Robert Love

Jądro Linuksa

Przewodnik programisty

Wnieś swój wkład w rozwój Linuksa!

Helion

Tytuł oryginału: Linux Kernel Development (3rd Edition)

Tłumaczenie: Przemysław Szeremiota

ISBN: 978-83-246-4273-1

Authorized translation from the English edition, entitled: LINUX KERNEL DEVELOPMENT, Third Edition; ISBN 0672329468; by Robert Love; published by Pearson Education, Inc, publishing as Addison Wesley.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/jadlin
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

18

Diagnostyka błędów jądra

Jedną z podstawowych miar odróżniających programowanie jądra od programowania przestrzeni użytkownika jest stopień trudności diagnostyki błędów. Diagnostyka jądra nie jest łatwa, zwłaszcza w porównaniu z przestrzenią użytkownika. Problem komplikuje jeszcze wysokość stawki — błąd w jądrze może załamać cały system komputerowy.

Rosnąca umiejętność diagnozowania błędów w jądrze — a co za tym idzie, coraz większa swoboda programowania jądra — jest w dużej mierze funkcją doświadczenia i stopnia zrozumienia systemu operacyjnego. Podziałać mogą zapewne również zaklęcia i czary, ale do udanej diagnostyki jądra konieczna jest jego pełna znajomość. Niniejszy rozdział poświęcony będzie właśnie zagadnieniom związanym z diagnostyką błędów jądra.

Od czego zacząć?

Diagnostyka błędów w jądrze bywa uciążliwym i żmudnym procesem. Niektóre z błędów całymi miesiącami myliły tropy, choć śledzone były przez wielu programistów. Na szczęście na dosłownie każdy tak pracochłonny błąd przypada wiele błędów bardziej oczywistych, z równie oczywistymi poprawkami. Przy odrobinie szczęścia wszystkie błędy, jakie popełnimy, będą błędami prostymi. Tego jednak przed rozpoczęciem poszukiwań nie będzie wiadomo na pewno. Dlatego warto zaopatrzyć się w:

- Błąd. To może brzmi śmiesznie, ale potrzebny będzie dobrze zdefiniowany i bardzo konkretny błąd. Dobrze, jeżeli jest to błąd powtarzalny, który można sprowokować w określonych warunkach. Niestety, błędy rzadko bywają stabilne i powtarzalne.
- Wersję jądra, w której występuje rzeczony błąd (na przykład najnowszą wersję, bo inaczej nikt nie zawracałby sobie głowy błędem). Najlepiej, jeżeli uda się wytypować najwcześniejszą wersję jądra, w której pojawił się błąd. W rozdziale będzie omawiana technika polowania na popsutą wersję jądra.
- Znajomość odpowiedniej części jądra albo szczęście. Diagnozowanie błędów jądra jest śledztwem, które daje tym lepsze wyniki, im lepsza jest znajomość otoczenia.

Większość bieżącego rozdziału będzie poświęcona technikom reprodukcji błędów. Skuteczność w ich poprawianiu jest bowiem wypadkową umiejętności prowokowania wystąpienia błędu. Niemożność sprowokowania błędu ogranicza możliwości jego usunięcia, zmuszając do jedynie koncepcyjnego określenia problemu i odnalezienia odpowiadającej mu luki w kodzie źródłowym. Często zdarza się zresztą i tak, ale szansa powodzenia jest znacznie większa przy możliwości obserwacji błędu.

Istnienie błędu, którego wpływu na system nie da się ujawnić, może być nieco wątpliwe. W programach przestrzeni użytkownika błędy są najczęściej bardziej dokuczliwe — na przykład „wywołanie funkcji `bla()` w takim a takim kroku pętli powoduje awaryjne zamknięcie programu". W kodzie jądra sprawy są zazwyczaj znacznie bardziej zagmatwane. Interakcje pomiędzy jądrem, przestrzenią użytkownika i sprzętem potrafią być bardzo delikatne. Niekiedy będące przyczyną błędu sytuacje hazardowe pojawiają się raz na milion iteracji jakiegoś algorytmu. Słabo zaprojektowany albo nawet źle skompilowany kod może dawać w niektórych systemach oczekiwane rezultaty, pogarszając wydajność systemów o innych konfiguracjach. Niejednokrotnie zdarza się wywołać błąd jedynie w określonej konfiguracji jądra, na pewnej przypadkowej maszynie, przy obciążeniu systemu nietypowymi operacjami itd. Dlatego kluczowe znaczenie ma ilość informacji o środowisku, w jakim wystąpił błąd. Zazwyczaj jednak możliwość powtórnego sprowokowania błędu to więcej niż połowa sukcesu.

Błędy w jądrze

Błędy w jądrze bywają bardzo różnorodne. Zdarzają się one z niezliczonych powodów i objawiają się w równie niezliczonych formach. Błędy od jawnie błędnego kodu (na przykład niezachowania poprawnej wartości w odpowiednim miejscu), przez błędy synchronizacji (wynikające z braku właściwego blokowania dostępu do współdzielonej zmiennej), po nieprawidłowe zarządzanie sprzętem (wysyłanie nieprawidłowej wartości do nieodpowiedniego rejestru sterującego) objawiają się pod wszelkimi postaciami, od niezadowalającej wydajności systemu po jego niewłaściwe działanie w skrajnych przypadkach powodujące utratę danych albo zawieszanie systemu.

Błąd w postaci objawiającej się użytkownikowi nieraz może dzielić od jego przyczyny tkwiącej w kodzie jądra długi łańcuch zdarzeń. Na przykład współużytkowana struktura pozbawiona licznika odwołań może prowokować sytuacje hazardowe. Wobec braku poprawnego zliczania odwołań może dojść do zwolnienia przez jeden proces struktury używanej jeszcze przez inny proces. Ów inny proces może próbować odwołać się do zwolnionej już struktury za pośrednictwem przetrzymywanego lokalnie, nieprawidłowego już wskaźnika. Może to zaowocować wyłuskaniem wskaźnika pustego, odczytem „śmieci" czy też czymś zupełnie niewinnym (kiedy obszar zwolnionej struktury nie został jeszcze niczym nadpisany). Wyłuskanie wskaźnika pustego daje w efekcie błąd „oops", podczas gdy odwołanie do przypadkowych danych („śmieci") w pamięci może prowadzić do naruszenia spójności danych (a w jej wyniku do niewłaściwego zachowania

programu, a w dalekiej konsekwencji do błędu „oops”). Użytkownik zgłasza więc albo błąd „oops”, albo wyłącznie niepoprawne działanie systemu. Programista jądra musi wyśledzić przyczynę błędu, wykryć, że nastąpił dostęp do zwolnionych wcześniej danych, że wcześniej niewłaściwie współużytkowana była zawierająca te dane struktura i wreszcie zaaplikować łatę w postaci poprawnego zliczania odwołań do współużytkowanej struktury (i być może jakiegoś rygla chroniącego ją przed współbieżnym dostępem).

Diagnostyka błędów jądra może sprawiać wrażenie zadania dla magików, ale w rzeczywistości jądro systemu nie różni się niczym od innych dużych projektów programowych. Co prawda w jądrze należy brać pod uwagę elementy specyficzne, takie jak ograniczenia czasowe wykonania kodu czy sytuacje hazardowe będące konsekwencją działania wielu wątków w ramach jądra.

Diagnostyka komunikatami

Funkcja jądra `printk()` działa niemal identycznie jak podobnie nazwana funkcja biblioteczna języka C, `printf()`. Podobieństwo działania jest tak duże, że jak na razie w książce nie wystąpiło zastosowanie funkcji `printk()` znacząco odbiegające od zastosowań `printf()`. Tak więc `printk()` to nazwa funkcji jądra generującej sformatowany wydruk znakowy. Nie jest to jednak tak do końca zwykła funkcja formatująca.

