Testowanie jądra Linuxa

... czyli czegoś, co nigdy nie miało być testowane.

„Given enough eyeballs, all bugs are shallow”
Linus Law, wg Eric'a S. Raymond'a

Dawno, dawno temu

Był sobie Linux

Pierwsze wersje Linuxa powstawały w czasach, kiedy, co prawda, niektórzy już słyszeli o testowaniu, ale było to zupełnie co innego niż testowanie, o którym się mówi, i które stosuje w naszych czasach. Software stosowało się na mniejszą skalę, zwłaszcza software open-source. Znacznie mniejsze były grona osób również tworzących to oprogramowanie, a w takich grupach jest znacznie łatwiej koordynować współpracę. W końcu, jądro Linuxa (mimo zapewnień p. Tannenbauma o jego monolityczności) było znacznie mniejsze niż jest dzisiaj. Swego czasu również wierzono, że programy open-source nie wymagają żadnego formalnego testowania, a wystarczy fakt dostępności kodu – przecież jeśli każdy może ten kod obejrzeć i uruchomić, to szybko znajdą się wszystkie błędy. Po co więc testować jądro?

Zwiększa się grono użytkowników

Linuxa z każdym dniem używa więcej i więcej prywatnych osób, jak i instytucji. Jedną z największych przewag Linuxa nad główną konkurencją jest to, że jest darmowy. Firmy mogą go używać bez wykupywania licencji na ogromne kwoty, za to za pieniądze mogą, jeśli chcą, wykupywać wsparcie. Powstają coraz to nowsze, przyjaźniejsze użytkownikowi dystrybucje Linuxa, przeznaczone dla osób niezainteresowanych grzebaniem w plikach konfiguracyjnych non-stop (np. piszący te słowa używa kubuntu :)). Błąd, który przedostałby się do jądra przy niedostatecznym jego przetestowaniu, miałby szansę się wielokrotnie powielać i powtarzać, mocno szkodząc ciężko wypracowanemu zaufaniu społecznemu.

Mniejszy margines błędu

Środowisko, w którym działa jądro, pozostawia znacznie mniejszy margines błędu. Program pracujący w userspace ma nad sobą jeszcze jądro, które całkowicie (lub prawie całkowicie) zasłania mu dostęp do samego sprzętu – jest jego piaskownicą. Programista kernela nie ma tej wygody.

A konsekwencje – bezlitosne (Oops...)

Konsekwencje czyhające na programistę aplikacji to, w najgorszym wypadku, nie obsłużony błąd, i wywalona aplikacja. W przypadku programisty kernela, tą aplikacją jest cały system...

Struś nie może sobie pozwolić na błąd, natomiast firma ACME nie testuje swoich produktów

Ale to takie trudne!

Jądro jest coraz większe

W październiku bieżącego roku stwierdzono, że źródła jądra kernela przekroczyły 10mln linii tekstu, z czego ok 7mln to sam kod. W tej liczbie są całe połacie kodu, które były pisane bez żadnego planu testowania, a więc dopiero trzeba je testami pokryć.

Coraz więcej osób jest developerami

Liczba developerów piszących kod jądra zwiększa się również przez cały czas, jak również ludzie którzy kiedyś zajmowali się tym kodem nie robią tego więcej, ani nie mogą pomóc kiedy chodzi o ich kod. Przy takiej sieci osób nie sposób jest korzystać z „tradycyjnych” metod komunikacji i organizacji pracy, takich jak listy dyskusyjne.

Coraz więcej śmieci ludzie próbują do jądra wcisnąć

W ramach projektów studenckich, z własnej inicjatywy, itd., jest też bardzo dużo kodu, który jest zgłaszany do jądra jako nowe moduły, lub jako poprawki. Takie inicjatywy są bardzo cenne dla społeczności open-source, ale muszą być dobrze weryfikowane – nie każda z tych inicjatyw rzeczywiście jest dodatkiem cennym.

C, symbole współdzielone

Kod i architektura jądra są dość „błędogenne” - łatwo jest linkować się z innymi fragmentami jądra, do których normalnie nasz kod nie miałby dostępu. Trzeba również korzystać z nie do końca doprecyzowanej, i dość uciążliwej konwencji nazw, żeby nie nadpisywać cudzych symboli.

To jak to jest?

Reply from actual kernel developer please . . . (Score:5, Interesting)

by EraserMouseMan (847479) on Wednesday October 22, @12:40PM (#25471153)

I'm a developer and was wondering **what kind of testing** is done to verify the code. Do they use unit testing? Regression testing?

I'm just curious because keeping 6+ million lines of code almost completely bug free is pretty amazing.

Re:Reply from actual kernel developer please . . . (Score:5, Funny)

by Anonymous Coward on Wednesday October 22, @12:50PM (#25471303)

Almost completely bug free? What are you smoking?

Parent

LTP – Linux Test Project

Założony przez SGI, utrzymywany przez IBM

LTP został założony przez SGI, jako pakiet 100 testów, w celu zwiększenie niezawodności oraz stabilności Linuxa. Dzisiaj testy są pisane przez developerów z SGI, IBM, OSDL, Bull, Wipro Technologies oraz, jako projekt open-source (wydawany na licencji GPL v2), przez indywidualnych developerów. Liczba testów przekroczyła już 3000.

Kod – ANSI-C, Bash, Perl

Testy w znakomitej większości są pisane w ANSI-C (94% kodu). Reszta to:

- Bash – 5% kodu, np. skrypty uruchamiające dopiero konkretne testy, czy proste testy niewymagające C takie jak
- Perl – ok, 1% kodu, np. kod testujący linkowanie plików

Komplet testów regresyjnych

Jako, że Linux ma swoją, określoną, specyfikację, testy zawarte w pakiecie LTP są testami regresyjnymi, czyli mającymi na celu rozpoznanie regresji w kodzie. Innymi słowy, są to testy pisane po fakcie powstania działającego kodu, mające na celu wykrycie naruszeń dotychczasowej specyfikacji przez nowe zmiany w kodzie.

Testy funkcjonalności, testy systemowe, stress-testy (wysiłkowe?)

Testy zawarte w pakiecie LTP obejmują trzy spośród faz testowania – testowanie funkcjonalności poszczególnych elementów systemu, testy systemowe, oraz stress-testy – testy stabilności.

