Automatyczne testowanie jądra Linuksa

P. Bednarz J. Iwaniuk M. Skrzypczak B. Wołowiec

21 grudnia 2008

Tytułem wstępu

Linus's Law:

given enough eyeballs, all bugs are shallow

Edsger Wybe Dijkstra:

testing shows the presence, not the absence of bugs

Najsłynniejszy robak świata

Wprowadzenie do użycia wspomnianego terminu przypisywane jest pani admirał Grace Hopper. Podczas prac prowadzonych – według różnych źródeł w 1945 lub 1947 r. – z prymitywnym komputerem Harvard Mark II operator stwierdził jego nieprawidłowe działanie, a po poszukiwaniach przyczyny znalazł pomiędzy gołymi przewodami przekaźnika drobnego insekta (ang. bug) – ćmę, która powodowała spięcie. Owad został usunięty i wklejony do dziennika. Dziennik obecnie znajduje się w Naval Surface Warfare Center Computer Museum w Dahlgren w stanie Wirginia (USA).

Rodzaje błędów

Mamy cztery podstawowe rodzaje błędów w Linuxie:

- program zwraca błędne wyniki,
- program się zawiesza (lockup),
- oops - użytkownik jest informowany o nieprawidłowościach w działaniu jądra,
- kernel panic - załamanie systemu.

Testy możemy dzielić na różne sposoby

- w zależności od fazy, na której odbywa się testowanie: testowanie wymagań, modułów, interakcji między modułami, działania całego systemu,
- szklanoskrzynkowe i czarnoskrzynkowe, czyli na takie, w których testujemy oprogramowanie zakładając znajomość kodu i wręcz przeciwnie - traktując aplikację jako maszynę przyjmującą dane wejściowe i zwracającą komunikaty,
- poprawnościowe, badające sensowność zwracanych wyników oraz wydajnościowe - sprawdzające, jak szybko są wykonywane zadania stawiane przed aplikacją.

Testy statyczne

Testy statyczne dotyczą analizy kodu poprzez przeglądanie go lub testy automatyczne (analiza syntaktyczna podczas kompilacji, Sparse), bez uruchamiania aplikacji. Statycznie możemy sprawdzać zarówno poprawność syntaktyczną, jak i semantyczną.

Testy dynamiczne

Testy dynamiczne polegają na uruchomieniu programu i sprawdzaniu, czy program zwraca poprawne wyniki dla danych dopuszczalnych w specyfikacji i czy odpowiednio reaguje na dane niepoprawne. Testy takie przeprowadzane są często przez ludzi niezwiązanych z informatyką. Niewskazane jest wręcz, aby takie testy przeprowadzały osoby, które daną funkcjonalność zaprogramowały.

Jakość testów możemy mierzyć na wiele sposobów

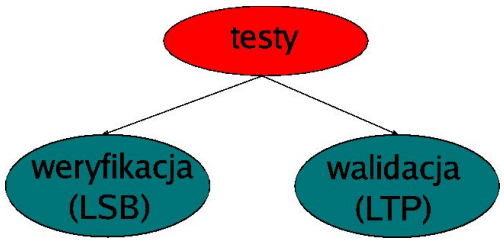
- procentem znajdowanych błędów w danej liczbie linii kodu (wskaźnik zależy od ilości wcześniej przeprowadzonych testów, jakości kodu itd. - np w kodzie Linux'a wykrywa się średnio 0.17 błędu na 1000 linii kodu - jest to ok. 25 razy mniej, niż wykrywa się ich w komercyjnym oprogramowaniu),
- częścią kodu, jaki testy pokrywają,
- procentem możliwych parametrów wejściowych,
- niezależnością od czynników zewnętrznych (interakcje z programami działającymi w tle itd.).

