ファイルシステムに oom_kill_allocating_task があります。これに 0 以外の値を設定すると、OOM Killer が動作するきっかけとなったプロセス自身がシグナルを受けます。全プロセスに対するポイント計算は省略します。

```
# echo 1 > /proc/sys/vm/oom_kill_allocating_task
```

これにより全プロセスを参照せずに済みますが、プロセスの優先度や root 権限などが考慮されず、一方的にシグナルが送られます。

/proc/sys/vm/oom_dump_tasks

2.6.25 カーネルから oom_dump_tasks を 0 以外の値に設定すると OOM Killer が動作したときの出力にプロセスの一覧情報が追加されます。

以下は設定例です。

```
# echo 1 > /proc/sys/vm/oom_dump_tasks
```

情報は以下のようなものになります。dmesg や syslog で確認できます。

```
[ pid ]   uid  tgid total_vm      rss cpu oom_adj name
[    1]     0     1     2580        1   0       0 init
[  500]     0   500     3231        0   1     -17 udevd
[ 2736]     0  2736     1470        1   0       0 syslogd
[ 2741]     0  2741      944        0   0       0 klogd
[ 2765]    81  2765     5307        0   0       0 dbus-daemon
[ 2861]     0  2861      944        0   0       0 acpid
...
[ 3320]     0  3320   525842   241215   1       0 stress
```

カーネルコンフィグ

2.4 カーネルではカーネルのコンフィグで OOM Killer の有効／無効が設定できました。

```
General setup
  [ ] Select task to kill on out of memory condition
```

2.6 カーネルからはこのコンフィグはなくなり、設定はできません。

RHEL の特徴

RHEL5 では OOM Killer をストックカーネルよりも慎重に動作させます。OOM Killer は呼び出された回数をカウントしており、ある時間内にある回数呼び出された場合にのみ動作します。

1. OOM Killer が前回に呼び出されてから次の呼び出しが 5 秒以上経過しているときは呼び出された回数をリセットします。これは突発的なメモリ負荷が生じただけでプロセスを終了させないためです。
2. カウントが 0 になってから 1 秒以内に呼び出された場合は、呼び出された回数としてカウントをしません。

3. OOM Killerの呼び出し回数が10回未満の場合は、実際には動作させません。OOM Killerが10回呼び出されたら初めてメモリが不足していると認めます。

4. 最後にOOM Killerが動作してから5秒以上経過しないと再度OOM Killerは動作しません。このため動作頻度は最大でも5秒に1回となります。不必要に連続して複数プロセスを終了させないためです。OOM Killerによりシグナルを受信したプロセスが終了する（メモリが解放される）のを待つ意味もあります。

5. OOM Killerが動作すると呼び出された回数は0にリセットします。

つまり、5秒以内にOOM Killerが呼び出される状態が10回以上続いた場合にのみ動作します。

これらの制限はもともとストックカーネル2.6.10まであったものです。そのため2.6.9がベースのRHEL4でもこれらの制限をしています。

RHEL4での動作確認

RHEL4（2.6.9カーネル）でOOM Killerの動作を確認しました。以下の例では、メモリ、スワップ領域はともに2GBの環境で、負荷テストツールstressを使って故意にメモリを消費させています。

stressはメモリ、CPU、ディスクI/Oの負荷ツールです。個別に負荷を与える、またはこれらを組み合わせて同時に負荷を与えることも可能です。stressは動作している間にシグナルを受信すると、その旨のメッセージを出力して終了します。

```
# wget -t0 -c http://weather.ou.edu/~apw/projects/stress/stress-1.0.0.tar.gz
# tar zxvf stress-1.0.0.tar.gz
# cd stress-1.0.0
# ./configure ; make ; make install
# stress --vm 2 --vm-bytes 2G --vm-keep     /* 2プロセスでメモリを2Gずつ消費 */
stress: info: [17327] dispatching hogs: 0 cpu, 0 io, 2 vm, 0 hdd
stress: FAIL: [17327] (416) <-- worker 17328 got signal 15        /* SIGTERMシグナルを受信 */
stress: WARN: [17327] (418) now reaping child worker processes
stress: FAIL: [17327] (452) failed run completed in 70s
```

以下はこのときのコンソール画面です。

```
oom-killer: gfp_mask=0xd0
Mem-info:
…
```

```
Node 0 Normal per-cpu:
cpu 0 hot: low 32, high 96, batch 16
cpu 0 cold: low 0, high 32, batch 16
cpu 1 hot: low 32, high 96, batch 16
cpu 1 cold: low 0, high 32, batch 16
...
Free pages:       13144kB (0kB HighMem)          /* reserve pages があるため 0 にはならない */
Active:251180 inactive:249985 dirty:0 writeback:0 unstable:0 free:3286 slab:2731 mapped:500625
pagetables:2245
...
Node 0 Normal free:1424kB min:1428kB low:2856kB high:4284kB active:1004592kB inactive:999940kB
present:2080512kB
pages_scanned:2384217 all_unreclaimable? yes
protections[]: 0 0 0
...
Node 0 DMA: 4*4kB 5*8kB 1*16kB 4*32kB 2*64kB 3*128kB 1*256kB 1*512kB 0*1024kB 1*2048kB 2*4096kB =
11720kB
Node 0 Normal: 0*4kB 0*8kB 1*16kB 2*32kB 1*64kB 0*128kB 1*256kB 0*512kB 1*1024kB 0*2048kB 0*4096kB =
1424kB
...
Swap cache: add 524452, delete 524200, find 60/102, race 0+0
Free swap:            0kB                      /* スワップの残りは 0 */
524224 pages of RAM                            /* 1 ページ 4k なので、メモリサイズは 2G */
10227 reserved pages                           /* カーネル内部で予約しているメモリ */
19212 pages shared
253 pages swap cached
Out of Memory: Killed process 17328 (stress).  /* シグナルで終了したプロセス */
```

ストックカーネルでは OOM Killer を無効にすることはできませんが、RHEL4 では /proc/sys/vm/oom-kill が用意されており無効にすることができます。

```
# echo 0 > /proc/sys/vm/oom-kill
または
# /sbin/sysctl -w vm.oom-kill=0
```

無効にすると OOM Killer はシグナルを送信しません。ただし上のメモリ情報のメッセージは出力されます。

RHEL5 での動作確認

RHEL5（2.6.18 カーネル）で OOM Killer の動作を確認しました。確認方法は RHEL4

と同じです。

```
# stress --vm 2 --vm-bytes 2G --vm-keep
stress: info: [11779] dispatching hogs: 0 cpu, 0 io, 2 vm, 0 hdd
stress: FAIL: [11779] (416) <-- worker 11780 got signal 9        /* SIGKILL */
stress: WARN: [11779] (418) now reaping child worker processes
stress: FAIL: [11779] (452) failed run completed in 46s
```

以下はこのときのコンソール画面です。バックトレースが追加されており、デバッグに役立ちます。

```
Call Trace:
 [<ffffffff800bf551>] out_of_memory+0x8e/0x321
 [<ffffffff8000f08c>] __alloc_pages+0x22b/0x2b4
...
 [<ffffffff800087fd>] __handle_mm_fault+0x208/0xe04
 [<ffffffff80065a6a>] do_page_fault+0x4b8/0x81d
 [<ffffffff800894ad>] default_wake_function+0x0/0xe
 [<ffffffff80039dda>] tty_ldisc_deref+0x68/0x7b
 [<ffffffff8005cde9>] error_exit+0x0/0x84

Mem-info:
...
Swap cache: add 512503, delete 512504, find 90/129, race 0+0
Free swap  = 0kB
Total swap = 2048276kB
Free swap:            0kB
524224 pages of RAM
42102 reserved pages
78 pages shared
0 pages swap cached
Out of memory: Killed process 11780 (stress).
```

RHEL5 では /proc/sys/vm/oom-kill はなくなっています。

まとめ

本 Hack では OOM Killer の仕組みと各種設定について説明しました。システムの動作がおかしくなったときには syslog など確認し OOM Killer の出力があれば、メモリ不足になっていたことがわかります。

参考文献

- stress

  http://weather.ou.edu/~apw/projects/stress/

—— Naohiro Ooiwa

HACK #57 フォルト・インジェクション

Linux カーネルのオプションとして提供されているフォルト・インジェクション（fault injection）について解説します。

フォルト・インジェクションとは、ソフトウェアのテストの際に失敗を発生させることで、エラー処理コードのようにあまり実行されないコードをテストする方法です。ソフトウェアの堅牢性を高めるのに役立ちます。また、エラー処理が正しく行われていれば、一目で原因を突き止めることができたような問題に悩まされることも防げたりするので、導入してみる価値のあるテクニックだと思います。

フォルト・インジェクションのテクニックを利用したものに failmalloc というライブラリがあります（http://www.nongnu.org/failmalloc/）。このライブラリをリンクすると malloc などのメモリ割り当て関数にフックを挿入して、意図的にメモリ割り当てを失敗させることができます。

通常のプログラムでは、メモリ割り当て関数の実行直後に、メモリ割り当てが失敗した場合のためのエラー処理コードがあり、プログラムを終了するか、あるいは、呼び出し元に通知するためにエラーコードを返すといった処理が行われます。そして、さらにその関数の呼び出し元の直後にも、同様のエラー処理コードがあるという構造が、関数の呼び出し元をさかのぼるごとに、繰り返し行われています。

つまり、メモリ割り当て関数の直後のエラー処理コードだけではなく、プログラム内のさまざまな箇所のエラー処理コードがテストできる可能性があります。

Linux カーネル フォルト・インジェクション

failmalloc に影響を受けて Linux カーネルの中で同じような仕組みを実装したものが、ここで説明する Linux カーネルフォルト・インジェクションです。さまざまなフォルト・インジェクションを簡単に実装できるフレームワークと、以下に挙げる実用的なフォルト・インジェクションが実装されています。

表6-3 フォルト・インジェクションの種類

種類	説明
fail_page_alloc	ページアロケータのメモリ割り当ての失敗
failslab	スラブアロケータのメモリ割り当ての失敗
fail_make_request	ディスクI/O要求の失敗
fail_io_timeout	ディスクI/Oタイムアウト

バージョン2.6.21以降のカーネルで利用できます。ただしfail_io_timeoutは、バージョン2.6.28以降のカーネルで利用できます。

failslab

ここではスラブアロケータのメモリ割り当てのフォルト・インジェクションの実装であるfailslabについて説明します。

スラブアロケータ

Linuxカーネルの内部で行われるメモリ割り当ては、用途に応じてさまざまなものが利用されますが、スラブアロケータは、その中でも一番よく使われているものです（『詳解Linuxカーネル 第3版』などをご参照ください）。

ここでは、できるだけいろいろな箇所のエラー処理コードをテストして、たくさんのバグを見つけ出すことが目的なので、スラブアロケータを失敗させるのが、もっとも適しています。

スラブアロケータのメモリ割り当て関数として代表的なkmalloc()関数を例に挙げて解説します。

```
void *kmalloc(size_t size, gfp_t flags)
```

メモリ割り当てが成功した場合は、割り当てられたメモリアドレスを返し、失敗した場合はNULLポインタを返します。

size引数は、割り当てるメモリサイズをバイト単位で指定します。gfp_mask引数は、複数のGFPフラグを論理和で指定します。GFPフラグとは、割り当てるメモリの属性や空きメモリを見つけるときの振る舞いなどを指定するフラグです。

failslabを有効にすると、実際にスラブアロケータによるメモリ割り当てが可能な場合でも、指定した条件で失敗させることができます（kmalloc()関数の場合、NULLポインタを返す）。

failslab を有効にする

failslab を利用するためには、以下の 4 つのオプションを有効にしたカーネルを利用する必要があります。

- CONFIG_SLAB または CONFIG_SLUB
- CONFIG_FAULT_INJECTION
- CONFIG_FAILSLAB
- CONFIG_FAULT_INJECTION_DEBUG_FS

make menuconfig でコンフィグレーションをする場合は、以下の手順で上記の 4 つのオプションを有効にすることができます。

1. トップメニューの General setup で Choose SLAB allocator の選択肢から SLAB または SLUB を選択します。
   スラブアロケータの実装には 3 種類のもの（SLAB, SLUB, SLOB）が選択可能ですが、failslab が利用できるのは SLAB と SLUB のみです（ただしバージョン 2.6.29 以前のカーネルでは SLAB のみです）。
2. トップメニューの Kernel hacking の中から Fault-injection framework を選択すると、以下のオプションが表示されるので両方とも選択します。
   - Fault-injection capability for kmalloc()
   - Debugfs entries for fault-injection capabilities

設定パラメータ

フォルト・インジェクションを発生させる条件は debugfs のマウントディレクトリに現れる failslab ディレクトリ配下にあるファイルを使って設定します。

failslab ディレクトリにあるファイルは以下のとおりです（以降、実行例では、debugfs のマウントディレクトリを /debugfs とします）。

```
$ ls /debugfs/fail_page_alloc
ignore-gfp-wait  interval  probability  space  task-filter  times  verbose
```

各ファイルに値を書き込むことによって設定を行い、読み出すことで、現在設定されている値を知ることができます。

probability:

フォルト・インジェクションを発生させる割合をパーセンテージで指定します（初期値は 0 です）。

例えば probability を 1 に指定すると、スラブアロケータのメモリ割り当て関数の呼び出しが、1% の確率でランダムに失敗します。0 の場合はフォルト・インジェクションによる失敗は発生しません。

interval:

フォルト・インジェクションが一度発生したあと、ここで指定した回数は発生しません（初期値 1）。

例えば interval を 100 に指定すると、スラブアロケータのメモリ割り当てが失敗した直後から 100 回のメモリ割り当ては失敗しません。ただし、フォルト・インジェクションではなく、本当のメモリ不足によるメモリ割り当ての失敗を妨げることは当然できません。

probability の値が 1% よりも少ない割合で失敗させたいという場合には、probability を 1 以上にして interval に大きな値を指定します。

times:

フォルト・インジェクションを発生させる回数の上限を指定します。

例えば times を 10 に指定すると、メモリ割り当ては 10 回までしか失敗しません。-1 を指定すると何回でも発生させ続けることができます（初期値 1）。

space:

フォルト・インジェクションが発生するようになるまでのメモリ割り当ての総サイズをバイト単位で指定します。

例えば space を 41943040（= 40MB）に指定すると、その直後からスラブアロケータによるメモリ割り当ての総数が 40MB に達するまでの間、フォルト・インジェクションによるスラブアロケータのメモリ割り当てが失敗することはありません（初期値 0）。

verbose:

フォルト・インジェクションが発生したときのカーネルメッセージの冗長度を指定します（初期値 2）。

1 を指定した場合は、フォルト・インジェクションによる失敗が発生する度に、カーネルログに以下のメッセージが出力されます。

FAULT_INJECTION: forcing a failure

2 を指定した場合は、上記のメッセージに加えてそのときのコールトレースも表示されます。

```
$ dmesg
...
FAULT_INJECTION: forcing a failure
Pid: 2237, comm: rsyslogd Not tainted 2.6.28-rc9 #9
Call Trace:
         [<ffffffff811557eb>] should_fail+0xc5/0x101
 [<ffffffff810ae093>] should_failslub+0x2b/0x34
 [<ffffffff810aebd6>] kmem_cache_alloc+0x20/0xb0
 [<ffffffff81084683>] mempool_alloc_slab+0x11/0x13
 [<ffffffff8108478f>] mempool_alloc+0x4a/0x106
 [<ffffffff81084683>] ? mempool_alloc_slab+0x11/0x13
 [<ffffffff8108478f>] ? mempool_alloc+0x4a/0x106
 [<ffffffff810d5966>] bvec_alloc_bs+0x90/0xd7
 [<ffffffff810d5a21>] bio_alloc_bioset+0x74/0xca
 [<ffffffff810d5ae1>] bio_alloc+0x10/0x1f
 [<ffffffff810d1805>] submit_bh+0x68/0x109
 [<ffffffff810d35f6>] __block_write_full_page+0x1d8/0x2da
 [<ffffffffa004ad91>] ? ext3_get_block+0x0/0xfc [ext3]
 [<ffffffff810d37ca>] block_write_full_page+0xd2/0xd7
 [<ffffffffa004c581>] ext3_ordered_writepage+0xd1/0x17b [ext3]
```

0 の場合は、何も表示されません。

task-filter:

特定のプロセスのみフォルト・インジェクションを発生させる仕組みを有効にするかどうかを指定します。Y の場合は有効で、N の場合は無効です（初期値 N）。有効にした場合は、対象とするプロセスのプロセス ID を ‹PID› とすると、/proc/‹PID›/make-it-fail に 1 を書き込みます。

すると、そのプロセスのコンテキストからのスラブアロケータのメモリ割り当てのみ失敗します。

それ以外のプロセスのコンテキストや、割り込みコンテキストからのスラブアロケーションは失敗しません。この属性は fork によって生成された子プロセスにも継承されるため、次のようなスクリプトを使ってコマンドを実行することによって、そのコマンドからのスラブアロケータのみ失敗させることができます。

failcmd スクリプト

```
#!/bin/sh

echo 1 > /proc/self/make-it-fail
exec $@
```

[実行例]

```
$ sh failcmd <command> <args...>
```

ignore-gfp-wait:

スラブアロケーションの際に指定される GFP マスクに __GFP_WAIT フラグが含まれている場合にフォルト・インジェクションを発生させるかどうかを指定します。Y の場合はフォルト・インジェクションを発生させません。N の場合は発生させます（初期値 Y）。この設定を無効にする場合は、通常は task-filter も同時に設定して組み合わせて利用することになるでしょう。詳細は**「フォルト・インジェクションを利用した Linux カーネルの潜在的なバグの発見 2」[HACK #58]** を参照してください。

まとめ

Linux カーネルのオプションとして提供されているフォルト・インジェクション（fault injection）について解説しました。

参考文献

- Wikipedia: Fault injection
  http://en.wikipedia.org/wiki/Fault_injection
- Failmalloc
  http://www.nongnu.org/failmalloc/
- カーネル付属文書 fault-injection.txt
  Documentation/fault-injection/fault-injection.txt

—— Akinobu Mita

HACK #58 フォルト・インジェクションを利用したLinuxカーネルの潜在的なバグの発見

フォルト・インジェクションのテクニックを利用したLinuxカーネルの潜在的なバグの発見手順をfailslabを例にとって説明します。

フォルト・インジェクションを発生させてみる

それでは、実際にフォルト･インジェクションを発生させてみます。試してみる場合は、カーネルパニックやファイルシステムの破壊を引き起こしてしまう可能性もあるためテスト用の環境で行ってください。

フォルト・インジェクションが発生する回数を10回までに制限します。これは、設定ミスなどで無制限にフォルト・インジェクションが発生し続けて操作不能に陥ってしまうことをあらかじめ防止するためです。

```
# echo 10 > /debugfs/failslab/times
```

フォルト・インジェクションの発生確率を1%にします。この直後からフォルト・インジェクションが発生するようになります。

```
# echo 1 > /debugfs/failslab/probability
```

フォルト・インジェクションはストレステストと組み合わせて行われることが多く、ここでは簡単なコマンドでシステムに負荷を与えてみます。

```
# dd if=/dev/zero of=/tmp/junk
^C
```

実際にフォルト・インジェクションが発生したかどうかは、verboseが1以上に設定してあれば、カーネルのログメッセージをdmesgコマンドで見ればわかります。ここでは、timesを10に制限しているのでtimesを表示させれば正確な発生回数がわかります。

```
# cat /debugfs/failslab/times
0
```

10から0になっているので10回のフォルト・インジェクションが発生しました。以降再度timesを設定し直すまでフォルト・インジェクションは発生しません。

滅多に失敗しないようなスラブアロケーションも失敗させる

先ほどの例ではignore-gfp-waitが有効（デフォルト）となっていましたが、次の例では

ignore-gfp-wait を無効にしてみます。このようにすると __GFP_WAIT フラグの指定されたスラブアロケーションも失敗するようになります。

__GFP_WAIT フラグが指定されたスラブアロケーションは、空きメモリを見つけるためにコンテキストをスリープすることができるため、実際の動作上はほとんど失敗することがありません。つまり、今回の例では滅多に実行されないようなエラー処理も実行されるようになり、通常は再現できないようなカーネルバグを引き起こすことができる可能性が高まります。

その代わり、無差別にユーザアプリケーションのシステムコールを失敗させてしまい終了してしまう可能性があるため、ignore-gfp-wait を無効にする場合は、task-filter も合わせて有効にして、対象のプロセス以外に影響を与えないようにします。

いったん発生確率を 0% にしてフォルト・インジェクションが発生しないようにします。

```
# echo 0 > /debugfs/failslab/probability
```

ignore-gfp-wait を無効にして task-filter を有効にします。

```
# echo N > /debugfs/failslab/ignore-gfp-wait
# echo Y > /debugfs/failslab/task-filter
```

先ほどの例と同様に、フォルト・インジェクションが発生する回数を 10 回までに制限し、発生確率を 1% にします。

```
# echo 10 > /debugfs/failslab/times
# echo 1 > /debugfs/failslab/probability
```

今回は task-filter を有効にしているので明示的に指定したコマンドでなければフォルト・インジェクションは発生しません。

先ほどの例と同じ dd コマンドでテストするためには failcmd スクリプトの引数にコマンドを指定するだけです（**[HACK #57]** で紹介した failcmd スクリプトを PATH の通ったディレクトリに配置し実行権限をつけておいてください）。

```
# failcmd dd if=/dev/zero of=/tmp/junk
dd: writing to `/tmp/junk': メモリを確保できません
105+0 records in
104+0 records out
53248 bytes (53 kB) copied, 0.00176169 s, 30.2 MB/s
```

今回はddコマンドの中でのwriteシステムコールがフォルト・インジェクションによりメモリ確保に失敗してddコマンドが終了してしまったようです。

カーネルOops発生

上記のような方法でフォルト・インジェクションとさまざまなストレステストを組み合わせて実行することができます。以下のカーネルログは、実際にフォルト・インジェクションとLTP（http://ltp.sourceforge.net/）を組み合わせて実行したときにカーネルOopsが発生したときのものです。

```
$ dmesg
...
FAULT_INJECTION: forcing a failure
Pid: 8187, comm: mincore01 Not tainted 2.6.28-rc9 #6
Call Trace:
 [<ffffffff811537cb>] should_fail+0xc5/0x101
 [<ffffffff810ac4ff>] should_failslab+0x36/0x3f
 [<ffffffff810ad0ef>] kmem_cache_alloc+0x18/0xfe
 [<ffffffffa063137c>] ext4_mb_free_metadata+0x6b/0x336 [ext4]
 [<ffffffffa06319b4>] ext4_mb_free_blocks+0x36d/0x5dd [ext4]
 [<ffffffffa0615475>] ext4_free_blocks+0x7b/0xcf [ext4]
 [<ffffffffa061b4e7>] ext4_clear_blocks+0xe8/0xf4 [ext4]
 [<ffffffffa061b5a3>] ext4_free_data+0xb0/0x103 [ext4]
 [<ffffffffa061b91f>] ext4_truncate+0x175/0x4d4 [ext4]
 [<ffffffffa062c9db>] ? __ext4_journal_dirty_metadata+0x1f/0x48 [ext4]
 [<ffffffffa0618cc7>] ? ext4_mark_iloc_dirty+0x454/0x4da [ext4]
 [<ffffffffa06193f7>] ? ext4_mark_inode_dirty+0x181/0x196 [ext4]
 [<ffffffffa061dd21>] ext4_delete_inode+0x109/0x1cc [ext4]
 [<ffffffffa061dc18>] ? ext4_delete_inode+0x0/0x1cc [ext4]
 [<ffffffff810c3d8d>] generic_delete_inode+0xc7/0x147
 [<ffffffff810c3e22>] generic_drop_inode+0x15/0x171
 [<ffffffff810c34fd>] iput+0x61/0x65
 [<ffffffff810bccc5>] do_unlinkat+0xfc/0x173
 [<ffffffff81075831>] ? audit_syscall_entry+0x141/0x17c
 [<ffffffff810bcd4d>] sys_unlink+0x11/0x13
 [<ffffffff8100bfaa>] system_call_fastpath+0x16/0x1b
BUG: unable to handle kernel NULL pointer dereference at 0000000000000030
IP: [<ffffffffa063137c>] ext4_mb_free_metadata+0x6b/0x336 [ext4]
PGD 3e8a5067 PUD 33083067 PMD 0
Oops: 0002 [#1] SMP
```

```
last sysfs file: /sys/devices/pci0000:00/0000:00:1e.0/0000:0a:0c.0/local_cpus
CPU 1
Modules linked in: ext4 jbd2 crc16 bridge stp bnep rfcomm l2cap
...
ata_piix ata_generic libata sd_mod scsi_mod ext3 jbd mbcache uhci_hcd
ohci_hcd ehci
_hcd [last unloaded: freq_table]
Pid: 8187, comm: mincore01 Not tainted 2.6.28-rc9 #6
RIP: 0010:[<ffffffffa063137c>]  [<ffffffffa063137c>]
ext4_mb_free_metadata+0x6b/0x336 [ext4]
RSP: 0018:ffff8800099a5b08  EFLAGS: 00010202
RAX: 0000000000000000 RBX: ffff8800099a5bb8 RCX: 0000000000018280
RDX: 0000000000018280 RSI: ffffffff8143f7f0 RDI: 00007fffa27f13f8
RBP: ffff8800099a5b58 R08: 000000000001827f R09: 0000000000000000
R10: 0000000000000010 R11: 00000000ffffffff R12: ffff88003ecb5cc8
R13: ffff88002a8b8000 R14: 0000000000000000 R15: 00000000000106e
FS:  00007fcc9a7c66f0(0000) GS:ffff88003f9cd240(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
CR2: 0000000000000030 CR3: 000000003d847000 CR4: 00000000000026e0
DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 0000000000000400
Process mincore01 (pid: 8187, threadinfo ffff8800099a4000, task
ffff88003d00c240)
Stack:
 0000000232482540 ffff880032482540 ffff88003ecb5cc0 ffff88002a8b8000
 ffff880009990000 0000000000000002 ffff88000a89e000 ffff88002a8b8000
 0000000000000002 000000000000106f ffff8800099a5c48 ffffffffa06319b4
Call Trace:
 [<ffffffffa06319b4>] ext4_mb_free_blocks+0x36d/0x5dd [ext4]
 [<ffffffffa0615475>] ext4_free_blocks+0x7b/0xcf [ext4]
 [<ffffffffa061b4e7>] ext4_clear_blocks+0xe8/0xf4 [ext4]
 [<ffffffffa061b5a3>] ext4_free_data+0xb0/0x103 [ext4]
 [<ffffffffa061b91f>] ext4_truncate+0x175/0x4d4 [ext4]
 [<ffffffffa062c9db>] ? __ext4_journal_dirty_metadata+0x1f/0x48 [ext4]
 [<ffffffffa0618cc7>] ? ext4_mark_iloc_dirty+0x454/0x4da [ext4]
 [<ffffffffa06193f7>] ? ext4_mark_inode_dirty+0x181/0x196 [ext4]
 [<ffffffffa061dd21>] ext4_delete_inode+0x109/0x1cc [ext4]
 [<ffffffffa061dc18>] ? ext4_delete_inode+0x0/0x1cc [ext4]
 [<ffffffff810c3d8d>] generic_delete_inode+0xc7/0x147
 [<ffffffff810c3e22>] generic_drop_inode+0x15/0x171
```

```
 [<ffffffff810c34fd>] iput+0x61/0x65
 [<ffffffff810bccc5>] do_unlinkat+0xfc/0x173
 [<ffffffff81075831>] ? audit_syscall_entry+0x141/0x17c
 [<ffffffff810bcd4d>] sys_unlink+0x11/0x13
 [<ffffffff8100bfaa>] system_call_fastpath+0x16/0x1b
Code: 08 48 89 45 d0 48 83 7e 10 00 75 04 0f 0b eb fe 48 83 3e 00 75
04 0f 0b eb fe 48 8b 3d 36 8c 01 00 be 50 00 00 00 e8 5b bd a7 e0 <44>
89 78 30 4c 89 70 28 49 89 c5 8b 55 b4 89 50 34 48 8b 55 b8
RIP  [<ffffffffa063137c>] ext4_mb_free_metadata+0x6b/0x336 [ext4]
 RSP <ffff88000099a5b08>
CR2: 0000000000000030
---[ end trace e8fc382609867b05 ]---
```

このカーネルログから次の2つのことが読み取れます。

1. "FAULT_INJECTION: forcing a failure" という出力のあとのコールトレースから ext4_mb_free_metadata() 関数内のスラブアロケーション kmem_cache_alloc() 関数がフォルト・インジェクションにより失敗したことがわかります。

2 "BUG: unable to handle kernel NULL pointer dereference at 0000000000000030" の出力のあとのコールトレースからは ext4_mb_free_metadata() 関数内で NULL ポインタ参照により Oops が発生したことがわかります。

つまり ext4_mb_free_metadata() 関数において、スラブアロケーションのエラー処理にバグがあるため Oops が発生したことが推測できます。

実際に ext4_mb_free_metadata() 関数のソースコードを見ると原因は一目瞭然でした。

[fs/ext4/mballoc.c]

```
static noinline_for_stack int
ext4_mb_free_metadata(handle_t *handle, struct ext4_buddy *e4b,
                          ext4_group_t group, ext4_grpblk_t block, int count)
{
        struct ext4_group_info *db = e4b->bd_info;
        struct super_block *sb = e4b->bd_sb;
        struct ext4_sb_info *sbi = EXT4_SB(sb);
        struct ext4_free_data *entry, *new_entry;
        struct rb_node **n = &db->bb_free_root.rb_node, *node;
        struct rb_node *parent = NULL, *new_node;
```

```
        BUG_ON(e4b->bd_bitmap_page == NULL);
        BUG_ON(e4b->bd_buddy_page == NULL);

        new_entry  = kmem_cache_alloc(ext4_free_ext_cachep, GFP_NOFS);
        new_entry->start_blk = block;
        new_entry->group  = group;
        new_entry->count = count;
        new_entry->t_tid = handle->h_transaction->t_tid;
        new_node = &new_entry->node;
…
```