Testy funkcjonalności

- Poszczególnych komend – ld, ldd, nm, objdump, size, cpio, test cron'a, eject, cp, ln, mkdir, mv, gzip, gunzip, logrotate, mail, tar, unzip. Same testy zazwyczaj w bashu, z ew. programami wspomagającymi.
- Testy poszczególnych funkcjonalności jądra: test modułu acpi, systemu plików, i/o, pseudoterminali, scheduler'a, IPC, pamięci, ładowania modułów, ogromna kolekcja testów syscalli, testy timerów.
- Misc – czyli m.in. testy matematyki, czy test „bugu f00f" (invalid operand with locked CMPXCHG8B instruction)
- Testy sieciowe – testy funkcji sieciowych, w tym duży pakiet poświęcony IPv6.

Testy systemowe

Mimo deklaracji o istnieniu testów systemowych, to nie ma żadnego formalnego. Ew. można traktować cały projekt uruchamiany naraz jako test systemowy.

Stress-testy

Testy badające zachowanie systemu przy maksymalnym obciążeniu:

- System plików – uruchamia na przemian większość dostępnych operacji dyskowych, takich jak read, create, symlink, chown, itd. z góry zadaną (dużą) liczbę razy, w każdym kroku sprawdzając poprawność sytuacji dyskowej, używając, oczywiście, wielu wątków.
- I/O – wielowątkowo czyta z CD, oraz czyta/pisze/formatuje dyskietkę (dyskietkę???)
- malloc/mmap/shmat
- IPC
- scheduler

Stabilizacja

W związku z bardzo wolnym tempem przenosin z kernela 2.3 na 2.4 społeczności Linuxa, IBM Linux Technology Center przygotował bardziej formalny proces „stabilizacji” kernela 2.5. Był to nie tylko plan, ale i deklaracja testowania kernela 2.5 w swoim własnym zakresie, na zadeklarowanym zestawie maszyn IBMa z różnymi dystrybucjami Linuxa zainstalowanymi. LTC nie jest ciałem, które decyduje, kiedy dany kernel staje się stabilny – testowanie które przeprowadzali było raczej metodą na wykrycie jak największej ilości bugów jądra. Proces ten składał się z:

1. Przeprowadzenia wszystkich testów dostępnych w ramach LTP (wliczając w to testy nie zaliczające się do *runall*)
2. Testy integracyjne, podzielone na:
   - Zautomatyzowane:
     - LTP z testowaniem systemu plików
     - dbgrinder dla MySQL
     - pagepoker dla Apache
   - Ręczne:
     - WebSphere z backendem DB2, z dodatkowym DOTS dla generowania ruchu na bazie danych
3. Stress-testy – długie, 30-dniowe maratony stress-testów LTP

Jak to właściwie jest z tą stabilizacją?

Rozwoje kerneli 2.4 i 2.6 były różne między sobą z wielu względów, a zmiany były, w większości, na lepsze. Co takiego ciekawego się stało? Otóż, kilka nowoczesnych metod prowadzenia większych projektów zostało wprowadzonych. Są to, m.in.:

- Użyto aplikacji BitKeeper jako repozytorium kodu. Wcześniej była ona używana przez wielu developerów jądra dla ich własnej pracy, z kolei przy pracy nad kernelami 2.5 i 2.6, na początku na próbę, potem na stałe, zaczął jej używać sam Linus Torvalds.
- Nightly regression testing – dzięki centralnemu repozytorium zmian, można było wykonywać regularne, co 24 godziny, testy regresyjne. Dzięki temu można było szybko wykrywać wiele błędów – a im szybciej wykryty bug, tym mniej kosztuje naprawienie go, choćby dlatego, że zmiana jest wciąż nowa, i wciąż w głowie autora danej zmiany jest ona świeża, jak i dlatego, że na tym błędzie nic dalej nie było budowane.
- STP (Scalable Testing Platform) – OSDL udostępniło społeczności developerów platformę, na której mogli oni testować swoje nowe zmiany.
- Kolejny wkład OSDL - uruchomiło ono platformę do śledzenia błędów.
- Wiele innych, pomniejszych projektów, które śledziły mniejsze lub większe błędy, listy plików które wymagają „sprzątania”, itp.

Autor tej części prezentacji był bardzo ciekaw, w którym momencie, właściwie, kernel 2.5 staje się kernelem 2.6 – tzn., jak się, właściwie, kończy proces „stabilizacji”. Wygląda na to, że wtedy, kiedy Linux Torvalds da na to swoje błogosławieństwo.

Narzędzia

LTP utrzymuje również kilka narzędzi, użytecznych do testowania. Należy do nich gcov (więcej o nim poniżej), jak również lcov – zbiór skryptów Perl'a, które z tekstowego wyjścia gcov'a produkują bardziej przyjazne użytkownikowi wyjście w postaci pliku HTML, gdzie dane te ujęte są w tabeli.

Device Simulator Framework

Device Simulator Framework jest narzędziem, udostępnianym w pakiecie LTP, reklamowanym jako narzędzie umożliwiające, czy usprawniające proces testowania kernela. Dokładniej, służy ono do komunikacji między userspace i kernelspace, uruchamianie z poziomu userspace funkcji w kernelspace, przekazywania parametrów między nimi, itd.
Brzmi nieźle?
Brzmi super. Device Simulator Framework jest tak naprawdę przykładem (który, oczywiście, można wykorzystywać w swoich testach), jak napisać dość prosty program do komunikacji z jądrem, oraz – również dość prosty – moduł kernela obsługujący określone urządzenie. Komunikacja odbywa się poprzez to urządzenie, wykorzystując funkcję ioctl.

LSB - Linux Standard Base

Specyfikacja

Projekt Linux Standard Base dostarcza specyfikacje (podzielone ze względu na składowe systemu), opisujące systemowe interfejsy. Celem takiego projektu jest zachowanie portowalności między różnymi implementacjami podstawowych bibliotek, instalacjami systemów, czy między różnymi dystrybucjami. Specyfikacje dzielą się na następujące kategorie:

Core

Kategoria core opisuje format podstawowych składników systemu. Do takich należą:
- Format plików ELF (Executable and Linking Format)
- Bibliotek bazowych: libc, libm, libpthread, libgcc itd.
- Bibliotek użytkowych – np. libncurses
- Zbiór podstawowych programów, np. ls, sync, czy md5sum, które powinny należeć do każdego systemu
- Środowisko, hierarchia katalogów, /dev – katalog z plikami do obsługi urządzeń, /etc – katalog z konfiguracją dla danego hosta, itd.
- Inicjację systemu – kolejne runlevel'e, co powinno być podczas każdego z nich obsłużone, jakie usługi muszą zostać uruchomione i kiedy, itd.
- Schemat użytkowników – superużytkownik, schemat grup, numerację, jak to wszystko jest zapisane

Itd.