Korzyści z wykrycia błędu wcześnie

Jak wykazują badania, im wcześniej wykryty zostaje błąd, tym mniejsze są koszty jego wykrycia (np. wykrycie błędu na poziomie projektowania systemu to w 2004 był przeciętnie koszt ok. 5 dolarów, podczas gdy wykrycie tego samego błędu na poziomie testów systemowych kosztować nas może ok. 50 dolarów).

Weryfikacja i walidacja

Testowanie oprogramowania to proces mający na celu podniesienie jakości wytworzonego oprogramowania. Proces ten dzielimy na weryfikację i walidację. Weryfikacja to sprawdzenie zgodności ze specyfikacją, podczas gdy walidacja sprawdza zgodność specyfikacji i samej aplikacji z wymaganiami użytkownika.



Co to jest?

Rozpoczęty przez SGI, obecnie wspierany przez IBM. Projekt mający na celu dostarczenie bogatego zestawu testów regresyjnych służących testowaniu działania zarówno jądra jak i innych części Linux'a. Jest to zestaw testów jednostkowych, stress-testów i testów systemowych. Używa się go do przyspieszenia procesu stabilizacji wersji Linux'a oraz do sprawdzania poprawności działania systemu po wprowadzeniu zmiany w jądrze.

Ballista

- wywołuje *syscall*'e w nietypowych konfiguracjach i z nietypowymi danymi wejściowymi,
- poszukuje błędów obsługiwanych w nieprawidłowy sposób ...
- ... oraz błędnych kodów powrotu,
- zestaw ten nastawiony jest na sprawienie, że system przestanie w ogóle działać.

File system 1

- sprawdzenie integralności systemu plików. Tworzy plik z danymi określonego lub losowego rozmiaru i kopiuje go na losową głębokość w danym systemie plików. Następnie oba pliki są porównywane i od ich identyczności zależy wynik testu. Procedura jest powtarzana 10 razy,
- test warunków brzegowych (maksymalna ilość linków do jednego pliku, pliki maksymalnego rozmiaru),
- zarządzanie uprawnieniami.

File system 2

- działanie w środowisku wielowątkowym - tworzy wiele wątków, które próbują jednocześnie tworzyć i otwierać pliki. Sprawdzane jest poprawne działanie mutexów,
- działanie w środowisku wieloprocesorowym - tworzy i usuwa pliki za pomocą wielu procesów działających w tle. Można modyfikować parametry: ilość podkatalogów, które ma stworzyć, ilość plików do stworzenia w każdym podkatalogu, ilość cykli utwórz/skasuj.

IO

- zestaw testów mających testować asynchroniczne żądania wejśia/wyjścia,
- testuje IO dla dysku, stacji CD i stacji dyskietek,
- dane są generowane losowo,
- następnie odbywa się zapis pliku na odpowiedni nośnik.

Pisanie do plików zwykle odbywa się w ten sposób, że wybierane jest jakieś ziarno do generowania losowych danych i plik jest zapełniany losową zawartością. Następnie inicjalizuje się generator tym samym ziarnem, plik jest ponownie otwierany i jego zawartość weryfikowana.

IPC

testy semaforów dwie wersje: w pierwszej są tworzone dwa procesy konkurujące ze sobą o jeden semafor; w drugim tworzone są liczne wątki i w tym środowisku sprawdzane jest działanie flagi *SEM_UNDO*,

ipc_stress sprawdza działanie semaforów, segmentów pamięci dzielonej, sygnałów, kolejek komunikatów i łączy nienazwanych. Aby sprawdzić poprawność przesyłanych komunikatów tworzone są sumy kontrolne przesyłane wraz z wiadomością pomiędzy procesami, przez co możliwa jest weryfikacja poprawności przesyłanych danych.

MM - *libm*

libm sprawdza *API* do obsługi pamięci dzielonej w Linux'ie. Tworzy dwa procesy próbujące uzyskać dostęp do danego segmentu pamięci. W czasie testu sprawdzane są między innymi funkcje alokowania pamięci, obsługi blokad na danym segmencie, poprawnej obsługi współbieżności, pisania do pamięci itd.