このように kmem_cache_alloc() 関数によるスラブアロケーションのエラーチェックが抜けていることが原因でした。

linux-ext4 メーリングリストに "[PATCH] ext4: fix unhandled ext4_free_data allocation failure" という件名でパッチを送りましたが、執筆時点ではまだ修正方法が確定していません。

まとめ

フォルト・インジェクションのテクニックを利用した Linux カーネルの潜在的なバグの発見手順を failslab を例にとって説明しました。

参考文献

- Linux Test Project
  http://ltp.sourceforge.net

—— Akinobu Mita

HACK #59

Linux カーネルの init セクション

カーネルのセクション、特に init セクションを意識した問題解析について説明します。

問題概要

LKML（Linux Kernel mailing list）にパニック発生の報告がありました。開発中のカーネルにアップデートをしたところ、ブート中にパニックしてしまうという内容です。下記が LKML に報告された、パニック発生時のカーネルメッセージです。

```
calling  tcp_congestion_default+0x0/0x12 @ 1
initcall tcp_congestion_default+0x0/0x12 returned 0 after 2 usecs
Freeing unused kernel memory: 448k freed
```

```
Write protecting the kernel read-only data: 4816k
int3: 0000 [#1] SMP
last sysfs file:
CPU 2
Modules linked in:
Pid: 0, comm: events/0 Not tainted 2.6.27-next-20081023 #1
RIP: 0010:[<ffffffff8078ba2b>]  [<ffffffff8078ba2b>] nmi_cpu_busy+0x1/0x15
RSP: 0018:ffff88017faa7f80  EFLAGS: 00000086
RAX: 00000000ffffffff RBX: ffff88027f60e000 RCX: ffff88017fa98000
RDX: ffffffff807eb480 RSI: 0000000000000000 RDI: ffffffff807b9e5c
RBP: ffff88017faa7f98 R08: 0000000000000000 R09: ffff88002802c768
R10: 0000000000000000 R11: ffff88027e023e90 R12: 0000000000000002
R13: 0000000000000000 R14: 0000000000000000 R15: 0000000000000000
FS:  0000000000000000(0000) GS:ffff88017fa32280(0000) knlGS:0000000000000000
CS:  0010 DS: 0018 ES: 0018 CR0: 000000008005003b
CR2: 0000000000000000 CR3: 0000000000201000 CR4: 00000000000006e0
DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 0000000000000400
Process events/0 (pid: 0, threadinfo ffff88017fa8c000, task ffff88017fa98000)
Stack:
 ffffffff80257afe ffffffff8076d938 0000000000000000 ffff88017faa7fa8
 ffffffff8021f1b0 ffff88017fa8de50 ffffffff8020cabb ffff88017fa8de50 <EOI>
 ffff88017fa8ded8 ffff88027e023e90 0000000000000000 ffff88002802c768
Call Trace:
 <IRQ> <0> [<ffffffff80257afe>] ?
generic_smp_call_function_interrupt+0x35/0xd7
 [<ffffffff8021f1b0>] smp_call_function_interrupt+0x1f/0x2f
 [<ffffffff8020cabb>] call_function_interrupt+0x6b/0x70
 <EOI> <0> [<ffffffff80212659>] ? default_idle+0x2b/0x40
 [<ffffffff8021287d>] ? c1e_idle+0xe5/0xec
 [<ffffffff8057072f>] ? atomic_notifier_call_chain+0xf/0x11
 [<ffffffff8020ad1d>] ? cpu_idle+0x48/0x66
 [<ffffffff80568784>] ? start_secondary+0x177/0x17c
Code: cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc <cc> cc cc cc cc cc
cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
RIP  [<ffffffff8078ba2b>] nmi_cpu_busy+0x1/0x15
 RSP <ffff88017faa7f80>
Kernel panic - not syncing: Fatal exception in interrupt
```

リグレッション

一般的に、特定のバージョン、特に開発中のカーネルにアップデートした際に今までうまく動作していたものが、動作しなくなったりすることがあります。これはリグレッション（regression）と呼ばれます。問題なく動作するカーネルのバージョンがはっきりしており、再現が容易なリグレッションである場合、何も考えずgit-bisectすることが近道である可能性があります。しかし、git-bisectは簡単ですが、手間暇のかかる作業です。

ここでは、別の方向から解析を試みます。

ログの詳細確認

下記メッセージからわかるとおり、int3命令が発行されています。

```
int3: 0000 [#1] SMP
```

int3命令はデバッグ割り込みであるINT3を発生させるデバッグ用の命令で、通常の処理にこの命令が埋め込まれることはありません。

次に命令ポインタRIPを確認します。

```
RIP: 0010:[<ffffffff8078ba2b>]  [<ffffffff8078ba2b>] nmi_cpu_busy+0x1/0x15
```

このint3命令は関数nmi_cpu_busy()において発行されていることがわかります。

なぜ、関数nmi_cpu_busy()関数にint3命令があるのでしょうか。問題の関数nmi_cpu_busy()を確認すると下記のように宣言されています。

```
static __init void nmi_cpu_busy(void *data)
```

ここで、注目する点は__initというキーワードです。このキーワードによって、この関数はinitセクションに配置されます。

initセクション

Linuxカーネルは ELFのセクションを駆使して構成されています。ここでは、initセクションについて説明します。Linuxカーネルにおいて、初期化処理に使われるコードやデータはinitセクションに集められます。これらの初期化用のコード・データはいったんカーネルが起動し、初期化処理が完了した時点で不要となります。そのため、初期化処理完了後にinitセクションのメモリを解放し、フリーなメモリとして再利用します。カーネルコンフィグによりますが、数百キロバイトのメモリとなります。

問題の原因から調査範囲を絞り込む

再度、問題のログを確認します。ログの頭に init セクションの解放が行われたことを示すメッセージが存在します。

```
Freeing unused kernel memory: 448k freed
```

つまり、問題となった nmi_cpu_busy() は初期化処理完了後にコールされています。しかし、nmi_cpu_busy() のコードは __init で修飾されているため、init セクションに存在します。

したがって、今回の問題は解放済みの init セクションのコードが実行されたことが原因と考えられます。

なお、x86 などの一部のアーキテクチャでは、解放済みメモリの使用というバグを見つけやすくするために、解放したメモリは特殊なバイト列で埋められます。POISON_FREE_INITMEM という名前で定義されており、内容は 0xcc です。

nmi_cpu_busy() の呼び出し元を調べると、SMP におけるプロセッサ間関数コールにて、呼び出されることが判明しました。

```
smp_call_function(nmi_cpu_busy, (void *)&endflag, 0);
```

そこで、SMP のプロセッサ間関数コールに関する修正を確認したところ、kernel/smp.c へのパッチにて、本問題が引き起こされていることがわかりました。問題のパッチにて、SMP のプロセッサ間関数コールの判定文でゴミ値をチェックするようになってしまい、目的の関数 nmi_cpu_busy() のコールが遅延したことが原因でした。問題点をパッチ作成者に報告してこの問題は対処されました。

まとめ

本 Hack では、init セクションに関係する問題を説明しました。

通常、リグレッションに対しては git-bisect を行うのですが、本問題では、不正な init セクション上の関数 nmi_cpu_busy() の呼び出しがあったことと、その関数コールがどこで行われているかを調査することで単純に git-bisect するよりも早く問題を引き起こしたパッチにたどり着けることを説明しました。

参考

- Intel® 64 and IA-32Architectures Software Developer's Manuals
  http://www.intel.com/products/processor/manuals/index.htm

- Linux Kernel Mailing List (LKML)
  http://lkml.org/lkml/2008/10/23/322

—— Hiroshi Shimamoto

HACK #60 性能の問題を解決する
oprofile を利用した性能調査とチューニング

アプリケーションのプログラムの性能が期待するものより低い場合の性能調査とチューニングについて述べます。

ここではLinux環境で定番となっているoprofileを利用した性能調査方法について記します。

oprofileの利用方法、初期化から計測まで

oprofileを利用した性能調査とチューニングのプロセスはおおむね下記のようになります。

① oprofileの初期化
② 計測するイベントの設定
③ oprofileデーモンの起動
④ アプリケーションプログラムの計測
⑤ 結果の分析と対応

それぞれを細かく見ていきましょう。

① oprofileの初期化

```
$ sudo opcontrol --init
```

② 計測するイベントの設定

計測するイベントを設定します。デフォルトの設定はハードウェアアーキテクチャによって異なります。Intelアーキテクチャの場合、Intelのマニュアルを参照してください。最新版は英語のマニュルしかありませんが、しっかり読むことによって、エンジニアとしての基礎体力がつくと思います。

測定できるイベントは下記で表示ができます。それぞれのイベント名の意味はマニュアルを参考にしてください。

```
$ sudo opcontrol --list-events
oprofile: available events for CPU type "P4 / Xeon with 2 hyper-threads"

See Intel Architecture Developer's Manual Volume 3, Appendix A and
Intel Architecture Optimization Reference Manual (730795-001)

GLOBAL_POWER_EVENTS: (counter: 0)
        time during which processor is not stopped (min count: 6000)
        Unit masks (default 0x1)
        ----------
        0x01: mandatory
BRANCH_RETIRED: (counter: 3)
        retired branches (min count: 6000)
        Unit masks (default 0xc)
        ----------
        0x01: branch not-taken predicted
        0x02: branch not-taken mispredicted
        0x04: branch taken predicted
        0x08: branch taken mispredicted
…
```

③ oprofile デーモンの起動

デーモンを起動するだけで、まだ計測は開始しません。

```
$ sudo opcontrol --start-daemon
Using 2.6+ OProfile kernel interface.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
```

前回計測したデータがあるのならば、消去しておきましょう。

```
$ sudo opcontrol --reset
Signalling daemon... done
```

④ アプリケーションプログラムの計測

oprofile の計測開始（opcontrol --start）、アプリケーションプログラムの実行、oprofile の停止（opcontrol --stop）という流れになります。

```
$ sudo opcontrol --start --no-vmlinux
Profiler running.
$ time make test-all
```

```
$ sudo opcontrol --stop
Stopping profiling.
```

結果の分析と対応

oprofileで計測したデータを表示して原因を調査してみます。

```
$ sudo opreport -l|head -10
warning: /no-vmlinux could not be found.
CPU: P4 / Xeon, speed 2400 MHz (estimated)
Counted BSQ_CACHE_REFERENCE events (cache references seen by the bus unit) with a unit mask of 0x100 (read
2nd level cache miss) count 3000
samples  %        image name               app name                 symbol name
67373    34.8114  no-vmlinux               no-vmlinux               (no symbols)
22364    11.5554  ruby                     ruby                     gc_mark
15421     7.9680  ruby                     ruby                     garbage_collect
10015     5.1747  ruby                     ruby                     st_foreach
10004     5.1690  ruby                     ruby                     gc_mark_children
9953      5.1427  libc-2.8.90.so           libc-2.8.90.so           free
8457      4.3697  ruby                     ruby                     iseq_mark
```

この例では、rubyに添付されているテストを実行してみました（make test-all）。L2キャッシュミス（BSQ_CACHE_REFERENCEというイベント名）を計測してみました。

上記のレポート（opreport -l）を見るとvmlinuxで約34.81%イベントが発生しています。vmlinuxはLinuxカーネルですが、デバッグ情報が付与されていないので、シンボル名（symbol name）が（no symbols）となっています。

その次にL2キャッシュミスが多発しているのは、rubyのgc_mark()で約11.56%、そして、garbage_collect()で約7.97%となっています。

それでは、詳細情報を見てみましょう。

```
$ sudo opreport -d
Counted BSQ_CACHE_REFERENCE events (cache references seen by the bus unit) with a unit mask of 0x100 (read
2nd level cache miss) count 3000
warning: some functions compiled without debug information may have incorrect source line attributions
vma       samples  %          linenr info                 image name                 app name
symbol name
00000000 67373     34.8114  (no location information)    no-vmlinux                 no-vmlinux
(no symbols)
  c0102020 1       0.0015   (no location information)
…
```

```
08064790 22364     11.5554  gc.c:1273                    ruby                    ruby
gc_mark
  08064790 45       0.2012  gc.c:1273
  08064791 40       0.1789  gc.c:1273
  08064796 51       0.2280  (no location information)
  08064799 7        0.0313  gc.c:1273
  0806479c 19       0.0850  gc.c:1273
  080647a2 44       0.1967  gc.c:1273
  080647a8 37       0.1654  gc.c:1273
  080647ac 58       0.2593  (no location information)
  080647b2 18       0.0805  (no location information)
  080647b4 10       0.0447  gc.c:1295
  080647b7 43       0.1923  gc.c:1295
  080647bf 1        0.0045  gc.c:1295
  080647c1 16       0.0715  gc.c:1295
  080647c2 3        0.0134  gc.c:1295
  080647c8 4        0.0179  gc.c:1278
  080647ca 21697   97.0175  gc.c:1278          ←①
  080647cc 29       0.1297  gc.c:1278
  080647ce 39       0.1744  gc.c:1279
  080647d0 6        0.0268  gc.c:1279
  080647d2 20       0.0894  gc.c:1280
  080647d5 169      0.7557  gc.c:1282
  080647db 1        0.0045  gc.c:1280
  080647df 1        0.0045  gc.c:1282
  0806480b 2        0.0089  gc.c:1282
  08064840 2        0.0089  gc.c:1294
  08064845 1        0.0045  gc.c:1295
  0806484b 1        0.0045  gc.c:1295
08064b10 15421      7.9680  gc.c:1919                    ruby                    ruby
garbage_collect
  08064c29 1        0.0065  gc.c:1957
  08064c41 1        0.0065  gc.c:1961
  08064c4a 1        0.0065  gc.c:1962
  08064c4c 3        0.0195  gc.c:1962
  08064c6b 1        0.0065  (no location information)
  08064cb2 1        0.0065  gc.c:1975
  08064de9 1        0.0065  (no location information)
  08064df0 1        0.0065  (no location information)
  08064e09 1        0.0065  (no location information)
```

```
08064ebf 8        0.0519  (no location information)
08064ed8 170      1.1024  (no location information)
08064eda 162      1.0505  (no location information)
08064ee0 134      0.8689  (no location information)
08064ee6 13       0.0843  (no location information)
08064eec 73       0.4734  (no location information)
08064eef 24       0.1556  (no location information)
08064ef2 3        0.0195  (no location information)
08064ef5 102      0.6614  (no location information)
08064ef8 9        0.0584  (no location information)
08064efe 105      0.6809  (no location information)
08064f00 41       0.2659  (no location information)
08064f02 13205   85.6300  (no location information)   ←②
08064f05 305      1.9778  (no location information)
08064f07 194      1.2580  (no location information)
08064f09 70       0.4539  (no location information)
08064f0c 11       0.0713  (no location information)
08064f0f 305      1.9778  (no location information)
08064f11 36       0.2334  (no location information)
08064f14 2        0.0130  (no location information)
08064f16 61       0.3956  (no location information)
08064f19 27       0.1751  (no location information)
08064f20 234      1.5174  (no location information)
08064f23 38       0.2464  (no location information)
08064f25 1        0.0065  (no location information)
08064f2b 8        0.0519  (no location information)
08064f59 1        0.0065  (no location information)
08064f87 1        0.0065  (no location information)
08064f8e 1        0.0065  (no location information)
08064faa 4        0.0259  (no location information)
08064fc2 1        0.0065  (no location information)
08064fde 2        0.0130  (no location information)
08065010 1        0.0065  (no location information)
08065019 2        0.0130  (no location information)
...
```