C++

W zakres tej specyfikacji wchodzi reprezentacja danych w plikach obiektowych C++ oraz standard mapowania symboli.

Desktop

Ta specyfikacja za to obejmuje biblioteki używane przez system X11. Są to m.in. biblioteki libX11 i pokrewne, biblioteka libGL, biblioteki dla Qt, XML, JPEG, GTK+ itd.

Certyfikacja

Linux Foundation oferuje certyfikacje dla produktów oraz dystrybucji, które spełniają standard LSB. Dla potrzeb testowania zgodności, istnieje zbiór testów, które to potwierdzają. Niewiele produktów tak naprawdę utrzymuje tę ceryfikację – należą do nich m.in. niektóre wersje Mandrake, Mandriva, Red Hat, SUSE oraz Ubuntu Linuxa.

Application Testkit

Application Testkit to zbiór narzędzi do testowania zgodności aplikacji z LSB. Główną jego funkcją jest sprawdzenie, jakie są zależności danej aplikacji, czy są zgodne ze standardem, w jakim stopniu, oraz porównuje z ok. 30 dystrybucjami zgodnymi z LSB.

Distribution Testkit

Distribution Testkit z kolei jest zbiorem narzędzi i testów do sprawdzania zgodności danej dystrybucji Linuxa z LSB. Na ten proces składa się m.in. sprawdzenie obecności wszystkich wymaganych bibliotek, weryfikacja systemu plików, itp., itd. Użytkownik może sam wybrać, jakie części specyfikacji mają być testowane – może np. przetestować tylko zgodność ze specyfikacją core, albo pominąć zgodność z printing.

TETware

Aplikacje testujące w pakiecie dostarczanym przez Linux Base korzystają z platformy Test Environment Toolkit. Umożliwia ona testowanie rozproszone, zarówno na platformach UNIX, jak i Windows NT, ma rozszerzenia dla wielu języków programowania oraz skryptowych.

Filozofia testowania

Biała skrzynka...

Wiele zdaje się sugerować, że testowanie jądra odbywa się na zasadzie białej skrzynki. Spójrzmy choćby analizę pokrycia kodu jądra testami, dokonywaną przez gcov/lcov. Jednym z celów projektu LTP jest pokrycie jak największej części jądra testami, tak, aby każdy wiersz kodu został wykonany (potencjalnie wielokrotnie) podczas testowania. Ale czy to jest rzeczywiście biała skrzynka? Spójrzmy na definicję białej skrzynki. W tej metodzie zazwyczaj testuje się:

- potoki kontrolne (control flow)
- przepływ danych
- gałęzie wykonania

Podczas gdy samo sprawdzenie pokrycia kodu testami sugeruje spełnienie ostatniego z tych punktów, to przetestowanie pierwszego i drugiego wymaga znajomości samego kodu. Kolejną przesłanką może być to, że metodę białej skrzynki wykorzystuje się zazwyczaj w unit testach.

... czy czarna skrzynka?

Duża część testów jądra powstawała znacznie później, niż sam testowany kod. Już to, samo w sobie, jest dość poważną przesłanką do tego, żeby te testy traktować jak testy metodą czarnej skrzynki – przecież, choćby z nazwy, metoda białej skrzynki oznacza dobrą znajomość i zrozumienie testowanego kodu. Kolejnym powodem, dla którego aktualnie istniejące testy jądra należałoby traktować jako testowanie metodą black box jest to, że do kodu testowanego często trzeba się przedostawać przez całe warstwy kodu jądra, który stanowi interfejs dla danych metod schowanych gdzieś głęboko, w niskiej warstwie, blisko sprzętu. Trudno tutaj mówić o testowaniu przepływu danych, czy o testowaniu control flow – co prawda wiemy, że dany kod został wykonany określoną liczbę razy, za to nie wiemy, z jakimi danymi wejściowymi, i co powstało na wyjściu. Może te dane za każdym razem były takie same?

Unit testing vs. fault injection

Popularną techniką stosowaną przy testowaniu jądra jest fault injection. Polega ona na zmienianiu fragmentów kodu z poprawnych na błędne, żeby sprawdzić, jak się zachowuje reszta kodu w obliczu błędnych danych. Na tyle często się ją stosuje, że drugim wynikiem wyszukiwania na Google hasła „fault injection” jest artykuł dot. stosowania tej techniki w kernelu. Jest to technika jednorazowa, stosowana podczas pisania danego modułu. Jest ona stawiana jako alternatywa przy rozwoju kernela dla unit testów. Nie istnieje żaden mechanizm utrzymywania unit testów dla kernela.

Pokrycie kodu testami

Czym jest pokrycie kodu testami?

Analiza pokrycia kodu testami pozwala nam określić, jaka część kodu aplikacji jest wykonywana podczas testów. Ta forma testowania bezpośrednio dotyka kodu, zatem jest to zastosowanie metody „białej skrzynki”.

Do czego służy?

Otrzymane wyniki pozwalają nam ocenić jakość testów i usprawnić proces przeprowadzania testów aplikacji.

Kryteria

Istnieje wiele kryteriów, pod kątem którym można analizować pokrycie kodu testami:

- Pokrycie funkcji (function coverage)
- Pokrycie warunków logicznych (condition coverage)
- Pokrycie wejść/wyjść (entry/exit coverage)
- Pokrycie ścieżek (path coverage)
- Pokrycie instrukcji (statement coverage)
- Pokrycie gałęzi (branch/decision coverage)

Pokrycie funkcji

- ile razy każda funkcja została wykonana?

Pokrycie warunków logicznych

- czy każde podwyrażenie logiczne wylicza się i do true i do false?
- nie musi być równoważne z pokryciem gałęzi

Pokrycie wejść/wyjść

- czy każdy rodzaj wywołania funkcji i każdy rodzaj powrotu z niej został wykonany?