MM - *mem*01

*mem*01 alokacja pamięci; sprawdza m.in. czy przy nieudanej alokacji pamięci zwracany jest wskaźnik na *NULL*; zapisuje całą wolną pamięć wirtualną, żeby wymusić zapis na dysku i sprawdzić jego poprawność.

MM - *mem*02

*mem*02 sprawdza poprawność działania funkcji *calloc*, *malloc*, *free*, *realloc*, *valloc*; sprawdza obsługę alokacji dużej ilości pamięci i zerowanie jej w wypadku *calloc*'a; pisze do zaalokowanej pamięci i sprawdza poprawność. W przypadku *valloc*'a sprawdza, czy pamięć jest wyrównywana do granicy strony i czy nie jest rzucany *SIGSEGV*.

MM - *page*

page sprawdza poprawność stronicowania pamięci. Tworzy wiele procesów, które proszą o zaalakowanie miejsca pod dużą tablicę. Następnie do tablic wpisywane są ustalone wartości i po zakończeniu pisania przez wszystkie procesy rozpoczyna się sprawdzanie poprawności zapisanych danych.

Scheduler

pthreads tworzy drzewa i listy wątków, uruchamia na nich obliczenia; następnie zbiera i weryfikuje wyniki,

proces_stress analogicznie ale dla procesów,

time – schedule testuje, ile trwają context-switche w przypadku mocno obciążonego schedulera.

Syscalls

Testy wywołań systemowych - ok. 250 zestawów testów *syscall*'i, nie wszystkie jeszcze są testowane.

Commands

Testuje podstawowe polecenia wbudowane. Są tu testy funkcji takich jak *ln*, *cp*, *mv*, *mkdir*, *mail*, *tar*.

Linux Standard Base

Projekt koordynowany przez Linux Foundation. Ma na celu stworzenie spójnego standardu, który pozwoliłby wyeliminować problemy z przenoszeniem aplikacji pomiędzy poszczególnymi dystrybucjami Linux'a. 1 listopada 2005 specyfikacja LSB 2.0.1 stała się standardem ISO 23360. Obecna wersja to 3.0.2. Standard jest z grubsza rozszerzeniem standardu POSIX i Single UNIX Specification. Jest to zbiór reguł na poziomie ABI (czyli działania binariów), a nie API. Certyfikaty zgodności z LSB jak dotąd uzyskały m.in. SUSE, Red Hat, Ubuntu (tylko 6.06), Mandriva.

Zalety LSB

- oszczędność pracy - piszemy tylko jedną wersję, która po skompilowaniu daje się przenosić pomiędzy wszystkimi dystrybucjami spełniającymi standard,
- to prowadziłoby do tego, że może wreszcie pisanie aplikacji komercyjnych (gry!) pod Linux'a stałoby się opłacalne,
- a to na pewno przysporzyłoby mu wielu nowych użytkowników...

Wady LSB

- jak na razie niewielu producentów oprogramowania jest zainteresowanych zadaniem sobie trudu dostosowania aplikacji do standardu LSB ...
- ... bo standard często jest przestarzały ...
- ... i zawiera błędy, ...
- ... a do tego nie testuje aplikacji wystarczająco dokładnie,
- co więcej istnieją liczne konflikty interesów (np. LSB zawiera klauzule mówiącą, że w systemie musi być obsługiwany przynajmniej ograniczony standard RPM, co rodzi protesty użytkowników debianopodobnych dystrybucji).

Jak uzyskać licencję?

Linux Foundation wydało aplikację nazwaną Linux Software Checker, która pozwala sprawdzić zgodność aplikacji ze standardami LSB. Uzyskanie certyfikatu dla aplikacji jest obecnie bezpłatne; płatne jest jedynie certyfikowanie dystrybucji.