それでは解析をしてみましょう。まず今回は vmlinux のところではなく、ruby の実装について追ってみることにします。

opreport -d での表示は下記のような形式です。

形式：アドレス イベント発生回数 関数内での発生率（100%） 発生場所

gc_mark() のアドレス（0x080647ca）でキャッシュミスが多発（21697 回）していて、gc_mark() 内で実に約 97.02%、その場所（gc.c ファイルの 1278 行目）で発生しているのがわかります①。

ソースコードを見てみましょう（1278 行）。obj->as.basic.flags でキャッシュミスが発生しているということがわかりました。

```
1271 static void
1272 gc_mark(rb_objspace_t *objspace, VALUE ptr, int lev)
1273 {
1274     register RVALUE *obj;
1275
1276     obj = RANY(ptr);
1277     if (rb_special_const_p(ptr)) return; /* special const not marked */
1278     if (obj->as.basic.flags == 0) return;       /* free cell */
1279     if (obj->as.basic.flags & FL_MARK) return;  /* already marked */
1280     obj->as.basic.flags |= FL_MARK;
1281
```

次に garbage_collect() 内のアドレス（0x08064f02）で、13205 回（約 85.63%）キャッシュミスが発生していますが、ソースコードの情報がありません（no location information）②。これは、おそらく呼び出している関数等がインライン展開などされたためと考えられます。

```
1917 static int
1918 garbage_collect(rb_objspace_t *objspace)
1919 {
…
1974     /* gc_mark objects whose marking are not completed*/
1975     while (!MARK_STACK_EMPTY) {
1976         if (mark_stack_overflow) {
1977             gc_mark_all(objspace);
1978         }
1979         else {
1980             gc_mark_rest(objspace);
1981         }
1982     }
```

gc_mark_all() を見てみましょう。

```
1089 gc_mark_all(rb_objspace_t *objspace)
1090 {
1091     RVALUE *p, *pend;
1092     size_t i;
1093
1094     init_mark_stack(objspace);
1095     for (i = 0; i < heaps_used; i++) {
1096         p = heaps[i].slot; pend = p + heaps[i].limit;
1097         while (p < pend) {
1098             if ((p->as.basic.flags & FL_MARK) &&
1099                 (p->as.basic.flags != FL_MARK)) {
1100                 gc_mark_children(objspace, (VALUE)p, 0);
1101             }
1102             p++;
1103         }
1104     }
1105 }
```

これを objdump を利用して逆アセンブルしてみます[†]。

```
$ objdump -CxS ruby
...
       p = heaps[i].slot; pend = p + heaps[i].limit;
        while (p < pend) {
 8064efb:       39 5d 84                cmp    %ebx,-0x7c(%ebp)
 8064efe:       76 25                   jbe    8064f25 <garbage_collect+0x415>
           if (!(p->as.basic.flags & FL_MARK)) {
 8064f00:       8b 13                   mov    (%ebx),%edx
 8064f02:       f6 c2 20                test   $0x20,%dl
 8064f05:       74 d1                   je     8064ed8 <garbage_collect+0x3c8>
               else {
                   add_freelist(objspace, p);
                   free_num++;
               }
           }
```

どこでキャッシュミスを多発しているのかが、わかりました[††]。

† objdump はオブジェクトファイルの各種情報を表示するコマンドです。

†† キャッシュとキャッシュミス。キャッシュとはメインメモリと CPU レジスタの間にあり、メインメモリのアクセス速度と CPU レジスタのそれとの差に着目し、より高速なアクセスができる小規模なメモリのことを言います。多くのプログラムのアクセスパターンには局所性があるので、高速で小規模なキャッシュは、システムの性能向上に貢献します。

キャッシュミスの削減方法

メインメモリの速度はCPUの速度に比較して遅いので、キャッシュを有効に利用することは性能向上には欠かせません。キャッシュミスとは、あるアクセスがキャッシュにとどまらなくてメインメモリへのアクセスを必要することを言います。キャッシュミスは性能劣化の原因になるので、キャッシュミス削減が重要な課題になります。

キャッシュミスの発生原因は、①最初のアクセス、②キャッシュの容量不足、③キャッシュラインコンフリクトなどが考えられます。それぞれキャッシュミスについての対策が知られています。

最初のアクセスでは、今まで一度もアクセスしたことがないので、当然キャッシュミスを発生させます。対策として、アクセスする前にあらかじめアクセスをしておいて、キャッシュに載せておくというのがあります。プリフェッチと呼ばれる技法です。

2番目はプログラムの実行範囲がキャッシュに比べて大きいときに発生します。例えば、8MBのメモリをコピーするとします。キャッシュのサイズが仮に2KBだとすると、コピー元はキャッシュに載りません。そのため、何度もキャシュに格納する、解放するというコストが発生しています。

3番目は、どのメモリがどのキャッシュに載り、場合によっては、同じキャッシュラインに格納されるために、コンフリクト（競合）が発生するということを表しています。

例えば、8KBのキャッシュがあったとして、メモリ空間を8KBで割った余りが等しい場合、同じキャッシュラインに載る場合があります。

```
int a[2048], b[2048];
for(i=0;i<n;i++)
  b[i] = a[i];
```

a[]のアドレスとb[]のアドレスは8KBの隔りで、8KBで割ると余りが等しいので、上記の例では、キャッシュミスが多発することになります。この例では、下記のように、キャッシュライン分、ずらすと余りが異なるので、キャッシュミスを減らすことができます。

```
#define CACHE_LINE_SIZE 128;   /* キャッシュラインの大きさ */
int a[2048];
char padding[CACHE_LINE_SIZE]; /* キャッシュライン分ずらす */
int b[2048];
for(i=0;i<n;i++)
  b[i] = a[i];
```

さて、今回のキャッシュミスは、ソースコードで確認したところ、最初のアクセスで発生していることがわかりました。そこで、プリフェッチをするパッチを書いてみました。

以下はそのパッチです。

```
$ svn diff
Index: gc.c
===================================================================
--- gc.c (revision 21331)
+++ gc.c (working copy)
@@ -1095,6 +1095,11 @@
     for (i = 0; i < heaps_used; i++) {
        p = heaps[i].slot; pend = p + heaps[i].limit;
        while (p < pend) {
+           if ( (p+1) < pend) {
+               __asm__ __volatile__ (
+                  " prefetch (%0)\n"
+                  : : "r" ((p+1)->as.basic.flags) );
+           }
           if ((p->as.basic.flags & FL_MARK) &&
              (p->as.basic.flags != FL_MARK)) {
              gc_mark_children(objspace, (VALUE)p, 0);
@@ -1657,6 +1662,11 @@

         p = heaps[i].slot; pend = p + heaps[i].limit;
         while (p < pend) {
+           if ( (p+1) < pend) {
+               __asm__ __volatile__ (
+                  " prefetch (%0)\n"
+                  : : "r" ((p+1)->as.basic.flags) );
+           }
            if (!(p->as.basic.flags & FL_MARK)) {
                   if (p->as.basic.flags &&
                       ((deferred = obj_free(objspace, (VALUE)p)) ||
```

これを元に同じテストを実行してみたところ、

```
CPU: P4 / Xeon, speed 2400 MHz (estimated)
Counted BSQ_CACHE_REFERENCE events (cache references seen by the bus unit) with a unit mask of 0x100 (read
2nd level cache miss) count 3000
samples  %        image name               app name                 symbol name
61186    33.2690  no-vmlinux               no-vmlinux               (no symbols)
22716    12.3515  ruby                     ruby                     gc_mark
11678     6.3497  ruby                     ruby                     garbage_collect
10014     5.4450  ruby                     ruby                     gc_mark_children
```

garbage_collect() のキャッシュミスが、15421 回から 11678 回へ減少していることが確認できました。

まとめ

oprofile を利用した性能調査とチューニング方法について ruby の実装を例にとってキャッシュミス削減方法を具体的に説明しました。

—— Hiro Yoshioka

HACK #61 VMware Vprobe を使用して情報を取得する

VMware Workstation 6.5 以降にある VProbe の機能を利用してゲスト OS の状態を調査することができます。

VProbe はハイパーバイザーレベルで動作し、仮想マシンと物理ハードウェアとの間のやり取りに関する情報を提供することができる分析・デバッグエンジンです。仮想 CPU レジスタ、ハードウェア仮想化ステータス、ゲスト OS ページフォルト、割り込み状況などのさまざまな情報を取得することができます。

Vprobe 機能の有効化手順

1. 以下の行を VMware の設定ファイル（config）に記述（追記）します。

```
vprobe.allow = TRUE

Linux の場合 : /etc/vmware/config
Windows の場合 : C:\Documents and Settings\All Users\Application Data\VMware\VMware
Workstation\config.ini
```

2. 対象の仮想マシンの設定ファイル（.vmx ファイル）に以下の行を記述（追記）します。

```
vprobe.enable = TRUE
```

Vprobe 機能の使用時の注意

Vprobe を使用するには対象の仮想マシンを起動しておく必要があります。

Vprobe 機能の状態確認

Vprobe 機能の使用は VMware Workstation と共にインストールされる vmrun コマンドを使用します。

```
$ ./vmrun vprobeVersion 'vmx ファイルのフルパス（以下 Linux.vmx）'
VProbes version: 0.2 (enabled)
```

状態が上記のように enabled となっていれば使用可能です。

Vprobe 機能のテスト

以下のようにするとテストが実行できます。

```
$ ./vmrun vprobeLoad 'Linux.vmx' '(vprobe VMM1Hz (printf "hello!\n"))'
```

.vmx と同じディレクトリに vprobe.out が作成され、hello! が 1 秒おきに出力されます。

上記は VMM1Hz という Vprobe で定義済みのスタティックプローブ（Static probe）で "hello!" という文字列と改行を出力せよと言う意味です。VMM1Hz は 1 秒に 1 回発生するスタティックプローブです。仮想マシンが動作している間、文字列が vprobe.out に出力されます。

出力停止、Vprobe 機能の停止

Vprobe 機能を停止し、出力を終了する場合は下記のように実行します。

```
$ ./vmrun vprobeReset 'Linux.vmx'
```

Vprobe 機能の使用例 boot デバイスの表示

次に特定のアドレスの命令が実行された際にプローブを実行するダイナミックプローブ（Dynamic probe）の例を示します。

1. 下記のファイルを作成します。なお、;（セミコロン）以下はコメントです。実際は入力する必要はありません。

```
$ cat printboot.emt
; Print the boot device.
(defstring device)       ; 文字列変数定義
(definteger dl)          ; 数値変数定義
(vprobe GUEST:0x7c00     ; ブートローダの開始アドレスでプローブ実行
(setint dl (& RDX 0xff)) ; RDX レジスタ下位 8bit=DL レジスタの内容を取得
(cond ((== dl 0x80)      ; 条件判断
(setstr device "hard drive"))
((== dl 0)
(setstr device "floppy drive"))
(1
(setstr device "unknown device CD etc.")))
(printf "Booting from %s (0x%x)\n" device dl))
```

2. 仮想マシン起動
   Vprobe のコードは仮想マシンが実行されていないと、読み込めないため、仮想マシンを起動して、F2キー等で BIOS セットアップ画面で止めておきます。

3. Vprobe の読み込み
   ホストから下記を実行します。

```
$ ./vmrun vprobeLoad 'Linux.vmx' "`cat printboot.emt`"
```

   スクリプトはコマンドラインからの読み込みとなるため、ダブルクオートとバッククオートでエスケープしています。

4. テスト
   ゲスト OS で BIOS セットアップを終了し、ブートを行うとブートデバイスに応じて vprobe.out に

```
Booting from hard drive (0x80)
Booting from floppy drive (0x0)
Booting from unknown device CD etc. (0x9f)
```

   等が出力されます。

関数名称でのアドレス指定

仮想マシンのアドレス情報を下記でシンボルファイルとして取得します。

```
# cat /proc/kallsyms > kallsyms.txt
```

上記で取得したシンボルファイルを vmx ファイルで下記のように指定することで、カーネルの関数名称でアドレスを指定することができます。

```
vprobe.guestSyms = "kallsyms.txt"
```

シンボルファイルの拡張子は .txt またはなしで指定して下さい。

シンボルファイルを使用した場合の例

1.

```
$ cat system_call.emt
(vprobe GUEST:system_call
(printf "Current RAX : 0x%016x  RSP : 0x%016x \n" RAX RSP))
```

2. Vprobe の読み込み

```
$ ./vmrun vprobeLoad 'Linux.vmx' "`cat system_call.emt`"
```

3. カーネルの system_call 関数が呼ばれたタイミングで、vprobe.out に出力されます。この場合、RAX（EAX）レジスタにはシステムコールの番号が入っていますので、何のシステムコールが呼ばれたかがわかります。

[例]

```
Current RAX : 0x0000000000000001  RSP : 0x00000000cee65fe4
Current RAX : 0x00000000000000af  RSP : 0x00000000cb2f1fe4
```

番号とシステムコールの対応はカーネルソースの下記のファイルに記載されています。

```
v2.6.23まで
include/asm-i386/unistd.h
include/asm-x86_64/unistd.h

v2.6.24
include/asm-x86/unistd_32.h
include/asm-x86/unistd_64.h
```

このように簡単に関数呼び出し時のレジスタの情報を取得することができます。詳細はレジスタについては、「**Intel アーキテクチャの基本**」**[HACK #8]**。関数呼び出しについては、「**関数コール時の引数の渡され方（x86_64 編）**」**[HACK #10]**、「**関数コール時の引数の渡され方（i386 編）**」**[HACK #11]** を参考にして下さい。

ホスト OS が Windows の場合、標準のコマンドプロンプトからの複数行の Vprobe のスクリプトの実行は難しいです。その場合、Cygwin のコンソールを使い、スクリプトファイル等を Linux と同様の方法で指定して、実行することができます。

まとめ

VMware の機能を使用することで、動的に内部状態を把握することができます。また、Vprobe で取得できる情報やスタティックプローブの種類については、下記参考資料にリファレンスがあります。参考にして下さい。

参考資料

- VProbes Programming Reference
  http://www.vmware.com/pdf/ws65_vprobes_reference.pdf

—— Shunsuke Yoshida

HACK #62 Xen でメモリダンプを取得する

Xen の仮想マシン（Domain-U、HVM Domain）で動作している Linux のメモリダンプを取得する方法について説明します。

Xen 上の仮想マシンとして Linux を動作させている場合には Xen のコンソール（Domain-0）から仮想マシンのメモリダンプを採取することができます。メモリダンプは仮想マシンを動作させたまま実行する livedump も可能です。