Pokrycie ścieżek

- czy każda możliwa ścieżka w kodzie została wykonana?
- Testowanie pełnego pokrycia ścieżek jest niemożliwe w przypadku prawdziwych aplikacji. Jeśli w kodzie występuje n instrukcji warunkowych, to liczba ścieżek może wynieść nawet 2^n. Dla pętli może istnieć nieskończenie wiele ścieżek. Ponadto istnieją ścieżki, które nigdy nie zostaną wykonane. Udowodniono, że znalezienie algorytmu wykrywającego takie ścieżki jest równoważne z rozwiązaniem problemu stopu.
- W praktyce stosuje się pokrycie ścieżek bazowych. W tym celu wyznacza się klasy ścieżek, które między sobą różnią się tylko ilością wykonanych obrotów pętli. Wynikiem jest pokrycie klas ścieżek.

Pokrycie instrukcji

Dzielimy kod na bloki – fragmenty kodu pomiędzy instrukcjami warunkowymi.
Każda instrukcja z jednego bloku będzie wykonana tyle samo razy.

Niestety, nawet jeśli otrzymamy w wyniku 100% pokrycia nie możemy być pewni, że nasz program nie zawiera błędów. Ilustruje to poniższy przykład:

```
int *iptr = NULL;
if (warunek)
        iptr = &i;
*iptr = j*10;
```

W przypadku, gdy warunek będzie nieprawdziwy iptr będzie wskazywać na NULL,

a program zgłosi wyjątek.

Nawet 100% pokrycia instrukcji nie daje gwarancji, że nasza aplikacja będzie wolna od błędów.

Pokrycie gałęzi

Analizujemy rozgałęzienia w kodzie, wyznaczamy możliwe ścieżki. Sprawdzamy, czy każdy warunek logiczny podczas wykonania programu wylicza się do true i do false.
W wyniku otrzymujemy ile razy każda gałąź została wykonana.

Ta metoda także może zawodzić, jeśli w testowanych programie w instrukcjach warunkowych występują złożone wyrażenia. Żeby mieć pewność, że nasz program nie zawiera błędów, powinniśmy przetestować wszystkie kombinacje.

```
struct somestructure *p = NULL;
if(i == 0 || (j == 1 && p->j == 10))
        printf("got here\n");
```

Zauważmy, że w powyższym przykładzie istnieją ścieżki wyliczające się i do true i false, które nie uwidocznią potencjalnego błędu.

Wynik 100% pokrycia gałęzi kodu nie daje gwarancji braku błędów w kodzie.

Martwy kod

Kod, który nigdy się nie wykona (np. umieszczony po return); nie wszystkie kompilatory go wykrywają. Jeśli pojawi się gdziekolwiek w programie, nie uzyskamy przy testowaniu 100% pokrycia kodu.

Gcov

Gcov to narzędzie, które dostarcza statystyk na temat pokrycia kodu, a dokładniej pokrycia instrukcji i gałęzi.

Użycie gcov wymaga skompilowania programu z dwoma opcjami gcc: -fprofile-arcs -ftest-cover-age. Zapisuje on wtedy dodatkowe informacje w budowanych plikach oraz tworzy dodatkowe pliki wymagane przez gcov. Uruchomienie programu powoduje zapisanie informacji diagnostycznych. Dla każdego pliku źródłowego skompilowanego z opcją -fpro-file-arcs powstaje towarzyszący plik .da.

Uruchomienie gcov z nazwą pliku źródłowego jako parametrem tworzy opis kodu źródłowego z podaniem częstotliwości wykonania każdej jego linii. Np. jeśli program nazywa się tmp.c to użycie gcov, oraz plik tmp.c.gcov mogą wyglądać tak:

```
$ gcc -fprofile-arcs -ftest-coverage tmp.c
$ a.out
$ gcov tmp.c
90.00% of 10 source lines executed in file tmp.c
Creating tmp.c.gcov.
```

```
        -:    0:Source:tmp.c
        -:    0:Object:tmp.bb
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:int main (void)
        1:    4:{
        1:    5: int i, total;
        -:    6:
        1:    7: total = 0;
        -:    8:
       11:    9: for (i = 0; i < 10; i++)
       10:   10:   total += i;
        -:   11:
        1:   12: if (total != 45)
    #####:   13:   printf ("Failure\n");
        -:   14: else
        1:   15:   printf ("Success\n");
        1:   16: return 0;
        -:   17:}
```

Oczywiście, gcov może być uruchamiany z różnymi flagami. Po szczegóły odsyłam do manuala.

Gcov–kernel

Aby móc przetestować jądro linuxa należy zainstalować gcov-kernel patch, a także przekompilować i zainstalować jądro. Stworzony zostanie moduł, który będzie zbierał statystyki i umieszczał je w /proc/gcov. Zmniejsza to, niestety, szybkość kernela o ok. 10%.

Po zainstalowaniu patcha należy ustawić opcje Gcova:

```
CONFIG_GCOV_PROFILE=y  : include basic kernel support for GCOV
CONFIG_GCOV_ALL=y      : profile all of the kernel source
CONFIG_GCOV_PROC=y/m   : provide proc fs entry to access GCOV data
```

Jeśli CONFIG_GCOV_PROC jest ustawione na 'm', to moduł gcov-proc musimy sami załadować po uruchomieniu systemu, poleceniem
modprobe gcov-proc

Utworzony zostanie /proc/gcov. Statystyki będą tam gromadzone. Jeśli chcemy wyzerować statystyki, musimy zapisać 0 do odpowiedniego pliku, np.:

```
echo 0 > /proc/gcov/kernel/signal.da
```

Wszystkie niezbędne źródła i informacje można znaleźć na stronie: http://sourceforge.net/projects/ltp.

Lcov

Lcov to narzędzie do wyświetlania danych z gcov w sposób bardziej przyjazny użytkownikowi. Może generować strony html z wynikami.

Jak tego używać?