Niezawodność printk()

Jedną z tych właściwości wywołania `printk()`, które szybko przyjmuje się za oczywistość, jest jej *niezawodność* i *wszechstronność*. Funkcja `printk()` daje się wywołać z dowolnego miejsca jądra w dowolnym momencie jego wykonania. Można ją wywołać z kontekstu procesu i kontekstu przerwania. Można ją wywoływać, przetrzymując równocześnie blokadę dowolnego rodzaju. Można też wywoływać ją współbieżnie na wielu procesorach, przy czym wywołujący nie musi pozyskiwać żadnej blokady.

To funkcja naprawdę nie do zdarcia. To bardzo ważne, ponieważ przydatność funkcji `printk()` wynika w dużej mierze właśnie z możliwości jej wywołania dosłownie zewsząd i z gwarancji jej działania w każdych warunkach.

Istnieje jednak luka we wszechstronności wywołania `printk()`. Nie da się jej bowiem zastosować przed określonym momentem rozruchu jądra, a ściśle mówiąc, przed momentem zainicjowania konsoli. To oczywiste, bo gdzie niby miałyby być wcześniej kierowane komunikaty? Zwykle nie jest to problemem, chyba że chodzi o diagnozowanie bardzo wczesnej fazy rozruchu (na przykład diagnostykę błędów działania funkcji `setup_arch()` odpowiedzialnej za operacje inicjalizacji zależne od architektury systemu). Taka diagnostyka to prawdziwe wyzwanie — a brak jakichkolwiek narzędzi wyprowadzania komunikatów dodatkowo je komplikuje.

Nawet diagnostyka wczesnych faz rozruchu daje jednak pewne możliwości. Niektórzy programiści wykorzystują do wyprowadzania komunikatów sprzęt, który działa zawsze,

na przykład port szeregowy. Nie jest to jednak przyjemna zabawa. Rozwiązaniem jest wariant wywołania `printk()` przystosowany do wyprowadzania komunikatów na konsolę we wczesnych fazach rozruchu — `early_printk()`. Działa ona identycznie jak `printk()` — funkcje te różnią się wyłącznie nazwami i możliwością operowania w pierwszych fazach rozruchu. Nie jest to jednak rozwiązanie przenośne, gdyż funkcja `early_printk()` nie jest implementowana we wszystkich architekturach. Dobrze, żeby diagnozowana architektura ją posiadała — a większość (z architekturą x86 na czele) posiada.

Podsumowując, o ile nie zachodzi potrzeba sygnalizacji na bardzo wczesnym etapie rozruchu, można polegać na wszechstronności i niezawodności funkcji `printk()`.

Poziomy diagnostyczne

Główna różnica pomiędzy wywołaniami `printk()` i `printf()` tkwi w zdolności tej pierwszej do określania tak zwanego *poziomu diagnostycznego* (ang. *loglevel*). Jądro określa poziom diagnostyczny, decydując tym samym, czy komunikat powinien zostać wyświetlony na konsoli czy też przekierowany gdzieś indziej. Jądro wyświetla na konsoli wszystkie komunikaty o poziomie diagnostycznym ustalonym na poziomie niższym od pewnego progu.

Poziom diagnostyczny określa się następująco:

```
printk(KERN_WARNING "To jest ostrzezenie!\n");
printk(KERN_DEBUG "To jest komunikat diagnostyczny!\n");
printk("Brak okreslenia poziomu diagnostycznego!\n");
```

Ciągi `KERN_WARNING` i `KERN_DEBUG` to po prostu definicje występujące w pliku *<linux/printk.h>*. Rozwijane są one odpowiednio do ciągów `<4>` i `<7>` i dołączane na początek komunikatu przekazywanego do `printk()`. Jądro na podstawie tych przedrostków decyduje o tym, które z komunikatów powinny być wyświetlane na konsoli, porównując określony poziom diagnostyczny z bieżącym poziomem diagnostycznym konsoli określonym parametrem `console_loglevel`. Pełny wykaz dostępnych poziomów diagnostycznych funkcji `printk()` został przedstawiony w tabeli 18.1.

Tabela 18.1. Poziomy diagnostyczne funkcji printk()

Poziom diagnostyczny	Opis
`KERN_EMERG`	Sytuacja awaryjna.
`KERN_ALERT`	Problem wymagający natychmiastowej interwencji.
`KERN_CRIT`	Sytuacja krytyczna.
`KERN_ERR`	Błąd.
`KERN_WARNING`	Ostrzeżenie.
`KERN_NOTICE`	Sytuacja normalna, ale warta odnotowania.
`KERN_INFO`	Komunikat informacyjny.
`KERN_DEBUG`	Komunikat diagnostyczny — najprawdopodobniej zbyteczny.

W przypadku nieokreślenia poziomu rejestrowania jego wartość jest przyjmowana przez domniemanie jako DEFAULT_MESSAGE_LOGLEVEL, która to stała jest z kolei rozwijana do stałej KERN_WARNING. Jednak z uwagi na to, że ustawienie domyślne może w kolejnych wersjach jądra ulec zmianie, warto jawnie opatrywać komunikaty stałymi poziomów rejestrowania.

Najwyższy poziom rejestrowania definiuje stała KERN_EMERG rozwijana do literału znakowego <0>. Poziom najniższy określa zaś stała KERN_DEBUG rozwijana do ciągu <7>. Na przykład po zakończenia fazy przetwarzania pliku kodu źródłowego przez preprocesor zamieszczone wcześniej wywołania będą miały postać:

```
printk("<4>To jest ostrzezenie!\n");
printk("<7>To jest komunikat diagnostyczny!\n");
printk("<4>Brak okreslenia poziomu diagnostycznego!\n");
```

Strategia przyjmowania poziomów rejestrowania w różnych sytuacjach leży całkowicie w gestii programisty. Rzecz jasna, zwykłe komunikaty należałoby opatrywać odpowiednim dla nich poziomem rejestrowania. Ale już komunikaty, którymi szpikuje się kod w poszukiwaniu źródła problemu — wypada się do tego przyznać, wszyscy tak robią — można opatrywać dowolnym poziomem rejestrowania. Jedną z możliwości jest pozostawienie w spokoju domyślnego poziomu rejestrowania konsoli i generowania wszystkich komunikatów diagnostycznych z poziomem KERN_CRIT lub wyższym. Można też generować komunikaty na poziomie KERN_DEBUG i w zamian zmodyfikować poziom rejestrowania konsoli, tak aby komunikaty były widoczne. Każda z metod ma swoje wady i zalety. Decyzję pozostawiam Czytelnikowi.

Bufor komunikatów

Komunikaty jądra umieszczane są w cyklicznym buforze o rozmiarze LOG_BUF_LEN. Rozmiar ten daje się konfigurować na etapie kompilacji przez ustawienie opcji CONFIG_LOG_BUF_SHIFT. Domyślny rozmiar bufora dla komputerów jednoprocesorowych to 16 kB. Innymi słowy, jądro może przechowywać do 16 kB komunikatów jądra. Jeżeli kolejka komunikatów jest zapełniona i dojdzie do kolejnego wywołania printk(), nowy komunikat nadpisze najstarszy z komunikatów w buforze. Bufor komunikatów nosi nazwę *cyklicznego*, ponieważ jego zapis i odczyt następują w sposób cykliczny.