準仮想化（Paravirtualization）の Domain-U、完全仮想化（FullVirtualization）の HVM Domain、どちらの仮想マシンでもメモリダンプが可能です。

手順

Xen の管理 OS の Domain-0 にログインします。

稼動しているドメイン（仮想マシン）を xm list コマンドで確認します。

```
# xm list
Name                                        ID   Mem   VCPUs    State   Time(s)
Asianux3GA_HV                                1  2048       1   r-----    580.1
Domain-0                                     0   668       8   r-----    173.0
```

xm dump-core コマンドで dump ファイルを出力します。--live オプションで対象ドメインを動作させたまま、対象ドメインのメモリダンプが取得できます。

[例]

```
# xm dump-core --live Asianux3GA_HV /home/user/axhvmstall.live
Dumping core of domain: Asianux3GA_HV ...
```

実行しているメモリサイズ分の dump ファイルが出力されますので、出力先のパーティションの空き容量は注意して下さい。

出力された dump ファイルは通常の crash コマンドで解析できます。

```
# crash System.map-2.6.18-xxx vmlinux axhvmstall.live
( 中略 )
  SYSTEM MAP: System.map-2.6.18-xxx
DEBUG KERNEL: vmlinux (2.6.18-xxx)
    DUMPFILE: dump/axhvmstall.live
        CPUS: 1
        DATE: Sun Sep 28 11:14:22 2008
      UPTIME: 00:53:02
LOAD AVERAGE: 0.00, 0.00, 0.00
       TASKS: 50
    NODENAME: axs3fullovm.miraclelinux.com
     RELEASE: 2.6.18-xxx
     VERSION: #1 SMP Sun Mar 16 20:22:54 EDT 2008
     MACHINE: i686  (2660 Mhz)
      MEMORY: 2 GB
       PANIC: ""
         PID: 0
     COMMAND: "swapper"
        TASK: c0664bc0  [THREAD_INFO: c06d9000]
         CPU: 0
       STATE: TASK_RUNNING
     WARNING: panic task not found
```

--crash オプションを指定した場合はメモリダンプを取得し、ドメインを停止させます。

[例]

```
# xm dump-core --crash Asianux3GA_HV /home/user/axhvmstall.crash
Dumping core of domain: Asianux3GA_HV ...
```

その他 Xen の情報について

『Xen 徹底入門』（翔泳社刊、ISBN:978-4-7981-1447-7）が参考になります。

—— Shunsuke Yoshida

HACK #63 GOT/PLT を経由した関数コールの仕組みを理解する

プログラムの流れをアセンブラレベルで調査するときに不可欠な GOT/PLT を使用した関数のコールの仕組みを説明します。

プログラムと共有ライブラリ

多くのプログラムでは、実行サイズを小さくできる共有ライブラリが使用されています。共有ライブラリ内のコード領域やデータ領域は、それを使用するプログラムの実行時に、そのプログラムの仮想メモリ空間にマップされます。その様子を具体的に見てみましょう。

いま、ライブラリ関数 rand() を使用する次のソースコードをビルドします。

[test1.c]

```
#include <stdio.h>
#include <stdlib.h>

int func1(void)
{
    return 1;
}

int main(void)
{
    int a, b;
    a = func1();
    b = rand();
    return a + b;
}
```

ここでは、仮想空間のメモリマップを確認するため、GDB を使って、このプログラムを途中で停止します。

```
$ gdb test1
...
(gdb) start
Breakpoint 1 at 0x400487
Starting program: /root/tmp/test1
0x0000000000400487 in main ()
```

この状態で、別のターミナルから次のように入力すると、test1の仮想メモリを表示させることができます。

```
$ cat /proc/`pidof test1`/maps
00400000-00401000 r-xp 00000000 fd:00 166905                 /root/tmp/test1
00600000-00601000 rw-p 00000000 fd:00 166905                 /root/tmp/test1
37d7e00000-37d7e1a000 r-xp 00000000 fd:00 360453             /lib64/ld-2.5.so
37d8019000-37d801a000 r--p 00019000 fd:00 360453             /lib64/ld-2.5.so
37d801a000-37d801b000 rw-p 0001a000 fd:00 360453             /lib64/ld-2.5.so
37d8200000-37d8344000 r-xp 00000000 fd:00 360460             /lib64/libc-2.5.so
37d8344000-37d8544000 ---p 00144000 fd:00 360460             /lib64/libc-2.5.so
37d8544000-37d8548000 r--p 00144000 fd:00 360460             /lib64/libc-2.5.so
37d8548000-37d8549000 rw-p 00148000 fd:00 360460             /lib64/libc-2.5.so
37d8549000-37d854e000 rw-p 37d8549000 00:00 0
2aaaaaaab000-2aaaaaaac000 rw-p 2aaaaaaab000 00:00 0
2aaaaaad7000-2aaaaaad9000 rw-p 2aaaaaad7000 00:00 0
7fff277ee000-7fff27803000 rw-p 7fff277ee000 00:00 0          [stack]
ffffffffff600000-ffffffffffe00000 ---p 00000000 00:00 0      [vdso]
```

test1では、ld-2.5.soとlibc-2.5.soの2つの共有ライブラリが使用されていることがわかります。ld-2.5.soは、主に共有ライブラリを使用するための関数を提供します。そのため、プログラムからこの共有ライブラリが提供する関数を明示的に呼ぶことは、あまり多くありません。一方、libc-2.5.soは、printf()、malloc()やrand()など、C言語の主要なライブラリ関数を提供します。

いま、libc-2.5.soは、0x37d8200000というアドレスにマップされていますが、このアドレスは、プログラムの実行時に決まります。また、実行ごとに異なることもあります。そのため、rand()のアドレスは、ビルド時にはわかりません。そのため、実行時に共有ライブラリ中の関数のアドレスを調べて、呼び出す仕組みが必要です。GNU/Linuxシステムでは、この仕組みにPLT（Procedure Linkage Table）とGOT（Global Offset Table）というプログラム中の領域が使用されています。

PLTとGOTを簡単に説明すると、次のようなものです。GOTは、ライブラリ関数のアドレスを保持するための領域です。この領域には、プログラムの実行時に、使用するライブラリ関数のアドレスが設定されます。PLTは、ライブラリ関数を呼び出す小さなコードの集まりです。これらのコードは、プログラムから、そのプログラムに含まれるユーザ関数と同様にコールすることができます。そのため、PLTには、おおよそ、使用するライブラリ関数と同じ数のその小さなコードが含まれます。このコードの動作は、簡単に言えば、GOTに設定されている値にジャンプするだけです。GOTにまだ、呼び出す関数の

アドレスが設定されていな場合、その値を GOT に設定してから、ジャンプします。ただし、ライブラリ関数のアドレスは、プログラム実行中に変わることがないので、GOT の値は一度設定されると、その後、変更されることはありません。そのため、ライブラリ関数が、呼び出される度に GOT に値が設定されているか否かをチェックするのは、無駄な操作です。glibc では、巧妙な方法を用いてそのような無駄が発生しない呼び出しを行います。その方法については、後ほど説明します。

関数コール

ここでは、どのように PLT や GOT が利用されるかを先ほどの test1 を使って、実際に見ていきます。先ほどの GDB で start コマンドを入力した状態で、main() を逆アセンブルすると、次のようなコードにコンパイルされていることがわかります。

```
(gdb) disas main
Dump of assembler code for function main:
0x0000000000400483 <main+0>:    push   %rbp
0x0000000000400484 <main+1>:    mov    %rsp,%rbp
0x0000000000400487 <main+4>:    sub    $0x10,%rsp
0x000000000040048b <main+8>:    callq  0x400478 <func1>
0x0000000000400490 <main+13>:   mov    %eax,0xfffffffffffffff8(%rbp)
0x0000000000400493 <main+16>:   callq  0x4003a8 <rand@plt>
0x0000000000400498 <main+21>:   mov    %eax,0xfffffffffffffffc(%rbp)
0x000000000040049b <main+24>:   mov    0xfffffffffffffffc(%rbp),%eax
0x000000000040049e <main+27>:   add    0xfffffffffffffff8(%rbp),%eax
0x00000000004004a1 <main+30>:   leaveq
0x00000000004004a2 <main+31>:   retq
```

自前の関数 func1() と、ライブラリ関数 rand() の呼び出し方は、どちらも相対アドレスを指定した call 命令です。func1() は、0x400478 番地からに次のような関数の実態が存在します。プログラムに含まれるユーザ関数は、ビルド時に呼び出し元との相対的な配置関係が確定できるため、通常、このように、相対アドレス指定の call 命令を使って、直接呼び出されます。

```
(gdb) disas func1
0x0000000000400478 <func1+0>:   push   %rbp
0x0000000000400479 <func1+1>:   mov    %rsp,%rbp
0x000000000040047c <func1+4>:   mov    $0x1,%eax
0x0000000000400481 <func1+9>:   leaveq
0x0000000000400482 <func1+10>:  retq
```

rand() の呼び出しでも、一見、func1() と同じように相対アドレス指定の call 命令が使用されています。コール先のアドレス 0x4003a8 のアセンブラコードを見てみましょう。

```
(gdb) disas 0x4003a8
Dump of assembler code for function rand@plt:
0x00000000004003a8 <rand@plt+0>:        jmpq   *2098370(%rip)
                                        # 0x600870 <_GLOBAL_OFFSET_TABLE_+32>
0x00000000004003ae <rand@plt+6>:        pushq  $0x1
0x00000000004003b3 <rand@plt+11>:       jmpq   0x400388
```

最初にジャンプ命令があります。ジャンプ先は、0x600870 番地に格納されている値です。その値は、次のように調べると、0x4003ae です。つまり、この jmpq 命令の次のアドレスです。

```
(gdb) x 0x600870
0x600870 <_GLOBAL_OFFSET_TABLE_+32>:    0x004003ae
```

結局、0x4003a8 番地の jmpq 命令が実行されても、次の命令に実行が移るだけです。実は、これは、GOT にまだ呼び出すライブラリ関数のアドレスが設定されていないためです。0x4003ae 以降の命令は、0x600870 番地に rand() 関数のアドレスを設定します。その設定が行われた後は、0x4003a8 番地の jmpq 命令で、直接、rand() が呼び出されるようになります。これが先に述べた GOT が設定されているか否かを無駄なくチェックする仕組みです。それでは、0x4003ae 番地以降の命令を追っていきましょう。push 命令に続いて、0x400388 番地にジャンプします。このアドレスのコードは、シンボル情報の関係で単純にアドレスを引数にするだけでは、次のように逆アセンブルできないので、とりあえず、x コマンドを使って、5 命令ほど表示させます。

```
(gdb) disas 0x400388
No function contains specified address.
(gdb) x/5i 0x400388
0x400388: pushq  2098378(%rip)     # 0x600858 <_GLOBAL_OFFSET_TABLE_+8>
0x40038e: jmpq   *2098380(%rip)    # 0x600860 <_GLOBAL_OFFSET_TABLE_+16>
…
```

2 つ目の命令でジャンプするので、3 つ目以降の命令はとりあえず関係ありません。ここでは省略します。アドレス 0x40038e 目の jmpq 命令は、下記のように _dl_runtime_resolve() という関数（厳密にはリターンしないので関数ではありませんが）を呼び出します。<_dl_runtime_resolve+61> でコールされる _dl_fixup() は、まさに rand() など、ライブラリ関数のアドレスを GOT に設定します。これ以降の処理はまだまだ続くので、興味があれば

GLIBC のソースを読んでみてください。

ちなみに、ここにたどり着くまでに 2 回 push 命令がありましたが、スタックに積まれたこれらの値は、これ以降の処理の引数として使用されます。_dl_fixup() 関数は、呼び出す関数のアドレスを rax に返します。_dl_runtime_resolve() では、それはさらに r11 に代入され、最後には、jmpq 命令のオペコードとなって、呼び出す関数へジャンプするために使われます。

```
(gdb) x 0x600860
0x600860 <_GLOBAL_OFFSET_TABLE_+16>:    0x00000037d7e122a0
(gdb) disas 0x00000037d7e122a0
Dump of assembler code for function _dl_runtime_resolve:
0x00000037d7e122a0 <_dl_runtime_resolve+0>:     sub    $0x38,%rsp
0x00000037d7e122a4 <_dl_runtime_resolve+4>:     mov    %rax,(%rsp)
0x00000037d7e122a8 <_dl_runtime_resolve+8>:     mov    %rcx,0x8(%rsp)
0x00000037d7e122ad <_dl_runtime_resolve+13>:    mov    %rdx,0x10(%rsp)
0x00000037d7e122b2 <_dl_runtime_resolve+18>:    mov    %rsi,0x18(%rsp)
0x00000037d7e122b7 <_dl_runtime_resolve+23>:    mov    %rdi,0x20(%rsp)
0x00000037d7e122bc <_dl_runtime_resolve+28>:    mov    %r8,0x28(%rsp)
0x00000037d7e122c1 <_dl_runtime_resolve+33>:    mov    %r9,0x30(%rsp)
0x00000037d7e122c6 <_dl_runtime_resolve+38>:    mov    0x40(%rsp),%rsi
0x00000037d7e122cb <_dl_runtime_resolve+43>:    mov    %rsi,%r11
0x00000037d7e122ce <_dl_runtime_resolve+46>:    add    %r11,%rsi
0x00000037d7e122d1 <_dl_runtime_resolve+49>:    add    %r11,%rsi
0x00000037d7e122d4 <_dl_runtime_resolve+52>:    shl    $0x3,%rsi
0x00000037d7e122d8 <_dl_runtime_resolve+56>:    mov    0x38(%rsp),%rdi
0x00000037d7e122dd <_dl_runtime_resolve+61>:    callq  0x37d7e0ca50 <_dl_fixup>
0x00000037d7e122e2 <_dl_runtime_resolve+66>:    mov    %rax,%r11
0x00000037d7e122e5 <_dl_runtime_resolve+69>:    mov    0x30(%rsp),%r9
0x00000037d7e122ea <_dl_runtime_resolve+74>:    mov    0x28(%rsp),%r8
0x00000037d7e122ef <_dl_runtime_resolve+79>:    mov    0x20(%rsp),%rdi
0x00000037d7e122f4 <_dl_runtime_resolve+84>:    mov    0x18(%rsp),%rsi
0x00000037d7e122f9 <_dl_runtime_resolve+89>:    mov    0x10(%rsp),%rdx
0x00000037d7e122fe <_dl_runtime_resolve+94>:    mov    0x8(%rsp),%rcx
0x00000037d7e12303 <_dl_runtime_resolve+99>:    mov    (%rsp),%rax
0x00000037d7e12307 <_dl_runtime_resolve+103>:   add    $0x48,%rsp
0x00000037d7e1230b <_dl_runtime_resolve+107>:   jmpq   *%r11
…
```

main() の 0x400493 で rand@plt がコールされて以来、_dl_runtime_resolve() までは、ジャンプの連続で、call は、一度も呼ばれていません。そのため、_dl_runtime_resolve() の最後でジャ