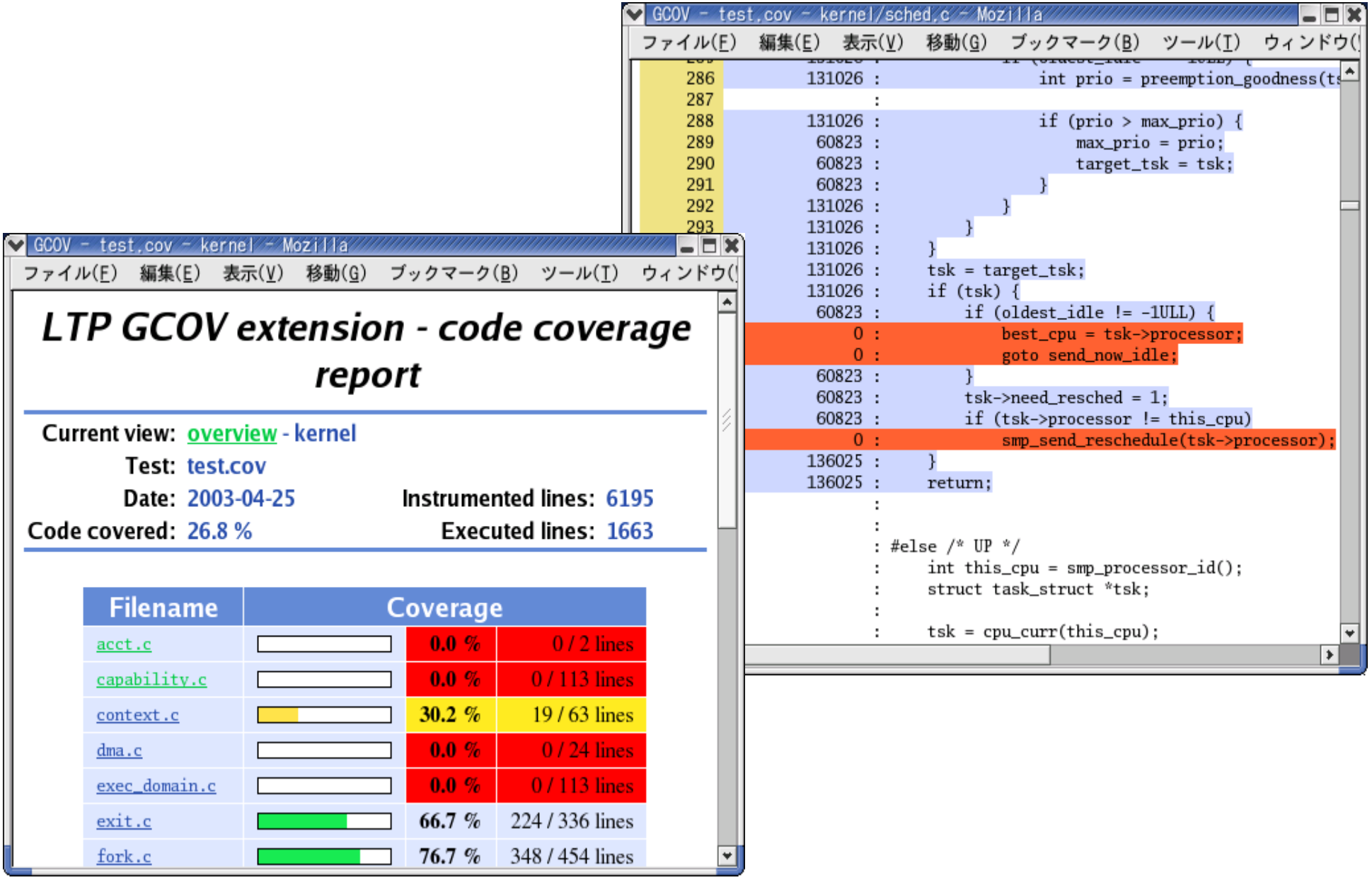
Zwykła aplikacja

```
// uruchamiamy aplikację skompilowaną z odpowiednimi flagami; kończymy jej działanie
lcov --directory [dirname] -c -o application.info
genhtml -o [outputdir] application.info
```

Kernel

```
cd lcov-kernel
lcov -c -o kernel.info  // to polecenie wygeneruje plik kernel.info
genhtml kernel.info -o kernel-lcov
```

Przykłady


GCOV - test.cov - kernel/sched.c - Mozilla
ファイル(F) 編集(E) 表示(V) 移動(G) ブックマーク(B) ツール(T) ウィンドウ(
286 131026 : int prio = preemption_goodness(ts
287 :
288 131026 : if (prio > max_prio) {
289 60823 : max_prio = prio;
290 60823 : target_tsk = tsk;
291 60823 : }
292 131026 : }
293 131026 : }
131026 : }
131026 : tsk = target_tsk;
131026 : if (tsk) {
60823 : if (oldest_idle != -1ULL) {
0 : best_cpu = tsk->processor;
0 : goto send_now_idle;
60823 : }
60823 : tsk->need_resched = 1;
60823 : if (tsk->processor != this_cpu)
0 : smp_send_reschedule(tsk->processor);
136025 : }
136025 : return;
: #else /* UP */
: int this_cpu = smp_processor_id();
: struct task_struct *tsk;
: tsk = cpu_curr(this_cpu);
GCOV - test.cov - kernel - Mozilla
ファイル(F) 編集(E) 表示(V) 移動(G) ブックマーク(B) ツール(T) ウィンドウ(
LTP GCOV extension - code coverage report
Current view: overview - kernel
Test: test.cov
Date: 2003-04-25
Instrumented lines: 6195
Code covered: 26.8 %
Executed lines: 1663
Filename
Coverage
acct.c 0.0 % 0 / 2 lines
capability.c 0.0 % 0 / 113 lines
context.c 30.2 % 19 / 63 lines
dma.c 0.0 % 0 / 24 lines
exec_domain.c 0.0 % 0 / 113 lines
exit.c 66.7 % 224 / 336 lines
fork.c 76.7 % 348 / 454 lines

Linux Kernel Performance

Projekt mający na celu ulepszanie działania Linuxa. Na ich stronie (http://kernel-perf.sourceforge.net/) zamieszczono wyniki testów kolejnych wersji jądra Linuxa, począwszy od 2.6.18, dla różnych architektur.

Zestaw testów pokrywa najważniejsze komponenty kernela – zarządzanie pamięcią wirtualną, szeregowanie procesów, mechanizm dostępów do dysku, system plików, sieć, sterowniki urządzeń.