Korzystanie z bufora cyklicznego ma szereg zalet. Z racji dużej łatwości równoczesnego zapisywania i odczytywania z takiego bufora funkcja printk() może być wywoływana nawet z poziomu kontekstu. Co więcej, cykliczność upraszcza zarządzanie buforem. W obliczu zbyt dużej liczby komunikatów najstarsze komunikaty są po prostu zastępowane nowymi. Jeżeli zdarzy się problem, w wyniku którego wygenerowana zostanie większa liczba komunikatów, cykliczność obsługi bufora wyeliminuje część z nich, nie dopuszczając do zwiększenia zapotrzebowania na pamięć. Jedyną wadą takiego rozwiązania jest prawdopodobieństwo utraty komunikatów, ale jest to niewielka cena za wszechstronność bufora.

Demony syslogd i klogd

W klasycznym systemie Linux za pobieranie komunikatów jądra z bufora odpowiedzialny jest demon przestrzeni użytkownika o nazwie `klogd` — demon ten wypełnia pobieranymi komunikatami systemowy plik dziennika, korzystając przy tym z pomocy demona `syslogd`. W celu odczytu rejestru komunikatów `klogd` może odczytywać plik */proc/kmsg* bądź korzystać z wywołania systemowego `syslog()`. Domyślnie wykorzystuje jednak system plików */proc*. Niezależnie od sposobu odczytywania komunikatów `klogd` ulega zawieszeniu aż do momentu pojawienia się nowych komunikatów. W obliczu zgłoszenia nowych komunikatów demon jest pobudzany, odczytuje nowe komunikaty i przetwarza je. Domyślnie przetwarzanie to polega na przekazaniu ich do demona `syslogd`.

Demon `syslogd` dołącza otrzymywane komunikaty do pliku, którym przez domniemanie jest plik */var/log/messages*. Działanie demona da się konfigurować za pośrednictwem pliku */etc/syslog.conf*.

Za pośrednictwem demona `klogd` można zmieniać poziom rejestrowania konsoli — wystarczy w wywołaniu demona określić argument `-c`.

printf(), printk() — łatwo o pomyłkę

Każdemu początkującemu programiście jądra zdarza się mylić wywołania `printk()` z wywołaniami `printf()`. To całkiem naturalne, ponieważ nie sposób zignorować lata doświadczeń i nawyku wykorzystywania funkcji `printf()` w programach przestrzeni użytkownika. Na szczęście takie pomyłki są szybko wychwytywane, jako że powodują przy kompilacji zalew komunikatami z protestami ze strony konsolidatora.

Pewnego dnia być może zdarzy się Czytelnikowi zastosować w wyniku wyniesionych z programowania jądra nawyków wywołanie `printk()` w programie przestrzeni użytkownika. Wtedy będzie on mógł o sobie powiedzieć, że jest prawdziwym hakerem jądra.

Błąd oops

Błąd *oops* to normalny sposób powiadamiania użytkownika o nieprawidłowościach działania jądra. Jako że jądro to nadzorca całego systemu, nie ma możliwości samodzielnego usunięcia usterki czy wykonania samounicestwienia, tak jak unicestwia się błędne procesy przestrzeni użytkownika. Jądro w takich sytuacjach zgłasza błąd oops. Polega to na wyświetleniu na konsoli komunikatu o błędzie wraz z zawartością rejestrów i śladem wykonania (ang. *back trace*). Błąd jądra jest trudny do obejścia, więc jądro w celu obsłużenia błędu musi się nieźle napracować. Niekiedy po zakończeniu obsługi pojawia się niespójność jądra. Jądro w momencie pojawienia się błędu mogło być na przykład w trakcie przetwarzania istotnych danych. Mogło przetrzymywać blokadę albo prowadzić komunikację ze sprzętem. W obliczu błędu konieczne jest natomiast ostrożne wycofanie się od poprzedniego kontekstu i w miarę możliwości przywrócenie kontroli nad systemem. W wielu przypadkach powrót taki jest niemożliwy. Jeżeli błąd wystąpi w kontekście przerwania, jądro nie może

kontynuować działania i „panikuje". Błąd „paniczny" (ang. *panic error*) powoduje zaś trwałe unieruchomienie systemu. Również pojawienie się błędu oops w trakcie wykonywania zadania jałowego (o numerze `pid` równym zero) bądź zadania `init` (`pid` równy jeden) oznacza błąd paniczny — jądro nie może kontynuować działania bez tych dwóch ważnych procesów. Jedynie wystąpienie błędu oops w kontekście jednego ze zwykłych procesów użytkownika daje możliwość unicestwienia tego procesu i kontynuowania działania reszty systemu.

Przyczyny błędu oops mogą być rozmaite, z nieuprawnionym dostępem do pamięci bądź próbą wykonania niedozwolonej instrukcji włącznie. Programista jądra jest skazany na diagnostykę takich błędów (nie mówiąc o tym, że sam przyczynia się do ich powstawania).

Poniższy wydruk to komunikat towarzyszący błędowi oops zaobserwowanemu w komputerze PPC w procedurze obsługi zegara karty sieciowej:

```
Oops: Exception in kernel mode, sig: 4
Unable to handle kernel NULL pointer dereference at virtual address 00000001

NIP: C013A7F0 LR: C013A7F0 SP: C0685E00 REGS: c0905d10 TRAP: 0700
Not tained
MSR: 00089037 EE: 1 PR: 0 FP: 0 ME: 1 IR/DR: 11
TASK = c0712530[0] 'swapper' Last syscall: 120
GPR00: C013A7C0 C0295E00 C0231530 0000002F 00000001 C0380CB8 C0291B80 C02D0000
GPR08: 000012A0 00000000 00000000 C0292AA0 4020A088 00000000 00000000 00000000
GPR16: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR24: 00000000 00000005 00000000 00001032 C3F70000 00000032 FFFFFFFF C3F7C1C0
Call trace:
[c013ab30] tulip_timer+0x128/0x1c4
[c0020744] run_timer_softirq+0x10c/0x164
[c001b864] do_softirq+0x88/0x104
[c0007e80] timer_interrupt+0x284/0x298
[c00033c4] ret_from_except+0x0/0x34
[c0007b84] default_idle+0x20/0x60
[c0007bf8] cpu_idle+0x34/0x38
[c0003ae8] rest_init+0x24/0x34
```

Użytkownicy komputerów klasy PC mogą dziwić się znaczną liczbą rejestrów (aż 32!). Błędy oops w znacznie popularniejszych systemach klasy x86-32 generują dużo krótszy wydruk zawartości rejestrów. Najważniejsze informacje są jednak w obu architekturach identyczne — liczy się zawartość rejestrów i ślad wykonania.

Ślad wykonania to łańcuch wywołania funkcji prowadzący do błędu. W prezentowanym przypadku ślad zdradza dokładnie przebieg wykonania aż do momentu wystąpienia błędu — system realizował proces jałowy w pętli `cpu_idle()`, w której wywoływana jest funkcja `default_idle()`. Zgłoszenie przerwania zegarowego spowodowało zainicjowanie obsługi liczników. Uruchomiona wtedy procedura obsługi takiego licznika dla karty sieciowej `tulip_timer()` wykonała w ramach przetwarzania wyłuskanie wskaźnika o wartości `NULL`. Na podstawie podanych na wydruku offsetów (tutaj *0x128/0x1c4*) można wręcz wytypować konkretny wiersz programu, którego wykonanie spowodowało błąd.

Równie przydatna jak ślad wykonania może być zawartość rejestrów. Dysponując kopią problematycznej funkcji w postaci kodu asemblerowego, można na podstawie zawartości rejestrów odtworzyć dokładnie stan jej wykonania w momencie wystąpienia błędu. Obecność w jednym z rejestrów nieoczekiwanej wartości może często rzucić światło na przyczynę błędu. W analizowanym przypadku widać, które z rejestrów zawierają wartości NULL (zera), i na tej podstawie można wytypować argument wywołania bądź zmienną lokalną funkcji, która przyjęła niewłaściwą wartość. Dla sytuacji podobnych do analizowanej typowym źródłem błędu jest sytuacja hazardowa — w tym konkretnym przypadku chodzi o rywalizację w dostępie do zasobu pomiędzy obsługą licznika a resztą kodu obsługi karty sieciowej. Diagnostyka sytuacji hazardowych jest zresztą zawsze problematyczna.