ンプした後、ジャンプ先の関数で ret 命令が呼ばれた時、復帰するアドレスは、main() 関数の次のアドレス 0x400498 になります。ここまで見たように、実際には紆余曲折を経て、目的のライブラリ関数をコールしますが、main の中では、あたかもユーザ関数を呼んだように同じように振る舞います。

GOT が設定されたことの確認

それでは、_dl_runtime_resolve() の中で、GOT が設定されたことを実際に確認してましょう。

```
(gdb) b *0x00000037d7e1230b ←――――――――――――――――――――_dl_runtime_resolve の最後のアドレス
Breakpoint 2 at 0x37d7e1230b
(gdb) c
Continuing.

Breakpoint 2, 0x00000037d7e1230b in _dl_runtime_resolve ()
   from /lib64/ld-linux-x86-64.so.2
(gdb) x 0x600870
0x600870 <_GLOBAL_OFFSET_TABLE_+32>:    0x00000037d8233a70
(gdb) x/i  0x00000037d8233a70
0x37d8233a70 <rand>:    sub    $0x8,%rsp
```

ブレークポイントは、_dl_runtime_resolve() の最後の jmpq 命令に設定しました。そこまで実行させたあと、rand@plt でジャンプするアドレス（0x600870 に格納されている値）を確認すると、それは、0x37d8233a70 であり、rand() 関数の先頭アドレスであることがわかります。すなわち、次回の rand@plt の呼び出しでは、_dl_runtime_resolve() を経由せずに直接、rand() にジャンプします。

まとめ

プログラムをトレースする際、頻繁に遭遇する PLT/GOT を利用したライブラリ関数コールの仕組みを説明しました。

―― Kazuhiro Yamato

HACK #64 initramfs イメージをデバッグ

多くのディストリビューションが起動時に使用する initramfs の実行をデバッグする方法を紹介します。

initramfs とは

initramfs は、ほとんどのディストリビューションで root ファイルシステムをマウントするために使用されています。それらの Linux ディストリビューションでは、SCSI カードやファイルシステムのような root ファイルシステムをマウントするために必要な機能もカーネルモジュールとして構築されています。というのも、使用している Disk のインタフェース（SATA か SCSI か）やそのカードの種類、root ファイルシステムのフォーマット（ext3、ext4、xfs、reiserfs など）などは、ユーザによって異なるからです。モジュール化することによってユーザは、自身のシステムに必要なモジュールのみをロードすることで、メモリ消費量を節約できます。ところが、それら root ファイルシステムをマウントするための、ディスクやファイルシステムのモジュールは、root ファイルシステムに格納されています。これは、カーネルが起動の最終段階で root ファイルシステムをマウントしようとしても、そのファイルシステムを扱うための機能をカーネル内部に持っていないため、マウントに失敗することを意味します。

これを解決するのが initramfs です。initramfs は、root ファイルシステムのマウントに必要なスクリプトやモジュールが展開されるための RAM 上に作成されるファイルシステムです。それらのスクリプトやモジュールは、gzip 圧縮された cpio アーカイブとして用意されます。カーネルは、起動の最終段階で、このアーカイブの中の init を実行します。この init には、ディスクやファイルシステムのモジュールをロードする処理が記述されており、最終的にはディスク上のルートファイルシステムパーティションを / にマウントします。

initramfs は、カーネル 2.6 から登場しました。それまでは、同様の役割を果たす initrd が用いられてきました。両者の違いは、initramfs は、gzip 圧縮された cpio イメージを使用するのに対して、initrd では、ext2 等のファイルシステムイメージを使用することです。後者の場合、ファイルシステムを作成する際、そのサイズが固定化されるため、柔軟性に欠けます。また、カーネルの内部実装も initramfs のほうが initrd よりもシンプルなため、現在では、initramfs が主流となっています。

initramfs のデバッグ

システムの起動時に比較的よく発生する問題は、root ファイルシステムがマウントできないことに起因するカーネルパニックや、処理途中での停止です。多くの場合、initramfs

内に必要なモジュールやコマンドが格納されていないことに起因しますが、多くのディストリビューションのinitramfsのスクリプトは、それらの詳細なメッセージを表示しません。そのため、問題が発生したとき、何が起こっているのかを知ることが困難な場合が多くあります。このHackでは、initramfsにデバッグメッセージを追加することで、どこで問題が発生しているかを知る方法を紹介します。

これまでの述べたようにinitramfsの中身は、gzipされたcpioアーカイブです。それらは通常、/bootディレクトリに格納されています。例えば、Fedora10では、次のようにカーネルバージョンごとにinitramfsが存在します。これらは、（おそらく歴史的な理由から）initrdという名前で始まっていますが、initramfs用のアーカイブです。

```
# ls -l /boot/initrd*
-rw------- 1 root 3.8M Jan  3 00:15 /boot/initrd-2.6.27.5-117.fc10.x86_64.img
-rw------- 1 root 3.9M Jan  3 01:18 /boot/initrd-2.6.27.9-159.fc10.x86_64.img
-rw------- 1 root 3.9M Jan  4 18:44 /boot/initrd-2.6.27.9.img
```

これらを展開して、内容を確認してみます。

```
# mkdir work
# cd work
# gunzip -c /boot/initrd-2.6.27.9-159.fc10.x86_64.img | cpio -id
# ls -l
total 40
drwx------ 2 root root 4096 Jan 25 21:59 bin
drwx------ 3 root root 4096 Jan 25 21:59 dev
drwx------ 5 root root 4096 Jan 25 21:59 etc
-rwx------ 1 root root 1933 Jan 25 21:59 init
drwx------ 6 root root 4096 Jan 25 21:59 lib
drwx------ 2 root root 4096 Jan 25 21:59 lib64
drwx------ 2 root root 4096 Jan 25 21:59 proc
lrwxrwxrwx 1 root root    3 Jan 25 21:59 sbin -> bin
drwx------ 2 root root 4096 Jan 25 21:59 sys
drwx------ 2 root root 4096 Jan 25 21:59 sysroot
drwx------ 4 root root 4096 Jan 25 21:59 usr
# cat init
#!/bin/nash

mount -t proc /proc /proc
setquiet
echo Mounting proc filesystem
```

```
echo Mounting sysfs filesystem
mount -t sysfs /sys /sys
echo Creating /dev
mount -o mode=0755 -t tmpfs /dev /dev                    (a)
mkdir /dev/pts
mount -t devpts -o gid=5,mode=620 /dev/pts /dev/pts
mkdir /dev/shm
mkdir /dev/mapper
echo Creating initial device nodes
mknod /dev/null c 1 3
mknod /dev/zero c 1 5
mknod /dev/systty c 4 0
mknod /dev/tty c 5 0                                     (b)
mknod /dev/console c 5 1
<< 省略 >>
/lib/udev/console_init tty0
daemonize --ignore-missing /bin/plymouthd
plymouth --show-splash
echo Setting up hotplug.
hotplug
echo Creating block device nodes.
mkblkdevs
echo Creating character device nodes.
mkchardevs
echo "Loading scsi_transport_spi module"
modprobe -q scsi_transport_spi
echo "Loading mptbase module"
modprobe -q mptbase
echo "Loading mptscsih module"
modprobe -q mptscsih                                     (c)
echo "Loading mptspi module"
modprobe -q mptspi
echo Making device-mapper control node
mkdmnod
modprobe scsi_wait_scan
rmmod scsi_wait_scan
mkblkdevs
echo Scanning logical volumes
lvm vgscan --ignorelockingfailure
echo Activating logical volumes
```

```
lvm vgchange -ay --ignorelockingfailure  VolGroup00
resume /dev/VolGroup00/LogVol01
echo Creating root device.
mkrootdev -t ext3 -o defaults,ro /dev/VolGroup00/LogVol00
echo Mounting root filesystem.
mount /sysroot                                          (d)
cond -ne 0 plymouth --hide-splash
echo Setting up other filesystems.
setuproot
loadpolicy
plymouth --newroot=/sysroot
echo Switching to new root and running init.
switchroot
echo Booting has failed.
sleep -1
```

このようにFedora10では、initの実態はスクリプトです。この中では/procなどマウントなどの基本設定(a)、最低限のデバイスファイルの作成(b)、モジュールのロード(c)、ディクス上のルートファイルシステムパーティションのマウント(d)などが行われています。echoコマンドを使ったメッセージが、ところどころで、表示されていますが、問題となる箇所を特定するために、1行ごとにechoコマンドを挿入したい場合もあります。そこで、(d)のloadpolicyとplymouthの実行直前にもecho messageを表示させてみます。

デバッグ用initramfsの作成

まず、展開したinitを編集します。

```
echo Setting up other filesystems.
setuproot
echo Debug: executes loadpolicy ←――――――――――――追加
loadpolicy
echo Debug: executes plymouth   ←――――――――――――追加
plymouth --newroot=/sysroot
echo Switching to new root and running init.
```

次に、下記のようにして、再度initramfsを作成します。

```
# find | cpio -o -H newc | gzip - -c > /boot/initrd-2.6.27.9-159.fc10.debug.x86_64.img
```

このinitramfsを、起動時に読み込ませるために ブートローダーのinitrd項目を設定し

ます。使用しているブートローダーが grub なら、以下のような項目 /boot/grub/menu.lst（または /etc/grub.conf）に追加します。また、デバッグ目的ですので、カーネルオプションの quiet（メッセージのコンソール出力を抑制するオプション）や rhgb（Fedora でグラフィカル起動画面を表示するためのオプション）は指定しない方が、よいでしょう。

```
title Fedora (2.6.27.9-159.fc10.x86_64) : Debug
        root (hd0,0)
        kernel /vmlinuz-2.6.27.9-159.fc10.x86_64 ro root=/dev/VolGroup00/LogVol00
        initrd /initrd-2.6.27.9-159.fc10.debug.x86_64.img ―――――――― 作成した initramfs を指定
```

システムを上記のエントリで起動した画面を以下に示します。追加した echo 行が表示されているのがわかります。もしエラーの発生や、フリーズすることがあれば、これでその場所を特定できます。

```
…
Creating root device.
Mounting root filesystem.
Setting up other filesystem.
Debug: executes loadpolicy
Debug: executes plymouth
Switching to new root and running init.
```

上記方法で、不具合を起こしている箇所が特定できても、不具合がストール等の場合、すぐにその原因はわかりません。このよう場合は、**「SysRq キーによるデバッグ方法」[HACK #18]** を参考にさらに調査をしてください。

まとめ

多くのディストリビューションが起動時に使用する initramfs の実行をデバッグする方法を紹介しました。

—— Kazuhiro Yamato

HACK #65 RT Watchdog を使ってリアルタイムプロセスのストールを検知する

Linux 2.6.25 以降から利用可能となった RT Watchdog とその使い方について説明します。

「リアルタイムプロセスのストール」[HACK #40] でも触れていますが、リアルタイムプロセスが暴走するとシステム全体に応答性が悪くなるなどの影響が出てしまいます。これ

に対して Linux では 2 つの対策が取られています。ひとつはプロセススケジューラ（正確にはタスクスケジューラ）自体がリアルタイムプロセスに割り当てる CPU 時間に制限を設けるというものです。これは Linux 2.6.23 から導入された CFS と呼ばれるスケジューラによって実現されています。これはすべてのリアルタイムプロセスを対象とします。もうひとつは本 Hack で紹介する RT Watchdog です。こちらは setrlimit(2) を拡張して実装されており、特定のリアルタイムプロセスを対象とします。RT Watchdog は Linux 2.6.25 以降から利用可能です。

RT Watchdog とは

ブロッキング API をコールせずに CPU 時間を使い続けるようなリアルタイムプロセスを検知し、あらかじめユーザが指定した制限値に達するとそのプロセスにシグナルを送信します。setrlimit(2) で RLIMIT_RTTIME に最大 CPU 時間（ソフトリミット、ハードリミット）を設定することで RT Watchdog が有効化されます。最大 CPU 時間はマイクロ秒単位で指定します。ソフトリミットに達すると SIGXCPU シグナルが、ハードリミットに達すると SIGKILL シグナルが当該プロセスに送信されます。

RT Watchdog の動作確認

ブロッキング API とはそのプロセスがスリープや I/O 待ちになるようなシステムコールのことです。read() や write()、sleep()、select()、recv() のようなシステムコールが該当します。注意としては、sched_yield() システムコールはブロッキング API ではないということです。sched_yield() は CPU を能動的に手放すシステムコールですが、これはブロッキング API とは見なされません。

次のようなサンプルプログラムを用意しました。これを使って RT Watchdog の動作を確認してみましょう。このプログラムでは、動作の少しずつ異なる子プロセスを 3 つ fork() しています。それぞれの動作の違いはソースコード上のコメントを参照してください。

```
$ cat rt-watchdog.c
#include <sched.h>
#include <sys/time.h>
#include <sys/resource.h>
/*
 * 筆者の環境 (Fedora9) では RLIMIT_RTTIME の定義が <sys/resource.h> にはない
 * ため、<asm/resource.h> をインクルードしている
 */
#include <asm/resource.h>
```

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>

#define USEC_PER_SEC   (1000000UL)
#define loop_10sec(start)   \
   for (start=time(NULL); time(NULL) < (start+10); )

int main(void)
{
   time_t start;
   struct rlimit rl;
   struct sched_param param;

   /* ソフトリミットを1秒、ハードリミットを4秒に設定 */
   rl.rlim_cur = USEC_PER_SEC;
   rl.rlim_max = USEC_PER_SEC << 2;
   setrlimit(RLIMIT_RTTIME, &rl);

   /* このプロセスをリアルタイムクラスに変更 */
   param.sched_priority = sched_get_priority_min(SCHED_RR);
   sched_setscheduler(0, SCHED_RR, &param);

   if ( fork() == 0 ) {
      /*
       * 子プロセス1は、10秒間 sched_yield() システムコールを
       * 連続コールする
       */
      printf("%lu: Child-1: PID%d\n", time(NULL), getpid());
      loop_10sec(start)
         sched_yield();

   } else if ( fork() == 0 ) {
      /*
       * 子プロセス2は、SIGXCPU シグナル受信時の動作を
       * 変更し、10秒間ループする
       */
```

```
        struct sigaction act;
        memset(&act, 0, sizeof(act));
        act.sa_handler = SIG_IGN;
        sigaction(SIGXCPU, &act, NULL);

        printf("%lu: Child-2: PID%d\n", time(NULL), getpid());
        loop_10sec(start)
            ;

    } else if ( fork() == 0 ) {
        /*
         * 子プロセス3は、10秒間usleep()を連続コールする
         */
        printf("%lu: Child-3: PID%d\n", time(NULL), getpid());
        loop_10sec(start)
            usleep(1);