Mierniki pokrycia

Mierniki pokrycia, to miary w jakim stopniu testy jakie zostały wykonane pokrywają całość danego programu. Zazwyczaj miara taka jest wyrażana w procentach. Podstawowy sposób wyliczenia takiej miary jest następujący:

- program jest uruchamiany w środowisku debugującym na przygotowanych testach,
- dane wygenerowane przez debugger są przetwarzane do postaci opisującej statystyki wykonywanych instrukcji/funkcji/...,
- narzędzie testujące, w oparciu o kod źródłowy programu generuje raport skuteczności testów i podaje w jakiej mierze testy te pokrywały testowany program.

Pokrycie funkcji

Pokrycie funkcji określa jaki procent funkcji programu został wywołany (przynajmniej raz). Nie ma znaczenia jakie były argumenty poszczególnych funkcji. Miara ta ma pewne znaczenie w przypadku języków programowania o dynamicznym typowaniu w trakcie wykonania. Jeśli typowanie jest statyczne, miara ta ma znacznie mniejsze znaczenie.

Pokrycie linii kodu

Pokrycie linii kodu określa jaki procent linii kodu danego programu został wykonany. Duży współczynnik tego pokrycia świadczy bardzo dobrze o danych testach. Nie musi to jednak oznaczać, że większość wyrażeń w programie została obliczona (np. w związku z leniwą ewaluacją warunków logicznych).

```
int war = 1;
int n = war ? 7 : pos (-1);
/* 100% pokrycia */
```

[a] Gdzie pos(n) to funkcja która zawiesza się dla n<0, a w przeciwnym przypadku zwraca n.

Pokrycie instrukcji

Pokrycie instrukcji określa jaki procent instrukcji programu został wykonany. Jest to warunek silniejszy od warunku pokrycia linii kodu (jedna linia kodu może zawierać kilka instrukcji, z których nie wszystkie muszą być wykonane).

Pokrycie rozgałęzień

Pokrycie rozgałęzień określa jaki procent wszystkich rozgałęzień programu został zrealizowany. Rozgałęzienie to:

- wybór `then` lub `else` w wyrażeniu `if then else`,
- wybór powrotu do pętli, lub wyjścia z niej.

```
int admin = 0;
int n = 7;
while (n >= 0 || admin)
  pos (n- -);
/* 100% pokrycia */
```

Pokrycie warunków logicznych

Pokrycie warunków logicznych określa jaki procent wszystkich wartości Boole'owskich w programie został wyliczony do obu możliwych wartości. Jest to warunek silniejszy od pokrycia rozgałęzień (wartość logiczna całego wyrażenia może być zdeterminowana przez jakiś jego składnik).

```
int n = 7;
int k = 1;
while (n >= 0)
  if (n >= k)
    pos (k*(n-k));
  else
    pos (k*(k-n));
/* 100% pokrycia */
```

Pokrycie ścieżek

Pokrycie ścieżek określa jaki procent wszystkich możliwych ścieżek programu zostało wykonane. Jest to warunek silniejszy od pokrycia rozgałęzień. Miara ta dobrze oddaje w jakim stopniu wszystkie przebiegi programu zostały zrealizowane. Problemem jest, że w przypadku pętli, liczba możliwych ścieżek jest nieskończona. Dlatego stosuje się techniki heurystyczne (np. każdą pętlę należy wykonać $0, 1, 2$ razy).

Pokrycie danych wejściowych

Pokrycie danych wejściowych określa jaki procent wszystkich możliwych danych wejściowych programu został wprowadzony. Jest to najogólniejsza metoda testowania i jeśli jakiś program zostałby tą metodą zweryfikowany w 100%, to (z dokładnością do problemów z niedeterminizmem) można założyć, że program taki jest poprawny. Niestety w przypadku większości programów zbiór danych wejściowych jest gigantyczny, lub nawet nieskończony. Jednak niektóre programy są tak testowane, przykładem mogą być mikrokontrolery prostych urządzeń, lub procesory jako takie.