Polecenie ksymoops

Prezentowany wcześniej wydruk generowany w ramach obsługi błędu oops nosi nazwę *zdekodowanego*, ponieważ adresy pamięci zostały na wydruku przetłumaczone na nazwy funkcji rezydujących pod tymi adresami. Wersja niezdekodowana tego samego błędu wyglądałaby następująco:

```
NIP: C013A7F0 LR: C013A7F0 SP: C0685E00 REGS: c0905d10 TRAP: 0700
Not tained
MSR: 00089037 EE: 1 PR: 0 FP: 0 ME: 1 IR/DR: 11
TASK = c0712530[0] 'swapper' Last syscall: 120
GPR00: C013A7C0 C0295E00 C0231530 0000002F 00000001 C0380CB8 C0291B80 C02D0000
GPR08: 000012A0 00000000 00000000 C0292AA0 4020A088 00000000 00000000 00000000
GPR16: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR24: 00000000 00000005 00000000 00001032 C3F70000 00000032 FFFFFFFF C3F7C1C0
Call trace: [c013ab30] [c0020744] [c001b864] [c0007e80] [c00033c4]
[c0007b84] [c0007bf8] [c0003ae8]
```

Adresy wymienione w ścieżce śladu muszą dopiero zostać przetłumaczone na nazwy symboliczne funkcji. Jest to możliwe za pośrednictwem polecenia ksymoops i pliku *System.map* generowanego w trakcie kompilacji jądra. Jeżeli jądro korzysta z ładowanych dynamicznie modułów, potrzebne będą również informacje o tych modułach. Polecenie ksymoops próbuje samodzielnie pozyskać wszystkie potrzebne do tłumaczenia informacje, stąd przeważnie wystarczy uruchomić je wywołaniem:

```
ksymoops zapis_wydruku_oops.txt
```

Wynikiem działania programu jest wydrukowanie zdekodowanej wersji wydruku błędu oops. Jeżeli domyślne ustawienia programu ksymoops nie spełniają swojej roli i zachodzi potrzeba określenia własnych źródeł informacji pomocnych w translacji adresów, program można konfigurować szeregiem opcji wywołania. Większość informacji potrzebnych do rozpoczęcia korzystania z programu zawiera dokumentacja systemowa man. Program ksymoops wchodzi w skład większości dystrybucji systemu Linux.

kallsyms

Na szczęście korzystanie z programu `ksymoops` nie jest już konieczne. Na szczęście, ponieważ w przeciwieństwie do programistów jądra szeregowi użytkownicy systemu miewali spore kłopoty z wytypowaniem odpowiedniego pliku *System.map* i tym samym zdekodowaniem wydruku błędu oops.

W wersji rozwojowej 2.5 jądra systemu Linux pojawiła się funkcja `kallsyms`, aktywowana za pośrednictwem opcji konfiguracyjnej `CONFIG_KALLSYMS`. Uaktywnienie tej opcji powoduje zapisanie w jądrze symbolicznych nazw adresów funkcji skonsolidowanych do obrazu jądra, dzięki czemu jądro może wypisać zdekodowany stos wywołań. Z tego względu dekodowanie błędów oops nie wymaga już dostępności pliku *System.map* ani polecenia `ksymoops`. Z drugiej strony powoduje to zwiększenie rozmiaru jądra, jako że odwzorowanie symboli musi być umieszczone w trwale odwzorowanej pamięci jądra. Koszt ten jednak szybko się zwraca, zwłaszcza przy rozwijaniu jądra, ale także podczas próbnych wdrożeń. Opcja konfiguracyjna jądra `CONFIG_KALLSYMS_ALL` dodatkowo zapisuje nazwy symboliczne wszystkich symboli (nie tylko funkcji), choć to akurat przydaje się tylko w specjalistycznych narzędziach diagnostycznych.

Opcja `CONFIG_KALLSYMS_EXTRA_PASS` powoduje, że w ramach kompilacji i konsolidacji jądra zostanie wykonany drugi przebieg po kodzie obiektowym jądra. Ta operacja służy z kolei przede wszystkim do diagnostyki działania samego mechanizmu `kallsyms`.

Opcje diagnostyczne jądra

W testowaniu kodu jądra i diagnostyce ewentualnych błędów pomocne mogą okazać się dostępne liczne opcje konfiguracyjne kompilacji jądra. Opcje te zebrane są w dziale *Kernel Hacking* opcji konfiguracji jądra. Aktywność wszystkich tych opcji uzależniona jest od włączenia ogólnej opcji `CONFIG_DEBUG_KERNEL`. Przy modyfikacjach jądra warto włączyć wszystkie z dostępnych opcji wspomagających testowanie i diagnostykę.

Niektóre z tych opcji pozwalają na wspomaganie diagnostyki błędów alokatora plastrowego, diagnostyki błędów pamięci wysokiej, odwzorowań wejścia-wyjścia, diagnostyki rygli pętlowych i kontroli przepełnienia stosu. Jedną z najprzydatniejszych opcji jest jednak opcja określana w menu konfiguracyjnym jako *sleep-inside-spinlock checking* (kontrola zawieszenia przy przetrzymywaniu rygla pętlowego, ale nie tylko).

Począwszy od wersji 2.5, jądro systemu Linux dysponuje znakomitą infrastrukturą wspomagającą wykrywanie wszelkiego rodzaju błędów niepodzielności operacji. Z rozdziału 9. wiadomo, że *niepodzielność* to zdolność do nieprzerwanego wykonania kodu — kod niepodzielny powinien zostać wykonany bez przerwania albo nie powinien być wykonany wcale. Kodem niepodzielnym jest kod przetrzymujący rygiel pętlowy bądź kod wykonywany przy wyłączeniu wywłaszczania jądra. Kod niepodzielny nie może być zawieszany — zawieszanie kodu przetrzymującego rygiel pętlowy prowadzi prosto do zakleszczenia.

Dzięki wywłaszczaniu jądra jądro to dysponuje centralnym licznikiem niepodzielności. Można dzięki niemu skonfigurować jądro tak, aby zawieszenie zadania w sekcji krytycznej albo nawet inicjowanie operacji potencjalnie zawieszającej zadanie spowodowało wyświetlenie komunikatu o błędzie wraz ze śladem wykonania. W ten sposób dają się wykryć takie błędy jak wywołanie funkcji `schedule()` przez proces przetrzymujący blokadę, inicjowanie przez taki proces blokującej operacji przydziału pamięci czy zawieszanie procesu odwołującego się do danych charakterystycznych dla procesora. Owa infrastruktura diagnostyczna pozwala na wychwycenie znacznej części błędów, więc jej wykorzystywanie jest w procesie testowania i diagnozowania gorąco zalecane.