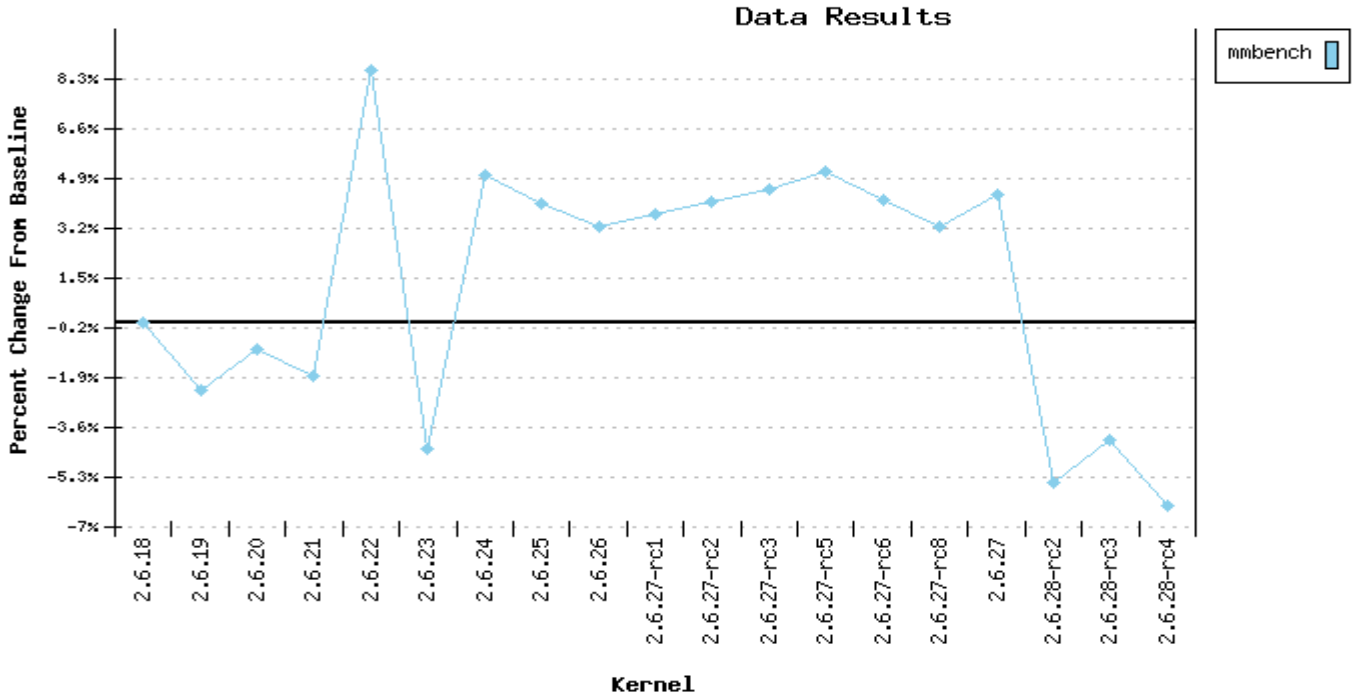
Lista narzędzi:

- **volanomark** - mierzy efektywność działania serwera i sieci. W wyniku zwraca średnią liczbę wiadomości przesłanych przez serwer na sekundę.
  http://www.volano.com/benchmarks.html
- **lmbench** - suite of portable micro-benchmarks for UNIX, measures key system performance
  http://www.bitmover.com/lmbench
- **aiostress** - mierzy wydajność asynchronicznego I/O z użyciem filesystemu i bez.
  ftp://ftp.suse.com/pub/people/mason/utils/aio-stress.c
- **netperf** – narzędzie do testowania sieci.
  http://www.netperf.org/netperf/NetperfPage.html
- **aim** - zestaw testów mierzący wydajność sieci, pamięci i operacji dyskowych
  http://sourceforge.net/projects/re-aim-7
- **fileio** - jeden z komponentów sysbench. Testy wydajnościowe operacji I/O.
  http://sysbench.sourceforge.net
- **iozone** - narzędzie do testowania systemów plików. W tym celu testowane są następujące funkcje: read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read, pread , mmap, aio_read, aio_write.
  http://www.iozone.org
- **vmregress** - moduł, który sprawdza wirtualny system plików.
  http://www.skynet.ie/~mel/projects/vmregress

- **JAVA*** - narzędzie biznesowe do testów o standardzie przemysłowym.
- **cpu-int** - testowanie obciążenia procesorów o standardzie przemysłowym.
- **cpu-fp** - testowanie obciążenia procesorów o standardzie przemysłowym.
- **mmbench** - testowanie zarządzania pamięcią.
- **tiobench** - testuje operacje I/O z wykorzystaniem wątków.
  http://sourceforge.net/projects/tiobench/

Przykładowy wynik testów przy użyciu mmbench.

Hardware:
CPU type: Intel Itanium® 2 processor
CPU frequency: 1600Mhz
Description: 1.6 Ghz: 4 Sockets
Memory: 4096Mb
Disk type: SCSI
Number of disks: 1
Disk size: 18GB



Bootchart

Ciekawe narzędzie pozwalające dogłębnie przyjrzeć się procesowi uruchamiania systemu. W wygenerowanym przez program raporcie poza standardowymi informacjami w postaci chronologicznego spisu wszystkich modułów czy kompletnego czasu startu systemu znaleźć można także informacje dotyczące używanej dystrybucji, stopniu wykorzystania zasobów procesora i dysku oraz dokładne informacje na temat jądra wraz z opcjami, z którymi zostało uruchomione.

Utworzony raport przekształcany jest potem przez aplikację Javy albo za pośrednictwem strony internetowej na plik graficzny w formacie PNG, SVG bądź EPS.

Jak to działa?