    } else {
        /* 親プロセスは子プロセスの終了ステータスをチェック */
        int status, i;
        pid_t pid;

        for (i = 0; i < 3; i++) {
            pid = wait(&status);
            if (WIFSIGNALED(status)) {
                if (WTERMSIG(status) == SIGKILL)
                    printf("%lu: PID%d is terminated by SIGKILL\n", time(NULL), pid);
                else if (WTERMSIG(status) == SIGXCPU)
                    printf("%lu: PID%d is terminated by SIGXCPU\n", time(NULL), pid);
            } else if (WIFEXITED(status))
                printf("%lu: PID%d normally exits\n", time(NULL), pid);
        }
    }

    return 0;
}
```

このプログラムを実行すると結果は次のようになります[†]。

† このプログラムはスケジューリングポリシーをリアルタイムクラスへ変更するため、実行するにはスーパーユーザになっておく必要があります。

```
# gcc rt-watchdog.c -o rt-watchdog
# ./rt-watchdog
1226852952: Child-1: PID5610
1226852952: Child-2: PID5611
1226852953: Child-3: PID5612
1226852953: PID5610 is terminated by SIGXCPU
1226852956: PID5611 is terminated by SIGKILL
1226852963: PID5612 normally exits
```

子プロセス 1 は sched_yield() を連続コールするプロセスです。起動から約 1 秒後にソフトリミットに達して SIGXCPU を受信しています。子プロセス 2 はブロッキング API をコールせずにループするプロセスです。ただ、SIGXCPU を無視するように設定しているため、起動から約 4 秒後にハードリミットに達したため SIGKILL で強制終了させられています。子プロセス 3 はブロッキング API をコールしているため、暴走プロセスとは見なされません。RT Watchdog に引っかかることなく最後まで動作することができました。

まとめ

リアルタイムプロセスの暴走を検知する仕組みである RT Watchdog について使用方法と動作の説明をしました。リアルタイムアプリケーションを Linux 上で作り込む場合、RT Watchdog を設定しておくことで、万が一アプリケーションが暴走した場合に復旧させる処理を作り込むことも可能です。

—— Toyo Abe

HACK #66 手元の x86 マシンが 64 ビットモード対応かどうかを調べる

プロセッサ情報を読み取って 64 ビットモードに対応しているか調べる方法を紹介します。

近頃は x86 系のプロセッサも 64 ビットモード対応が当たり前のようになってきました。自分の持っている PC が 64 ビット対応かどうかなんて所有者からすれば知っていて当然かもしれません。しかし本 Hack では、あえてそれを調べる方法を紹介します。

proc ファイルシステム上で調べる

動作している OS が Linux であれば /proc/cpuinfo の内容を見ることで 64 ビットモードに対応しているかどうかわかります。

```
# cat /proc/cpuinfo
processor       : 0
```

```
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
…
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
 mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe sy
scall nx lm constant_tsc arch_perfmon pebs bts rep_good nopl pni monit
or ds_cpl est tm2 ssse3 cx16 xtpr lahf_lm
…
```

ここで flags の行に lm という表記があるかどうかを確認してください。lm とは Long Mode の略で、64 ビットモードに対応していることを表しています。

CPUID 命令で調べる

プロセッサの情報を取得できる CPUID 命令を使って、プロセッサがどんな機能をサポートしているかを調べることができます。これを利用して 64 ビットモードに対応しているかどうか調べるプログラムを書いてみました。このプログラムは GCC でコンパイルする環境があればよいので、FreeBSD や Cygwin などでも使えます。

```
$ cat chklm.c
#include <stdio.h>

/* x86-64 Long Mode flag */
#define X86_FEATURE_LM (1<<29)

void cpuid(int op, unsigned int *eax, unsigned int *ebx,
                    unsigned int *ecx, unsigned int *edx)
{
        __asm__("cpuid"
                : "=a" (*eax),
                  "=b" (*ebx),
                  "=c" (*ecx),
                  "=d" (*edx)
                : "0" (op));
}

int main(void)
{
         unsigned int eax,ebx,ecx,edx;
```

```
        /* 拡張機能 CPUID 情報が取得可能かをチェック */
        cpuid(0x80000000, &eax, &ebx, &ecx, &edx);
        if (eax < 0x80000001)
                goto no_longmode;

        /* Long Mode ビットをチェック */
        cpuid(0x80000001, &eax, &ebx, &ecx, &edx);
        if (!(X86_FEATURE_LM & edx))
                goto no_longmode;

        printf("x86_64 Long Mode is supported.\n");
        return 0;
no_longmode:
        printf("x86_64 Long Mode is not supported.\n");
        return 1;
}
```

これを 64 ビットモード対応のシステム上で実行すると、こうなります。