Wspomniana infrastruktura jest najlepiej wykorzystywana przy następujących ustawieniach opcji:

```
CONFIG_PREEMPT=y
CONFIG_DEBUG_KERNEL=y
CONFIG_KALLSYMS=y
CONFIG_DEBUG_SPINLOCK_SLEEP=y
```

Asercje i wypisywanie informacji o błędach

Do oznaczania błędów, wprowadzania asercji i wyprowadzania informacji służy w jądrze szereg wywołań i makrodefinicji. Najczęściej używane spośród nich to `BUG()` i `BUG_ON()`. Ich wywołanie powoduje wyprowadzenie komunikatu o błędzie oops wraz z informacjami o stanie systemu i śladzie wykonania. Sposób prowokowania błędu w tych wywołaniach zależny jest od architektury systemu. W większości architektur makrodefinicje `BUG()` i `BUG_ON()` rozwijane są do postaci instrukcji niedozwolonych, których próba wykonania prowokuje błąd oops. Makrodefinicje te wykorzystuje się w roli asercji do oznaczania sytuacji, które nie powinny mieć miejsca:

```
if (bad_thing)
    BUG();
```

albo nawet lepiej:

```
BUG_ON(bad_thing);
```

Większość programistów jądra uważa, że `BUG_ON()` stosuje się i czyta łatwiej niż `BUG()`; ponadto `BUG_ON()` ujmuje asercję w instrukcji `unlikely()`. Wśród programistów jądra dyskutowano o pomyśle opcji kompilacji wyrażeń z `BUG_ON()` jako pustych, co dawałoby znaczne oszczędności miejsca w jądrach przeznaczonych do systemów wbudowanych. To wymaga jednak, aby asercja umieszczona w `BUG_ON()` nie posiadała żadnych efektów ubocznych. Zaproponowano też makrodefinicję `BUILD_BUG_ON()` jako asercję czasu kompilacji — jeśli znajdujące się w niej wyrażenie daje wartość `true` w czasie kompilacji, kompilacja zostanie przerwana z komunikatem o błędzie.

Błędy bardziej krytyczne mogą być sygnalizowane wywołaniem `panic()`. Wywołanie to powoduje wyświetlenie komunikatu o błędzie i zatrzymanie systemu. Z oczywistych względów należy je stosować wyłącznie w sytuacjach najtrudniejszych:

```
if (terrible_error)
    panic("okropna sytuacja: %ld!\n", terrible_error);
```

Niekiedy testowanie i diagnostykę można sobie znakomicie ułatwić, wyprowadzając na konsolę wydruk śladu stosu. Służy do tego wywołanie `dump_stack()`. Jego działanie ogranicza się do wyprowadzenia na konsolę zawartości rejestrów i śladu wywołania funkcji:

```
if (!debug_check) {
    printk(KERN_DEBUG "tu dodatkowy komunikat ...\n");
    dump_stack();
}
```

Funkcja Magic SysRq Key

Sporo nerwów testerowi jądra zaoszczędzić może funkcja Magic SysRq Key aktywowana za pośrednictwem opcji konfiguracyjnej `CONFIG_MAGIC_SYSRQ`. Klawisz *SysRq* (żądanie systemowe) obecny jest na większości standardowych klawiatur. W architekturach i386 i PowerPC aktywuje się go za pośrednictwem kombinacji klawiszy *Alt+Print Screen*. Po uaktywnieniu opcji `CONFIG_MAGIC_SYSRQ` można za pośrednictwem specjalnych kombinacji klawiszy komunikować się z jądrem. Możliwość ta pozwala na wykonywanie pewnych operacji nawet w obliczu „martwego" systemu.

Funkcję specjalnych kombinacji klawiszy można też włączyć, zmieniając parametry działającego systemu za pośrednictwem następującego polecenia:

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

Lista dostępnych operacji wyświetlana jest na konsoli po naciśnięciu kombinacji klawiszy *SysRq* i *h*. Naciśnięcie *SysRq* i *s* powoduje utrwalenie brudnych buforów na dysku, *SysRq* i *u* odmontowuje wszystkie zamontowane systemy plików, natomiast kombinacja *SysRq* i *b* powoduje restart systemu. Użycie powyższych trzech kombinacji klawiszy w podanej kolejności to najbezpieczniejszy sposób restartu niestabilnego systemu, nieporównanie bezpieczniejszy od prostego wciśnięcia przycisku *reset* na obudowie komputera.

System, który uległ całkowitemu zawieszeniu, może nie reagować na kombinacje z klawiszem *SysRq*, ewentualnie może niepoprawnie realizować wydawane w ten sposób polecenia. Jednak przy odrobinie szczęścia włączenie opcji może pozwolić na uratowanie danych i znakomicie wspomóc diagnostykę jądra. Lista poleceń obsługiwanych w ramach funkcji Magic SysRq umieszczona została w tabeli 18.2.

Tabela 18.2. Obsługiwane polecenia SysRq

Kombinacja klawiszy	Opis
SysRq i *b*	Restart komputera.
SysRq i *e*	Wysłanie sygnału `SIGTERM` do wszystkich procesów z wyjątkiem procesu `init`.
SysRq i *h*	Wyświetlenie listy poleceń `SysRq` na konsoli.
SysRq i *i*	Wysłanie sygnału `SIGKILL` do wszystkich procesów z wyjątkiem procesu `init`.
SysRq i *k*	Unicestwienie wszystkich programów uruchomionych z danej konsoli.
SysRq i *l*	Wysłanie sygnału `SIGKILL` do wszystkich procesów, również do procesu `init`.
SysRq i *m*	Wygenerowanie zrzutu pamięci i przekazanie go na konsolę.
SysRq i *o*	Zatrzymanie systemu i wyłączenie komputera.
SysRq i *p*	Przekazanie na konsolę wydruku zawartości rejestrów.
SysRq i *r*	Wyłączenie trybu RAW klawiatury i przełączenie jej do trybu XLATE.
SysRq i *s*	Synchronizacja wszystkich systemów plików (utrwalenie zmian buforów dyskowych).
SysRq i *t*	Przekazanie na konsolę wydruku opisującego proces.
SysRq i *u*	Odmontowanie wszystkich zamontowanych systemów plików.

Więcej informacji o funkcji Magic `SysRq` można znaleźć w pliku *Documentation/sysrq.txt*, w drzewie katalogów kodu źródłowego jądra. Faktyczna implementacja funkcji zdefiniowana jest zaś w pliku *drivers/tty/sysrq.c*. Funkcja Magic `SysRq` to ważne narzędzie diagnostyki i ratowania „padającego" systemu, jednak z uwagi na to, że jej włączenie daje znaczne możliwości każdemu użytkownikowi konsoli, należałoby ograniczyć stosowanie specjalnych kombinacji klawiszy w komputerach, których bezpieczeństwo jest z jakichś względów ważne. Na prywatnym komputerze wykorzystywanym do programowania i testowania jądra nie ma przeciwwskazań co do włączania funkcji Magic `SysRq`.

Saga debugera jądra

Wielu programistów jądra od dawna domagało się działającego w jądrze debugera. Niestety, Linus nie podzielał tej chęci i blokował próby wprowadzenia debugera do jądra. Uzasadniał to tym, że obecność debugerów powoduje wprowadzanie błędnych poprawek

jest łata wyprowadzona z pełnej znajomości i kodu jądra, i objawów błędu. Mimo to potrzeba wyposażenia jądra w debuger wcale nie osłabła. Z racji małego prawdopodobieństwa spełnienia tych postulatów w najbliższej przyszłości zdesperowani programiści opracowali szereg łat uzupełniających jądro systemu Linux o obsługę debugera. I choć są to łaty nieoficjalne, stanowią zwykle dobrze wyposażone i efektywne narzędzia. Przed ich omówieniem wypadałoby jednak sprawdzić, w jakim zakresie do diagnostyki jądra nie przyda się standardowy linuksowy debuger `gdb`.

gdb

Zajrzeć w głąb działającego jądra można za pomocą standardowego debugera GNU. Uruchomienie debugera na rzecz jądra nie różni się od inicjowania debugera na rzecz zwykłych procesów:

```
gdb vmlinux /proc/kcore
```

Plik *vmlinux* to nieskompresowany plik obrazu jądra przechowywany w katalogu głównym drzewa katalogów kompilacji jądra.