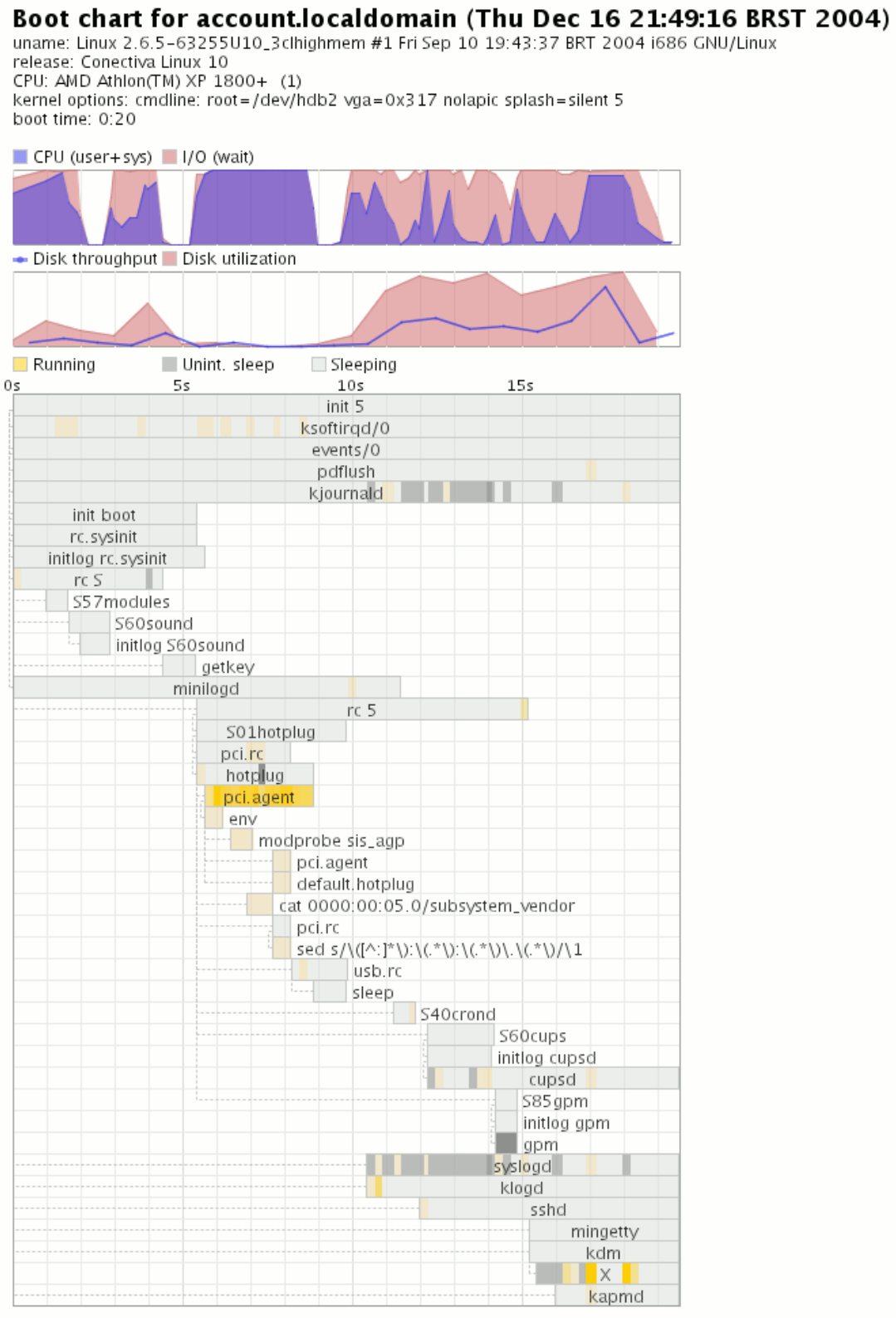
Boot logger (/sbin/bootchartd) jest uruchamiany przez jądro zamiast /sbin/init. Aby to osiągnąć należy zmodyfikować listę poleceń dla kernela w GRUB albo LILO, np.:

/boot/grub/menu.lst
[...]
title Fedora Core (2.6.10) - bootchart
    root (hd0,1)
    kernel /vmlinuz-2.6.10 ro root=/dev/hda1 **init=/sbin/bootchartd**
    initrd /initrd-2.6.10.img

Skrypt instalujący i pakiet RPM będą próbowały automatycznie dodać boot loader.

Boot logger uruchomi się w tle i bezzwłocznie odpali /sbin/init, który zacznie wykonywać się jak podczas zwykłego bootowania.

Zbieranie danych



Podczas bootowania systemu partycja root jest załadowana w trybie tylko do odczytu, więc bootchartd musi przechowywać dane w pamięci, korzystając z wirtualnego systemu plików (tmpfs).

Jak tylko zostanie zamontowany system plików /proc, bootchartd zacznie zbierać dane z różnych plików.

/proc/stat	system-wide CPU statistics: user, system, IO and idle times
/proc/diskstats	system-wide disk statistics: disk utilization and throughput (only available in 2.6 kernels)
/proc/[PID]/stat	information about the running processes: start time, parent PID, process state, CPU usage, etc.

Zawartość tych plików jest dołączana do odpowiednich plików log, domyślnie co 0.2 sekundy.

Logger orientuje się, że proces bootowania zakończył się, obserwując pewne procesy. Np. jeśli uruchamiamy system w trybie 5 (multi-user graphical mode), bootchartd będzie szukał gdmgreeter, kdm_greet, itp. Jak tylko zauważone zostanie, że któryś z tych procesów jest uruchomiony logger przestanie zbierać dane. Uzyskane statystyki zapisze w /var/log/bootchart.tgz.

Uwaga co do dokładności zebranych danych:
Dane o istnieniu procesów, które bardzo szybko skończyły swoje działanie mogą zostać zgubione. Jeśli taki proces wykonał fork, to może się okazać, że odtworzenie drzewa procesów na podstawie logów jest niemożliwe. Wtedy te „osierocone” procesy mogą zostać niewłaściwie przydzielone do grupy podczas tworzenia wykresu.

Jeśli zależy nam na dużej dokładności w zbieraniu informacji o procesach, możemy temu zaradzić. Wystarczy zainstalować pakiet psacct albo acct i odpowiednio skonfigurować jądro.
Więcej informacji pod adresem: http://www.bootchart.org/docs.html

Jak uzyskać wykres?

Uruchomić bootchart

Aby ta metoda zadziałała trzeba mieć zainstalowany JPackage.
Jeśli nie mamy JPackage, ale mamy zainstalowany Java Development Kit:

```
java -jar bootchart.jar
```

Wykorzystać stronę internetową - http://www.bootchart.org

Można otrzymać wykres uruchamiając:

```
curl --form format=svg --form log=@/var/log/bootchart.tgz \   http://render.bootchart.org:8080/bootchart/render > bootchart.svgz
```

Można zamienić svg/bootchart.svgz na png/bootchart.png albo eps/bootchart.eps.gz, w zależności od tego, w jakim formacie chcemy otrzymać wynik.

Więcej informacji na http://www.bootchart.org/

Przykładowe testy jądra

Poprawnościowe:

- inode01

Poprawność tworzenia plików i katalogów. Tworzy rozbudowane drzewo i sprawdza, czy faktycznie jest takie, jakie utworzył. Pozwala wykryć błędy w zapisach do inode-ów.

- fs_full

Sprawdza poprawność zachowania się systemu plików przy całkowitym zapełnieniu (funkcje zapisujące nowe dane powinny zwracać ENOSPC). Test składa się z trzech elementów, opisanych dalej. Test własny.

Wydajnościowe:

- inode02

Podobny do inode01, ale uruchamia wiele procesów jednocześnie, żeby sprawdzić, czy przy dużym obciążeniu również działa poprawnie.

- gf14, gf15

Testy sprawdzające równoległe dopisywanie do plików w wielu procesach naraz.