```
$ gcc -o chklm chklm.c
$ ./chklm
x86_64 Long Mode is supported.
```

まとめ

本書を執筆中に「どうしたら自分の PC が確実に 64 ビット対応であると言えるのか？」という話題が執筆メンバー内で盛り上がりました。これを Hack に入れようという話になり、まとめたのが本 Hack です。本 Hack はデバッグには関係ありませんが、興味を持っていただければ幸いです。

参考

- インテル ® エクステンデッド・メモリ 64 テクノロジ・ソフトウェア・デベロッパーズ・ガイド
  http://download.intel.com/jp/developer/jpdoc/EM64T_VOL1_30083402_i.pdf
- インテル ® プロセッサの識別と CPUID 命令
  http://download.intel.com/jp/developer/jpdoc/Processor_Identification_071405_i.pdf

- AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions
  http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24594.pdf

—— Toyo Abe

付録

Debug Hacks 用語の基礎知識

ここでは、本書に登場する用語のうち代表的なものについて簡単に解説します。

ABI

Application Binary Interface の略。ソースコードレベルでのインタフェースを規定した Application Programming Interface（API）とは異なり、オブジェクトコード（つまりバイナリ）レベルでのインタフェースを規定したもので、CPU アーキテクチャごとに存在する。具体的には関数コールにおける引数の渡し方や戻り値の返し方などが規定されている。デバッグ情報がない実行ファイルを GDB でデバッグしたり、クラッシュダンプ解析を行う時に、ABI を知っている人と知らない人とではデバッグ力に差がつく。

APIC、Local APIC、I/O APIC

Advanced Programmable Interrupt Controller の略で、SMP システムにおいて割り込み信号を管理するコントローラ。APIC には、各 CPU の内部に実装される Local APIC と、周辺機器からの割り込みを受け付け、それをいずれかの CPU に通知する I/O APIC がある。

Asianux（アジアナックス）

日本のミラクル・リナックス（MIRACLE LINUX CORPORATION）、中国の北京中科紅旗軟件技術有限公司（Red Flag Software Co., Ltd.）、韓国のハーンソフト（HAANSOFT,Inc.）および上記 3 社の合弁会社 Asianux Corporation の 4 社共同開発による Linux OS、およびその開発プロジェクトの名称。

Asianux Server 3

Asianux プロジェクトで開発されているサーバ向け Linux ディストリビューション。kdump、ライブパッチ（KAHO）、kprobes、jprobes 等、本書で紹介する機能の多くを開発、搭載し、カーネル / ユーザランドの機能をサーバ向けに最適化

している。MIRACLE LINUX 4.0 の後継となり、MIRACLE LINUX V5 ともいう。

CentOS

RHEL との完全互換を目指した無償の Linux ディストリビューション。Red Hat 社が無償公開したソースコードより、同社の商標、商用パッケージなどを含まない形でリビルドされている。

crash

クラッシュダンプを解析するためのコマンドツール。動作中のカーネルを解析することも可能だが、カーネルの実行を停止させたりはできないため、デバッガと言えるまでの機能はない。本書では取り上げていないが、Linux でカーネルデバッガと言えば、kgdb や KDB が挙げられる。

Debian GNU/Linux

Debian Project により開発されている非商用の Linux ディストリビューション。コミュニティで開発された豊富なパッケージがある。パッケージ管理システム、設定方法等、RedHat 系ディストリビューションとは多くの部分で異なる。Debian 社会契約（Debian Social Contract）とそれに含まれる、Debian フリーソフトウェアガイドライン（DFSG）の遵守を重視している。

diskdump

カーネルダンプを採取する機能。RHEL4 など、RedHat 系の一部のディストリビューションで採用されている。詳細は**「diskdump を使ってカーネルクラッシュダンプを採取する」[HACK #19]** を参照。

Fedora

Red Hat 社が支援するコミュニティ「Fedora Project」によって開発されている、無償の Linux ディストリビューション。コミュニティにより開発が行われ、その成果が RHEL に取り込まれ、開発、検証目的としての位置づけがなされている。リリース 6 までは Fedora Core、リリース 7 以降は単に Fedora の呼称が使用されている。

GDB、gdb

GNU/Linux システムにおける標準デバッガ。プログラムの名称を指す場合は GDB、コマンド名を指す場合は gdb と表記されることが多い。GDB を使うと、プログラムの実行を任意の箇所で中断して、変数値の検査や、コールシーケンスの確認など、デバッグに必要な操作を行うことができる。本書では、GDB の使い方について詳しく説明している。

git

ソースコード管理ツールのひとつ。Linux カーネルのソース管理に使用されている。本書では、単にツールだけではなく、git で作られたリポジトリを指している場合もある。

i386

Intel の 32 ビットアーキテクチャの通称。このアーキテクチャが最初に搭載された CPU の名称に由来する。Intel の 32 ビットプロセッサのアーキテクチャは、その後、何度か拡張され、それらは、i486、i586、i686 と呼ばれている。これらの総称を i386 と呼ぶこともある。

IPMI watchdog

システムのフリーズ（ストール）を検出する機能。フリーズの検知時、カーネルダンプを取得して、リブートすることができる。また、それらの処理すら実行できないような致命的な状況でも、H/W による強制リセットや電源オフ等を行うことができる。ただし、フリーズの検知には専用の H/W が必要であり、それは一般的にミドルクラス以上のサーバ機にしか搭載されていない。詳細は、**「IPMI watchdog timer により、フリーズ時にクラッシュダンプを取得する」[HACK #23]** を参照。

kdump

カーネルダンプを採取する機能。RHEL5 など、バージョン 2.6.13 以降の Linux カーネルを採用したディストリビューションで利用できる。詳細は**「Kdump を使ってカーネルクラッシュダンプを採取する」[HACK #20]** を参照。

kill [-9]

シグナルをプロセスに送信するコマンド。オプション「-9」を指定すると、KILL シグナルが送信され、プロセスは OS により強制的に終了される。なお、オプションを指定しない場合、TERM シグナルが送信される。この場合、一般的には、プロセスが自発的に終了のための処理を実行するが、プロセスがストールしている場合には、その処理が実行されず、終了しないことがある。

NMI watchdog

システムのフリーズ（ストール）を検出する機能。フリーズの検出時、カーネルダンプを取得して、リブートを行うことができる。この機能は、比較的多くの PC で利用可能であるが、IPMI watchdog と違って、致命的な状況で H/W による強制リセットを実行することはできない。詳細は、**「NMI watchdog timer によ**

り、フリーズ時にクラッシュダンプを取得する」[HACK #22] を参照。

objdump

Linux カーネル（vmlinux）や実行ファイルの情報を得るためのコマンド。ファイル中のヘッダやセクションの表示、コードの逆アセンブルやデータのダンプをすることができる。詳しくは「objdump の便利なオプション」[HACK #44] を参照。

OOM Killer（Out of Memory Killer）

OS が必要なメモリ領域を新たに確保できない場合に，プロセスを強制終了させて空きメモリを確保する、Linux カーネルの仕組み。空きメモリ不足により OS 自体が停止するという最悪の事態を避けるために用意されている。

ps

プロセスの一覧を表示するコマンド。オプションにより、実行ユーザ、プロセスグループ ID、セッション ID、PID、親 PID、スレッド ID、TTY、メモリ使用量、実行状態、開始時刻、実行時間、優先度、コマンドラインオプションなど詳細な情報を得ることができる。

RHEL

Red Hat, Inc.（レッドハット社）によって開発、販売されている有償の企業向け Linux ディストリビューション、Red Hat Enterprise Linux の略。Update の適用された版として RHEL4.7（旧称 Red Hat Enterprise Linux 4 Update 7）、RHEL5.2 等と略す。パッケージ管理システムに RPM（RPM Package Manager、以前の名称は Red Hat Package Manager）を採用している。

SIGSEGV、セグメンテーションフォルト

メモリの不正アクセスを意味する。コードが存在しないアドレスにジャンプしたり、書き込み不可な領域に書き込みを行った際、OS からシグナルとして通知される。対応するシグナルハンドラがない場合、そのままプロセスは終了する。バグによって発生する典型的な現象のひとつ。[HACK #26] を参照。

SMP

Symmetric Multi Processing の略で、複数の CPU コアが搭載され、どの CPU コアでも同じ処理が実行できるシステム。複数の処理が同時に実行されるため、UP に比べ、ソフトウェアで考慮すべき事項が多く、SMP であることに起因したバグもしばしば発生する。

softdog

システムのフリーズ（ストール）を検出する機能。フリーズの検出時、カーネルダンプを取得して、リブートを行うことができる。IPMI watchdog や NMI watchdog と異なり、すべての PC で使用することができるが、フリーズの要因によっては、機能しないこともある。

strace

プロセスのシステムコールをトレースするコマンド。システムコールでエラーになるような不具合の調査や、不具合箇所の切り分けに威力を発揮する。詳しくは、**「strace を使って、不具合原因の手がかりを見つける」[HACK #43]** を参照。

syslog

カーネルやプロセスからのメッセージが記録されるファイル。このファイルの名称は、ディストリビューションごとに差異があり、RedHat 系では、/var/log/messages であることが多く、Debian 系では、/var/log/syslog であることが多い。不具合の発生時、このファイルに何らかの情報が記載されていることも多い。

SysRq キー

101（およびその互換）キーボードでは「 Alt + PrintScreen 」が該当する。Linux での使い方の詳細は**「SysRq キーによるデバッグ方法」[HACK #18]** 参照。

top

プロセスを CPU 負荷順にソートし、表示するコマンド。CPU 負荷の他にも、メモリ使用量、優先度、実行時間や PID なども同時に表示されるため、プロセスの状態を簡単に知ることができる。

UP

Uni Processing の略で、CPU コアが 1 つだけのシステム。

VMware

VMware Workstation、VMware Server、VMware Fusion は、各ハードウェアで動作するホスト OS 上で仮想マシンを作成、実行する仮想マシンソフトウェアである。VMware ESX 、ESXi はハイパーバイザタイプの仮想マシンソフトウェア。ホスト OS が存在せず、VMkernel と呼ばれるソフトウェアが直接ハードウェア上（RING0）で動作し仮想マシン環境を構成する。

x86_64

AMD の 64 ビットアーキテクチャ、およびその互換である Intel の 64 ビットアー

キテクチャの通称。Intel の 64 ビットアーキテクチャには、ia64 と呼ばれるものもあるが、それとは別物。x86_64 アーキテクチャの特徴のひとつは、i386 の命令も実行できることであり、そのため、i386 用のソフトウェア資産を活用しながら、64 ビット環境を利用できる。

Xen

ハイパーバイザタイプの仮想マシンソフトウェアで RING0 で動作する。Xen では、仮想マシンの実行単位を Domain（ドメイン）と呼ぶ。Domain には、実ハードウェアへのアクセスやその他のドメインを管理する特権的な Domain の Domain 0、通常の仮想マシンとして Domain U や HVM Domain がある。

アセンブリ言語

マシン語を、人が理解しやすい形式で記述する低級言語。アセンブリ言語のプログラムはニーモニックと呼ばれる命令列で記述される。ニーモニックはマシン語と一対一で対応する表記形式であり、CPU アーキテクチャごとに異なる。特定用途向けに最適化されたプログラムなどでは、いまだにアセンブリ言語を用いて記述されているものもある。Linux カーネルにおいても例外処理やプロセッサモードの切り替え処理など、多くのアセンブリ言語が含まれている。本書では、アセンブラ、アセンブリコード、アセンブラコードと表記している Hack もある。

アタッチ、デタッチ

デバッガなどを使って動作中のプロセスに接続することをアタッチと言い、切断することをデタッチと言う。本書では、GDB や strace を使ってプロセスにアタッチするデバッグ方法を紹介している。

カーネル

OS の中核部分のプログラムのこと。本書では Linux カーネルを指す。

カーネルコンフィグ

カーネルの機能追加や不要な機能の削除、あるいはカーネルの組み込みパラメータ設定を行うカーネルコンフィグレーションのこと。デフォルトで無効化されているデバッグ機能を使いたい場合は、カーネルコンフィグを変更する必要がある。**「カーネルパニック（リスト破壊編）」[HACK #34]**、**「フォルト・インジェクション」[HACK #57]** にて事例が紹介されている。

カーネルダンプ、クラッシュダンプ

ある時点のカーネルのメモリイメージとレジスタの内容などをファイルに保存すること、あるいは保存されたファイル自体を表す。保存するメモリイメージはカー

ネルメモリ、全メモリなどいくつかの条件から選択できる実装が多い。一般にカーネルに異常が発生した時に作成される。crash などの解析ツールとクラッシュダンプを利用すれば、問題発生時のカーネルの状態を知ることができる。

カーネルパラメータ、カーネルブートパラメータ

GRUB、LILO などブートローダから Linux カーネルを起動する際に指定する。再コンパイルせずにカーネルの動作、設定を変更するためのパラメータ。

カーネルメッセージ

カーネルが出力するメッセージ。内容は、デバッグ情報から、致命的なエラーを伝えるものまで多岐にわたる。内容の重要度にもよるがコンソールや syslog に出力される。また、dmesg コマンドでも確認できる。

逆アセンブラ

マシン語を、人が理解しやすいアセンブリ言語へ変換するソフトウェアのこと。Linux では objdump コマンドが代表的。一般にデバッガも逆アセンブラ機能を持っている。逆アセンブラは本書のいたるところで使用しており、デバッグには必須のアイテムである。

クラッシュ

ユーザアプリケーションや OS が突然終了すること。ソフトウェアバグが原因であることが多いが、ハードウェアの不具合により発生することもある。ソフトウェアバグの場合、通常は、ユーザアプリケーションであればコアダンプ、OS であればクラッシュダンプを採取してデバッグすることになる。

コアダンプ

ある時点のプロセスのメモリイメージとレジスタの内容などをファイルに保存すること、あるいは保存されたファイル自体を表す。保存されたファイルはコアファイルとも言う。一般にプログラムが異常終了した時に、OS によってコアダンプが作成される。GDB などのデバッガとコアダンプを利用すれば、問題発生時のプロセスの状態を知ることができる。詳しくは「**プロセスのコアダンプを採取する**」**[HACK #4]** を参照。

スケジューラ

マルチタスク OS でプロセスやスレッドなど、処理の実行単位に CPU 実行権を割り当てる機能のこと。Linux ではタスクスケジューラと呼ばれることもある。スケジューラはシステムの応答性に大きな影響を及ぼすため、OS 機能の中核と言える。スケジューラにバグがあった場合、プログラムが終了しなかったり、シ

ステムがハングしたり、クラッシュしたりなど致命的な障害になる傾向が高く、かつデバッグにも非常に苦労する。

スタックオーバーフロー

スタック領域のサイズよりも多くのデータをスタックに格納すること。スタックオーバーフローが発生すると、プロセスに SIGSEGV が配信される。再帰処理を行うプログラムで発生しやすい現象。

ストール、ハング

システムあるいはプロセスが応答しなくなる状態を示す。ハングアップやフリーズ、「スタックする」と表現されるときもある。日本語では「固まる」、「ささる」とも言われる。

ストックカーネル（Stock kernel）

Linus Torvals 氏がリリースする標準となる Linux カーネルのこと。main line、vanilla kernel ともいう。

スピンロック

ロック機構のひとつ。Linux カーネルで最も多用されているロックで、ロックにかかる処理コストが低いため、クリティカルセクションが短い場合に使用される傾向がある。スピンロックでデッドロックが起きる場合、NMI watchdog や IPMI watchdog でクラッシュダンプが取れる場合が多い。詳しくは**「カーネルのストール（スピンロック編その 1）」[HACK #37]**、**「カーネルのストール（スピンロック編その 2）」[HACK #38]** を参照。

スレッド、プロセス

Linux において、スレッドは、カーネルが管理する最小単位の実行中のプログラムである。プロセスは、1 つまたは複数のスレッドから構成される実行中のプログラムである。プロセス内の各スレッドは、メモリ空間を共有しているため、1 つのスレッドによるメモリ不正アクセスなどのバグが、他のスレッドの動作に影響が及ぶことがある。

セクション

ELF や COFF などのオブジェクトファイルフォーマットの一部。本書では、ELF フォーマットを対象としている。ELF では、プログラムの読み取り専用データは .rodata セクション、初期値なしのデータは .bss セクション、実行コードは .text セクションというように、その役割に応じたセクションにコンパイラあるいはリンカにより配置される。本書では、**「配列の不正アクセスによるメモリ内容の破壊」**

[HACK #28] と「Linux カーネルの init セクション」**[HACK #59]** で、セクションに関連したデバッグ事例を紹介している。

セマフォ

ロック機構のひとつ。Linux のユーザアプリケーションが利用できるものは Posix セマフォと IPC セマフォである。Linux カーネルが内部で使用するセマフォもあり、カーネルセマフォあるいは単にセマフォと呼ばれる。カーネルセマフォでデッドロックが起きる場合、SysRq キーでクラッシュダンプが取れる場合が多い。詳しくは「**カーネルのストール（セマフォ編）**」**[HACK #39]** を参照。

タスク

本書では、プロセス、スレッドの他、一般的な処理の意味で用いる。

デッドロック

排他処理の欠陥のひとつで、永遠にロック獲得待ちに陥ってしまう状態を表す。同じロックを複数回ロックしようとする場合や、ロックする順番を誤ったために複数のスレッドで互いのロックを待ち続けてしまう場合、ロックを獲得したままイベント待ちに陥ってしまう場合などがある。これはストールの代表的な原因のひとつである。

バッファオーバーラン

バッファの領域外にデータを書き込むバグ。C 言語で配列を使う場合に、しばしば混入する。バッファ領域のすぐ外側が、他のデータ領域の場合、それらのデータを破壊するので、プログラムにおかしな挙動を引き起こすことがある。また、バッファのすぐ外側にメモリがマップされていない場合、セグメンテーションフォルトを引き起こす。

パニック、カーネルパニック

カーネル内で致命的なエラーが発生し、OS の処理が完全に停止すること。Linux であれば Oops などのエラーメッセージを表示して停止する。パニックが発生すると再起動するしかなくなるが、クラッシュダンプの設定をしていれば、ダンプ採取が行われる。本書では、このエラーメッセージやクラッシュダンプからの解析事例を扱っている。

パリティエラー

ノイズなどハードウェア要因により、メモリ内の 1 ビットが不正に書き換えられる障害。この障害は、原理的にすべての PC で起こり得るが、検出には、それに対応したメモリコントローラなどのハードウェアが搭載されている必要がある。

プロセス / タスクの状態

Linux の主なプロセスの状態には、RUNNING、INTERRUPTIBLE、UNINTERRUPTIBLE、STOPPED、TRACED、ZOMBIE がある。RUNNING はプログラムを実行している状態。INTERRUPTIBLE は中断可能なスリープ状態。UNINTERRUPTIBLE は中断不可能なスリープ状態。STOPPED は停止状態。TRACED はデバッグされている状態。ZOMBIE はプロセスが終了した後、親プロセスにその終了ステータスが読み取られるのを待っている状態。

マシン語

CPU が直接理解できる命令のことで機械語とも呼ばれる。実体は 0 と 1 の 2 値で表されるバイナリ列だが、プロセッサマニュアル等では 16 進数で表記されることが多い。人が直接マシン語を扱うことは稀で、通常はマシン語と一対一で対応するアセンブリ言語、あるいは C 言語などの高級言語を扱いプログラミングやデバッグを行う。

無限ループ

プログラム内で、一連の処理が無限に繰り返される状態を表す。ただし本書では、プログラマが意図しない場合、つまりバグによって無限にループしている状態を指している。単純なデータの取り扱いミスから、複雑な排他制御の考慮もれまで、原因は多岐にわたる。これはストールの代表的な原因のひとつである。詳しくは**「アプリケーションのストール（無限ループ編）」[HACK #32]**、**「カーネルのストール（無限ループ編）」[HACK #36]** を参照。

ユーザモード、カーネルモード

CPU の実行モードのことで、特権レベルとも言われる。CPU は少なくとも 2 つ以上の実行モードがあり、その実行モードに応じてアクセスできるアドレス空間や CPU 命令が制限される。カーネルモードは特権モードであり、すべてのアドレス空間と CPU 命令が実行可能。ユーザモードは非特権モードであり、カーネル空間へのアクセスや、CPU の特権命令は実行できない。Linux も含め一般の OS では、カーネルのコードはカーネルモードで動作し、ユーザアプリケーションのコードはユーザモードで動作する。

ユーザ空間、カーネル空間

仮想記憶方式の OS で利用される仮想アドレス空間の分類のこと。Linux や Windows など、一般に利用される OS は仮想記憶方式である。ユーザアプリケーションがアクセスできるアドレス空間をユーザ空間あるいはユーザランドと言い、カーネルからのみアクセスできるアドレス空間をカーネル空間あるいはカーネルランドと言う。

リアルタイムプロセス

通常のプロセスより高い優先度を持ち、自発的に CPU を手放すか、より高い優先度を持つリアルタイムプロセスにプリエンプトされるまで、CPU を占有し続けるプロセス。リアルタイムプロセスが暴走するとシステムがストールするなどの障害が発生する場合がある。詳しくは、**「リアルタイムプロセスのストール」[HACK #40]** や **「RT Watchdog を使ってリアルタイムプロセスのストールを検知する」[HACK #65]** 参照。

例外

CPU が実行中に異常や特殊な状態を検出すること。例えば、0 での除算、アクセス権のないメモリの参照、無効な命令の実行、デバッグ命令の実行などにより例外が発生する。

レースコンディション、競合

複数のスレッドあるいはプロセスが共有リソースに対して何らかの処理を行う際、タイミングによって予期しない結果となる状態のこと。この手の問題は複雑な場合が多く、関係する共有リソースが多ければ多いほど原因の特定が難しくなる。排他もれやデータアクセスの順序の不整合によるものが多い。

ロック

排他制御に使用される同期機構のひとつ。複数のスレッド間でリソースを共有する際にロックを課すことでデータの一貫性を保つことができる。ロックに守られた処理区間をクリティカルセクションと呼ぶ。ユーザアプリケーションであれば mutex、セマフォなどが代表例である。Linux カーネルであればスピンロック、セマフォ、RCU、シーケンシャルロックなどが挙げられる。ロックもれによるデータ破壊、デッドロックなどバグが入り込みやすいのが特徴である。ロック以外の同期機構では、アトミック命令でリソースを操作するという方法もある。

割り込み

主に周辺機器からの非同期な通知を CPU に伝えるための機構。Linux カーネルを含め、多くの OS では、割り込みが発生すると実行中の処理を中断して、その通知への対応処理（割り込みハンドラ）を実行する。ソフトウェアで、このような中断が任意のタイミングで発生することが考慮されていないと、バグが混入する可能性がある。

まとめ

Debug Hacks に登場する用語を紹介しました。それぞれの用語は本文中でも必要に応

じて解説していきます。ここで紹介した以外にも『BINARY HACKS』の「**Binary Hack 用語の基礎知識**」**[HACK #2]** が参考になります。

参考文献

- 『BINARY HACKS —— ハッカー秘伝のテクニック 100 選』（オライリー・ジャパン刊）

—— Toyo Abe、Kazuhiro Yamato、Shunsuke Yoshida

索　引

数字・記号

A

B

C

D

E

F

G

H

I

O

P

Q

R

S

か行

さ行

た行

な行

は行

ま行

や行

ら行

わ行

● ミラクル・リナックス株式会社について

ミラクル・リナックスは、Linux サーバ関連製品とサービス専門事業会社として 2000 年 6 月 1 日より業務を開始し、Linux サーバ OS「MIRACLE LINUX」および「Asianux Server3 ==MIRACLE LINUX V5」の開発および販売、24 時間 365 日連続稼動を実現するクラスタソリューション「MIRACLE CLUSTERPRO X」の提供、さらに Linux 関連のコンサルティング、教育、保守等のサポート・サービスの提供など、幅広く事業を展開しています。また、2003 年 12 月には、「4-co（4 つの共同）」をコンセプトにもつ、アジアから発信する全く新しいソフトウェアビジネスプロジェクト「Asianux®（アジアナックス）」を立ち上げ、「アジア市場に最適化し信頼性の高い共通のエンタープライズ Linux ディストリビューション」の共同開発に取り組んでいます。

ミラクル・リナックス株式会社 URL：http://www.miraclelinux.com/

● カバーの説明

表紙の絵は蚊遣り豚（かやりぶた）です。「デバッグ」は虫を退治するという意味ですので、日本の伝統的な虫退治の手法、蚊取線香を中に吊り下げて使う蚊遣り豚を表紙に選びました。三重県四日市市の萬古焼（ばんこやき）のものが有名で、江戸時代後期に登場し、現在でも夏の風物詩のひとつとして根強い人気があります。

Debug Hacks

—— デバッグを極めるテクニック & ツール

2009 年 4 月 22 日　初版第 1 刷発行
2011 年 6 月 6 日　初版第 5 刷発行

著　　者　吉岡 弘隆（よしおか ひろたか）、大和 一洋（やまと かずひろ）
　　大岩 尚宏（おおいわ なおひろ）、安部 東洋（あべ とうよう）
　　吉田 俊輔（よしだ しゅんすけ）
発 行 人　ティム・オライリー
印刷・製本　株式会社平河工業社
発 行 所　株式会社オライリー・ジャパン
　　〒 160-0002　東京都新宿区坂町 26 番地 27　インテリジェントプラザビル 1F
　　Tel　(03)3356-5227
　　Fax　(03)3356-5263
　　電子メール　japan@oreilly.co.jp
発 売 元　株式会社**オーム**社
　　〒 101-8460　東京都千代田区神田錦町 3-1
　　Tel　(03)3233-0641（代表）
　　Fax　(03)3233-3440

Printed in Japan（ISBN978-4-87311-404-0）
乱丁、落丁の際はお取り替えいたします。