Opcjonalny parametr wywołania debugera */proc/kcore* pełni w uruchomieniu debugera rolę pliku obrazu pamięci, za pośrednictwem którego `gdb` może uzyskać podgląd pamięci jądra. Odczyt pliku */proc/kcore* wymaga uprawnień użytkownika uprzywilejowanego (*root*) systemu.

Po uruchomieniu debugera można korzystać z dowolnych implementowanych w nim poleceń podglądu danych. Na przykład w celu podejrzenia wartości zmiennej `global_variable` należy wystosować polecenie:

```
p global_variable
```

Do deasemblacji funkcji służy zaś polecenie `disassemble`:

```
disassemble nazwa-funkcji
```

Jeżeli jądro było kompilowane z opcją `-g` (czyli było kompilowane po dodaniu `-g` do zmiennej środowiskowej `CFLAGS` w pliku *Makefile* kompilacji jądra), debuger `gdb` może wydobyć z niego znacznie więcej informacji. Możliwe jest wtedy na przykład podglądanie zawartości struktur i wyłuskiwanie wskaźników. Określenie opcji `-g` powoduje jednak znaczny rozrost jądra, więc nie należy automatycznie uaktywniać tej opcji przy kompilacji jądra do celów innych niż testowe i diagnostyczne.

Niestety, na tym wyczerpuje się lista możliwości debugowania jądra za pomocą programu `gdb`. Nie da się za jego pośrednictwem modyfikować danych działającego jądra. Nie można też przy jego użyciu wykonywać kodu jądra krokowo ani też ustawiać punktów wstrzymania wykonania. Niemożność modyfikacji struktur danych jądra to poważna wada. I choć możliwość deasemblacji funkcji jest od czasu do czasu przydatna, ogólna przydatność debugera `gdb` jest — w wyniku braku możliwości modyfikacji danych jądra — ograniczona.

kgdb

`kgdb` to łata programowa na jądro systemu Linux pozwalająca na pełne zdalne debugowanie jądra realizowane za pośrednictwem łącza szeregowego. Wykorzystanie debugera `kgdb` wymaga zastosowania dwóch komputerów. Pierwszy z nich wykonuje kod jądra wyposażonego w łatę `kgdb`, podczas gdy drugi kontroluje wykonanie kodu jądra za pośrednictwem łącza szeregowego (tak zwanego kabla null-modem łączącego oba komputery) i debugera `gdb`. Dzięki `kgdb` debugowanie jądra można wzbogacić o cały szereg niezwykle przydatnych operacji implementowanych przez program `gdb`, jak definiowanie punktów wstrzymania wykonania, krokowe wykonanie kodu jądra czy podgląd i modyfikacja dowolnych danych jądra itd.! Niektóre z wersji `kgdb` pozwalają nawet na wykonywanie wybranych funkcji.

Konfiguracja dwóch komputerów jest dość złożona, ale włożony w nią wysiłek owocuje późniejszym prostym debugowaniem jądra. Łata oprócz kodu debugera zawiera obszerną dokumentację instalowaną w podkatalogu *Documentation* drzewa katalogów kodu źródłowego jądra. Warto się z nią zapoznać.

Łaty `kgdb` dostępne są w rozmaitych wersjach dostosowanych do najróżniejszych architektur sprzętowych i wersji jądra. Najlepszym sposobem pozyskania łaty jest więc przeszukiwanie zasobów internetu.

Stymulowanie i sondowanie systemu

W miarę nabywania doświadczenia w diagnostyce błędów jądra poznaje się sztuczki pozwalające na stymulowanie i sondowanie jądra. Debugowanie jądra jest niewątpliwie czynnością skomplikowaną, stąd wszelka pomoc, nawet w postaci prostych sztuczek, jest na wagę złota. Oto kilka z takich trików.

Uzależnianie wykonania kodu od identyfikatora UID

Jeżeli tworzony kod ma związek z obsługą procesów, możliwe jest opracowanie kodu alternatywnego bez eliminowania z jądra dotychczasowej implementacji. Jest to szczególnie przydatne w przypadku tworzenia nowych wersji istotnych wywołań systemowych, kiedy to pożądane jest w pełni poprawne działanie systemu, uruchamiającego nowy kod tylko na wyraźne żądanie. Weźmy na przykład wywołanie `fork()`. Jeżeli algorytm tego wywołania zostanie zmodyfikowany pod kątem uwzględnienia w nim nowej, fascynującej możliwości, testowanie i diagnostyka nowej wersji wywołania będą w przypadku błędu implementacji bardzo utrudnione — system, w którym nie działa funkcja `fork()`, nie będzie bowiem ani trochę stabilny. Nie można się jednak poddawać.

W takich przypadkach warto pozostawić istniejący algorytm i wstrzymywać niekontrolowane wykonywanie jego nowej wersji. Można na przykład uzależnić uruchomienie nowego algorytmu od wartości identyfikatora użytkownika (UID):

```
if (current->uid != 7777) {
    /* dotychczasowy algorytm */
} else {
    /* nowa wersja algorytmu... */
}
```

Dla wszystkich użytkowników o identyfikatorze UID różnym od 7777 realizowana będzie dotychczasowa, na pewno działająca wersja algorytmu. Testowanie nowej wersji odbywa się wtedy przez utworzenie specjalnego użytkownika o identyfikatorze UID 7777. Znakomicie ułatwia to testowanie kodu o krytycznym znaczeniu dla poprawnego działania procesu.

Korzystanie ze zmiennych warunkowych

Jeżeli niepewny kod nie jest realizowany w kontekście procesu albo jeżeli sterowanie jego wykonaniem ma być bardziej globalne, należy skorzystać ze zmiennych warunkowych. Ta metoda jest nawet prostsza od rozróżniania identyfikatora UID. Wystarczy bowiem utworzyć zmienną globalną i jej wartość wykorzystywać jako barierę wykonania eksperymentalnego kodu. Na przykład wartość zero zmiennej powoduje wykonanie jednej ścieżki kodu, a wartość niezerowa inicjuje wykonanie innej ścieżki kodu. Zmienna może być ustawiana za pośrednictwem eksportowanego do przestrzeni użytkownika interfejsu bądź z poziomu debugera.

Korzystanie ze statystyk

Nierzadko diagnostyka niewłaściwego działania jądra opiera się na informacji o częstotliwości zachodzenia określonego zdarzenia. Niekiedy zachodzi konieczność porównania wielu zdarzeń i wygenerowania pewnych wielkości porównawczych. Można to łatwo zrealizować, kompletując odpowiednie statystyki i implementując mechanizm eksportu ich wartości.

Niech porównywanymi wartościami będą na przykład liczby zdarzeń `bla` i `blabla`. W pliku kodu (najlepiej tym, który obsługuje dane zdarzenia) wystarczy wtedy zdefiniować dwie zmienne globalne:

```
unsigned long bla_stat = 0;
unsigned long blabla_stat = 0;
```

Przy każdym wystąpieniu obserwowanego zdarzenia należy zwiększyć wartość odpowiedniej zmiennej i opracować dowolny mechanizm eksportu danych. Na przykład można w tym celu utworzyć plik w podsystemie plików */proc* i w nim umieszczać wartości zmiennych, ewentualnie zaimplementować specjalne wywołanie systemowe. W ostateczności można też podglądać wartości zmiennych za pomocą debugera.

Warto jednak pamiętać, że taka metoda nie sprawdza się najlepiej w środowisku wieloprocesorowym. W takim środowisku manipulowanie wartościami zmiennych wymaga bowiem stosowania operacji niepodzielnych. Jednak w przypadku prostych statystyk wykorzystywanych jedynie do diagnostyki taka ochrona może być zbędna.