Przykładowe testy systemu plików

Zapełnianie urządzenia samymi katalogami

```
for i in `seq 100`; do
    myfuncall mkdir mount/$i || RC=$?
    if [ $RC -eq 2 ]; then
        RC=0; break
    else if [ $RC -ne 0 ]; then
            tst_brkm TFAIL NULL "Test #1: Error other than ENOSPC"
            return $RC
        fi
    fi
    for j in `seq 100`; do
        myfuncall mkdir mount/$i/$j || RC=$?
        if [ $RC -eq 2 ]; then
            RC=0; break
        else if [ $RC -ne 0 ]; then
                tst_brkm TFAIL NULL "Test #1: Error creating directory other than ENOSPC"
                return $RC
            fi
        fi
        for k in `seq 100`; do
            myfuncall mkdir mount/$i/$j/$k || RC=$?
            if [ $RC -eq 2 ]; then
                RC=0; break
            else if [ $RC -ne 0 ]; then
                    tst_brkm TFAIL NULL "Test #1: Error creating directory other than ENOSPC"
                    return $RC
                fi
            fi
        done
    done
done
```

Zapełnianie urządzenia samymi pustymi plikami

```
for i in `seq 1000000`; do
   myfuncall touch mount/$i || RC=$?
   if [ $RC -eq 2 ]; then
      RC=0
      break
   else
      if [ $RC -ne 0 ]; then
         tst_brkm TFAIL NULL "Test #3: Error other than ENOSPC"
         return $RC
      fi
   fi
done
```

Zapełnianie urządzenia strukturą katalogów i pustych plików

```
for i in `seq 100`; do
    myfuncall mkdir mount/$i || RC=$?
    if [ $RC -eq 2 ]; then
        RC=0; break
    else if [ $RC -ne 0 ]; then
            tst_brkm TFAIL NULL "Test #3: Error other than ENOSPC"; return $RC
        fi
    fi
    for j in `seq 100`; do
        myfuncall mkdir mount/$i/$j || RC=$?
        if [ $RC -eq 2 ]; then
            RC=0; break
        else if [ $RC -ne 0 ]; then
                tst_brkm TFAIL NULL "Test #3: Error other than ENOSPC"; return $RC
            fi
        fi
        for k in `seq 100`; do
            myfuncall touch mount/$i/$j/$k || RC=$?
            if [ $RC -eq 2 ]; then
                RC=0; break
            else if [ $RC -ne 0 ]; then
                    tst_brkm TFAIL NULL "Test #3: Error other than ENOSPC"; return $RC
                fi
            fi
        done
    done
done
```

```
Running tests.......
<<<test_start>>>
tag=fs_full stime=1229395242
cmdline="fs_full.sh"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
incrementing stop
(...)
test01      0  INFO  :  Test #1: create lots of directories
No space left on device, all correct
No space left on device, all correct
No space left on device, all correct
test01      1  PASS  :  Test #1: filled the filesystem.
test02      0  INFO  :  Test #2: create lots of files
No space left on device, all correct
test02      2  PASS  :  Test #2: filled the filesystem.
test03      0  INFO  :  Test #3: create lots of directories and files
No space left on device, all correct
No space left on device, all correct
No space left on device, all correct
test03      3  PASS  :  Test #3: filled the filesystem.
<<<execution_status>>>
duration=443 termination_type=exited termination_id=0 corefile=no
cutime=1477 cstime=35653
<<<test_end>>>
INFO: pan reported all tests PASS
LTP Version: LTP-20081130
```

LTP GCOV extension - code coverage report

Current view: directory - fs/ext3
Test: fs_full.info
Date: 2008-12-16
Code covered: 43.8 %

Instrumented lines: 6323
Executed lines: 2768

Filename	Coverage		
acl.c		6.1 %	16 / 262 lines
acl.h		0.0 %	0 / 12 lines
balloc.c		68.6 %	386 / 563 lines
dir.c		80.4 %	176 / 219 lines
ext3_jbd.c		66.7 %	20 / 30 lines
file.c		62.5 %	15 / 24 lines
fsync.c		78.6 %	11 / 14 lines
hash.c		60.5 %	46 / 76 lines
ialloc.c		68.8 %	234 / 340 lines
inode.c		58.3 %	687 / 1178 lines
ioctl.c		2.8 %	4 / 144 lines
namei.c		69.3 %	727 / 1049 lines
resize.c		0.0 %	0 / 454 lines
super.c		31.2 %	408 / 1306 lines
symlink.c		100.0 %	4 / 4 lines
xattr.c		4.4 %	26 / 585 lines
xattr_security.c		32.0 %	8 / 25 lines
xattr_trusted.c		0.0 %	0 / 17 lines
xattr_user.c		0.0 %	0 / 21 lines

```
64                 : int ext3_check_dir_entry (const char * function, struct inode * dir,
65                 :                           struct ext3_dir_entry_2 * de,
66                 :                           struct buffer_head * bh,
67                 :                           unsigned long offset)
68           28001 : {
69           28001 :         const char * error_msg = NULL;
70           56002 :         const int rlen = ext3_rec_len_from_disk(de->rec_len);
71                 :
72           28001 :         if (rlen < EXT3_DIR_REC_LEN(1))
73               0 :                 error_msg = "rec_len is smaller than minimal";
74           28001 :         else if (rlen % 4 != 0)
75               0 :                 error_msg = "rec_len % 4 != 0";
76           28001 :         else if (rlen < EXT3_DIR_REC_LEN(de->name_len))
77               0 :                 error_msg = "rec_len is too small for name_len";
78           28001 :         else if (((char *) de - bh->b_data) + rlen > dir->i_sb->s_blocksize)
79               0 :                 error_msg = "directory entry across blocks";
80           56002 :         else if (le32_to_cpu(de->inode) >
81                 :                         le32_to_cpu(EXT3_SB(dir->i_sb)->s_es->s_inodes_count))
82               0 :                 error_msg = "inode out of bounds";
83                 :
84           28001 :         if (error_msg != NULL)
85               0 :                 ext3_error (dir->i_sb, function,
86                 :                         "bad entry in directory #%lu: %s - "
87                 :                         "offset=%lu, inode=%lu, rec_len=%d, name_len=%d",
88                 :                         dir->i_ino, error_msg, offset,
89                 :                         (unsigned long) le32_to_cpu(de->inode),
90                 :                         rlen, de->name_len);
91           28001 :         return error_msg == NULL ? 1 : 0;
92                 : }
```