Ograniczanie częstotliwości i liczby komunikatów diagnostycznych

Popularną metodą diagnostyki błędów jest szpikowanie podejrzanego kodu instrukcjami testującymi wartości pewnych zmiennych i wyprowadzającymi komunikaty diagnostyczne na konsolę. Należy jednak wziąć pod uwagę fakt, że w jądrze niektóre z funkcji są wywoływane dziesiątki razy w ciągu sekundy. Jeżeli taka funkcja zostanie naszpikowana wywołaniami `printk()`, system zostanie przeładowany komunikatami diagnostycznymi.

Problem natłoku komunikatów można rozwiązać na dwa proste sposoby. Pierwszy z nich to *ograniczanie częstotliwości* diagnostyki przydatne przy obserwacji postępu zdarzeń, które zachodzą z dość dużą częstotliwością. Aby uniknąć zalewu komunikatów diagnostycznych, należy ograniczyć częstotliwość generowania komunikatów, wyprowadzając je, powiedzmy, wyłącznie raz na kilka sekund. Na przykład:

```
static unsigned long prev_jiffy = jiffies   /* ograniczanie częstotliwości */

if (time_after(jiffies, prev_jiffy + 2*HZ)) {
    prev_jiffy = jiffies;
    printk(KERN_ERR "ple ple ple\n");
}
```

W powyższym przykładzie komunikat diagnostyczny jest wyświetlany na konsoli co dwie sekundy. Zapobiega to zalewowi informacji i ryzyku zablokowania konsoli. Częstotliwość można ograniczać jeszcze bardziej bądź pozwalać na częstsze generowanie komunikatów, zależnie od potrzeb.

Można też skorzystać ze specjalnej funkcji ograniczającej częstotliwość generowania komunikatów diagnostycznych:

```
if (error && printk_ratelimit())
    printk(KERN_DEBUG "awaria=%d\n", error);
```

Albo z jeszcze wygodniejszego wariantu:

```
printk_ratelimited(KERN_DEBUG "awaria=%d\n", error);
```

Funkcja `printk_ratelimit()` zwraca zero, jeśli wskutek ograniczania częstotliwości komunikatów diagnostycznych wywołanie `printk()` powinno być pominięte, albo wartość niezerową, gdyby należało je wykonać. Domyślnie funkcja ta przepuszcza jeden komunikat co 5 sekund, ale przedtem przepuszcza nielimitowaną serię do 10 komunikatów. Oba parametry (częstotliwość i rozmiar nielimitowanej serii) można ustawić za pośrednictwem wymuszenia `sysctl` dla parametrów (odpowiednio) `printk_ratelimit` i `printk_ratelimit_burst`. Zaprezentowana powyżej makrodefinicja `printk_ratelimited()` realizuje instrukcję warunkową z `printk_ratelimit()` i ewentualne wywołanie `printk()` bez konieczności jawnego stosowania instrukcji `if`.

Osobny problem pojawia się wtedy, kiedy analizowany ma być przebieg wykonania wzdłuż ścieżki kodu — tym razem chodzi nie o doczekanie się pewnego stanu jądra, a po prostu o powiadomienie o wystąpieniu zdarzenia. Przydatne jest pierwsze takie

powiadomienie, ewentualnie pierwszych kilka. Taka sytuacja dotyczy problemów, w których pierwsze zajście określonego zdarzenia pociąga za sobą lawinę takich samych zdarzeń. Tutaj rozwiązaniem nie jest ograniczanie częstotliwości komunikatów, lecz proste *zmniejszenie* ich liczby:

```
static unsigned long limit = 0;

if (limit < 5) {
    limit++;
    printk(KERN_ERR "ple ple ple\n");
}
```

W tym przykładzie liczba komunikatów diagnostycznych ograniczona jest do pięciu. Po wystosowaniu pięciu komunikatów warunek kontynuowania diagnostyki jest już zawsze fałszywy.

W obu ostatnich przykładach zmienne powinny być statyczne i lokalne względem funkcji — gwarantuje to podtrzymanie wartości zmiennej i tym samym kontynuowanie strategii ograniczania w kolejnych wywołaniach funkcji.

Żaden z powyższych przykładów nie jest zbyt dobrze przystosowany do działania w środowisku wieloprocesorowym — niedomaganie to można jednak szybko wyeliminować, wdrażając w nich operacje niepodzielne. Ale prawdę mówiąc, w kodzie diagnostycznym nie warto chyba zawracać sobie tym głowy.

Szukanie winowajcy — wyszukiwanie binarne

Niejednokrotnie znaczną pomocą w wytypowaniu błędu jest wiedza o momencie, w którym błąd pojawił się w jądrze, w sensie znajomości numeru wersji jądra, od której zaczęto obserwować objawy błędu. Jeżeli na przykład wiadomo, że błąd ujawnił się w wersji 2.6.33, ale nie w wersji 2.6.28, to sprawa jest jasna — wytypowanie błędu polega zazwyczaj na przejrzeniu i przeanalizowaniu listy zmian pomiędzy tymi wersjami (niekiedy wystarczy wycofać feralną zmianę).

Wielokrotnie jednak nie wiadomo na pewno, w której wersji pojawił się dany błąd. Wiadomo jedynie, że obecny jest w wersji *bieżącej*, i wydaje się, że był w tej wersji od zawsze. Wtedy określenie momentu wprowadzenia błędu do kodu jądra wymaga przeprowadzenia śledztwa, które przy odrobinie wysiłku zaowocuje jednak wytypowaniem winowajcy. Po określeniu zmiany wprowadzającej błąd jego usunięcie jest zwykle kwestią czasu, i to niezbyt długiego.

W takich poszukiwaniach potrzebny jest błąd, którego wystąpienie można w sposób pewny sprowokować. Najlepiej, jeśli błąd daje się wywołać zaraz po zakończeniu rozruchu systemu. Następnie należy zaopatrzyć się w jądro, co do którego wiadomo na pewno, że jest wolne od rzeczonego błędu. Może to być na przykład jądro wykorzystywane kilka miesięcy temu, na którym nigdy nie udało się sprowokować błędu. Równie dobrze może to być jednak wersja znacznie wcześniejsza. O ile błąd nie czaił się w jądrze od zawsze, poszukiwania takie dają zwykle wyniki stosunkowo szybko.

Potrzebne jest też jądro, o którym wiadomo na pewno, że jest obarczone błędem. Najlepiej, jeżeli będzie to jak najwcześniejsza wersja jądra na pewno zawierająca błąd.

Po uzyskaniu jądra poprawnego należy rozpocząć wyszukiwanie binarne od wersji skażonej do wersji dobrej. Załóżmy, że najnowsze jądro, o którym wiadomo, że jest wolne od błędu, to jądro 2.6.11, a najwcześniejsze jądro z błędem to jądro 2.6.20. Poszukiwania należy rozpocząć od wersji w miarę równo odległej od obu wersji skrajnych, na przykład 2.6.15. Jeżeli w wyniku testów okaże się, że wersja ta jest wolna od błędu, to wiadomo, że błąd pojawił się w kodzie następnych wersji. Należy więc zbadać wersję środkową pomiędzy wersjami 2.6.15 i 2.6.20 — na przykład 2.6.17. Jeżeli zaś test wykaże obecność błędu już w wersji 2.6.15, należy zawęzić poszukiwania do wersji 2.6.11 – 2.6.15 i rozpocząć je od na przykład wersji 2.6.13. Poszukiwania trzeba następnie kontynuować w odpowiednich połówkach tak ograniczonego zakresu wersji.

W końcu kolejne testy doprowadzą do zawężenia zakresu poszukiwań do dwóch kolejnych wersji jądra, w których jedna będzie obarczona błędem, a druga będzie od niego wolna. W tym momencie można już w prosty sposób określić pechową zmianę pomiędzy wersjami. Warto zauważyć, że taki sposób przeszukiwania jest znacznie efektywniejszy niż przeszukiwanie wszystkich kolejnych wersji jądra!

Binarne wyszukiwanie wersji za pomocą Gita

System zarządzania zmianami w kodzie źródłowym Git udostępnia użytkownikom bardzo przydatny i wygodny mechanizm przeprowadzania binarnego wyszukiwania wersji. Każdy, kto kontroluje kopię drzewa kodu źródłowego jądra systemu Linux za pośrednictwem Gita (a jest to metoda preferowana i powszechnie stosowana), może zautomatyzować wyszukiwanie feralnej wersji, składając go na samo repozytorium. Co więcej, Git pozwala na przeprowadzenie wyszukiwania binarnego nie według wersji wydawniczych jądra, ale według zmian zatwierdzanych do repozytorium, a więc z dokładnością do jednej *rewizji* (wersje wydawnicze to agregaty nawet setek rewizji kodu jądra). Dodatkowo sama operacja jest naprawdę prosta (co w przypadku Gita nie zawsze jest oczywiste). Na początek wystarczy zażądać rozpoczęcia wyszukiwania binarnego:

```
$ git bisect start
```

Następnie podaje się najwcześniejszą znaną rewizję, w której błąd występował:

```
$ git bisect bad rewizja
```

Jeśli nie wiadomo, w której wersji błąd się ujawnił, nie trzeba w ogóle podawać rewizji — Git domyślnie uzna, że chodzi o rewizję najnowszą:

```
$ git bisect bad
```

Potem podaje się najpóźniejszą rewizję, o której wiadomo, że nie zawierała błędu:

```
$ git bisect good v2.6.28
```

Wtedy Git przystępuje do automatycznego wyciągnięcia drzewa kodu źródłowego Linuksa z rewizji środkowej podanego przedziału. Wystarczy jądro skompilować, uruchomić, no i przetestować. Jeśli błędu nie ma, polecenie:

```
$ git bisect good
```

wymusi wyszukiwanie binarne wśród nowszych rewizji. Jeśli błąd się ujawni (testowana rewizja manifestuje błąd), należy powiadomić Gita, że kod z tej rewizji jest niepoprawny, i tym samym skierować poszukiwania na wcześniejsze rewizje:

```
$ git bisect bad
```

Za każdym razem Git dokona bisekcji drzewa kodu źródłowego w kolejnym zakresie rewizji. Wystarczy uparcie powtarzać bisekcję do momentu, w którym następnej nie da się już wykonać — wtedy Git wypisze feralny numer rewizji, która wprowadziła błąd do kodu jądra.

Szukanie błędnej rewizji może potrwać, ale automatyzm wydatnie upraszcza całą procedurę. Może ona być jeszcze łatwiejsza i precyzyjniejsza, jeśli przybliżymy Gitowi położenie błędnego kodu. Jeśli na przykład podejrzewamy, że pojawił się błąd w kodzie rozruchu systemu dla architektury x86, możemy zawęzić poszukiwania do tych rewizji, które faktycznie zmieniały coś w ścieżce zawierającej podejrzewany kod:

```
$ git bisect start – arch/x86
```

Koledzy — kiedy wszystko inne zawiedzie

Jeżeli wszelkie próby wytypowania momentu i miejsca wprowadzenia błędu do kodu jądra spełzną na niczym i jeżeli godziny — a najpewniej dni — spędzone nad klawiaturą okażą się bezowocne, to gdy błąd dotyczy głównej wersji jądra, można zawsze liczyć na pomoc innych programistów, tworzących społeczność jądra systemu Linux.

Receptą w takich przypadkach jest wysłanie krótkiego, ale treściwego listu, zawierającego możliwie pełny opis błędu, na adres listy dystrybucyjnej poświęconej programowaniu jądra. W końcu nikt nie lubi błędów i prędzej czy później (zwykle prędzej) pojawi się odzew.

Adresy list dystrybucyjnych, z których korzysta społeczność programistów Linuksa, i adres głównego forum tej społeczności, jakim jest lista *Linux Kernel Mailing List* (*LKML*), podane zostały w rozdziale 20.

Podsumowanie

Kończący się rozdział poświęcony był zagadnieniu diagnostyki błędów w kodzie jądra, czyli procesowi ustalania, *dlaczego* implementacja rozjechała się z planowanym zachowaniem. Wśród zaprezentowanych środków diagnostycznych znalazły się zarówno mechanizmy wbudowane w samo jądro, jak i zewnętrzne debugery. Wśród omawianych technik

w pierwszej kolejności uwzględniono diagnostykę przez komunikaty, na koniec zaś udało się zaprezentować skuteczną technikę wyszukiwania binarnego feralnej wersji kodu w repozytorium Git. Diagnozowanie błędów jądra to doprawdy zadanie nietrywialne, znacznie trudniejsze od diagnostyki zachowania programów przestrzeni użytkownika. Dlatego przyswojenie technik i mechanizmów z tego rozdziału ma zasadnicze znaczenie nie tylko dla testerów, ale przede wszystkim dla programistów jądra.

Następny rozdział poświęcimy innemu dość ogólnemu zagadnieniu, a mianowicie przenośności jądra systemu Linux.

Skorowidz

C

D

E

F

G

H

I

J

Ł

M

N

O

P

R

S

Ś

T

U

Z

Przekonaj się, jak działa jądro Linuksa!

Jądro systemu Linux to jedno z największych osiągnięć otwartej społeczności programistów. Projekt ten, zainicjowany przez Linusa Torvaldsa, światło dzienne ujrzał w 1991 roku. Od tamtego czasu coraz więcej ochotników wspiera jego doskonalenie. Pozwoliło to na wprowadzenie do jądra wielu nowatorskich funkcji i wyznaczenie trendów w rozwoju współczesnego oprogramowania oraz systemów operacyjnych. Prace nad tym gigantycznym projektem przyczyniły się też do powstania mnóstwa innych produktów — w tym zdobywającego ogromną popularność rozproszonego systemu kontroli wersji Git.

Tak dynamiczny rozwój ma jeden niedobry skutek uboczny. Początkujący programiści mają problem z opanowaniem ogromu projektu i poznaniem jego architektury. Ten długotrwały proces będzie zdecydowanie krótszy dzięki przewodnikowi, który właśnie trzymasz w rękach. Przeprowadzi Cię on przez tajemnice systemu. Dowiesz się, jak pobrać kod źródłowy jądra, jak go skonfigurować oraz skompilować. W kolejnych rozdziałach poznasz kluczowe mechanizmy: zarządzania procesami, szeregowania zadań, wywołań systemowych oraz przerwań. Ponadto nauczysz się korzystać z urządzeń blokowych, pamięci podręcznej, sterowników i modułów. Ta książka jest obowiązkową lekturą dla wszystkich programistów, którzy chcą mieć swój wkład w rozwój projektu Torvaldsa. Sprawdzi się ona również w rękach osób, które po prostu chcą zrozumieć, jak działa ten niezwykły system.

Dzięki tej książce:

- pobierzesz, skonfigurujesz i skompilujesz źródła jądra
- zrozumiesz zasady zarządzania procesami
- poznasz struktury danych jądra
- wykorzystasz wirtualne systemy plików
- zyskasz możliwość samodzielnego rozwijania jądra systemu Linux

Addison Wesley
cena: 79,00 zł
ISBN 978-83-246-4273-1
9 788324 642731