Problemy z testami

Aby którąkolwiek z powyższych miar pokrycia kodu zastosować, wpierw należy utworzyć zbiór testów dla danego programu, oraz dobrze zdefiniować warunki, kiedy program dany test przeszedł. Zadania te są trudne, a czasami wręcz niemożliwe. Często, konkretnych przebiegów programu, lub konkretnych zestawów warunków logicznych nie daje się uzyskać, niezależnie od danych wejściowych programu. Po wtóre, gdy działanie programu jest probabilistyczne, lub związane ze zmieniającym się otoczeniem, trudno jest zdefiniować co to znaczy, że program działa poprawnie.

Program GCOV

Program GCOV jest standardowym narzędziem GNU. Współpracuje on z kompilatorem GCC, generując statystyki wykonywania danego programu. W zależności od podanych opcji, generowane raporty przedstawiają czas wykonania poszczególnych kawałków kodu, krotność ich wykonania, lub procentowy udział w całości kodu. Raportowanie może dotyczyć pojedynczych linii kodu, rozgałęzień i funkcji programu. Konkretne opcje i możliwości programu można poznać na stronie manuala z nim związanej [6].

Program LCOV

Program LCOV to graficzny font-end dla narzędzia GCOV. Dane generowane przez GCOV są prezentowane na wykresach, oraz w odniesieniu do kodu źródłowego.

O testach wydajnościowych

Testy wydajnościowe można podzielić na następujące kategorie:

benchmark dowolny test wydajnościowy (np. sprzętu, oprogramowania, bazy danych),

low-level benchmarks testy dla hardware'u, np. średni czas dostępu do pamięci,

high-level benchmarks testy dla sterowników, systemów operacyjnych itp.

Po co przeprowadzać testy wydajnościowe

Zwykle po to, żeby zmierzyć wydajność czegoś. Problem polega na tym, że mierząc wydajność jakiejś części systemu chcemy, żeby wynik był możliwie niezależny od innych części, oraz zmierzona wartość oznaczała raczej maksymalne możliwości, a nie średnie, które można uzyskać przy codziennym użytkowaniu systemu. Dlatego benchmarki zwykle nie odzwierciedlają prawdziwego działania systemu, a otrzymane wyniki są dalekie od tego co zobaczymy w rzeczywistości. Dobrym powodem, żeby jednak używać testów wydajnościowych jest to, że właściwie nie ma innego wyjścia. Można co prawda zmierzyć po prostu szybkość działania jakichś aplikacji, jednak taki wynik nie będzie zbyt przydatny ani nam, ani nikomu innemu).

Zalety testów wydajnościowych

Oto co jeszcze możemy zyskać przeprowadzając testy wydajnościowe:

- zbierając statystyczne wyniki testów, możemy zobaczyć jak zmienia się wydajność po wprowadzeniu jakiejś zmiany,
- wykonując testy na różnych maszynach, wersjach jądra itp. możemy porównać ze sobą te maszyny, wersje jądra itp..
- jeśli test mierzy różne własności systemu, to możemy sprawdzić która z nich ma najsłabsze wyniki, a zatem którą warto zoptymalizować.

Przed wykonaniem testu 1

Pierwsze co należy zrobić to zdecydować co dokładnie chce się testować - przeprowadzając test wydajnościowy systemu zwykle chcemy dowiedzieć się jak działa jedna konkretna jego część w różnych warunkach. Oczywiście można testować wiele części systemu podczas jednego testu, warto jednak żeby wyniki były oddzielne, a końcowy wynik był np. średnią z częściowych (w takim przypadku należy jednak pamiętać, że średnia może służyć do porównywania, ale nie niesie treści).

Przed wykonaniem testu 2

Do wykonania testu używa się standardowych narzędzi: stabilnej wersji systemu, standardowego gcc... Jeśli używamy gotowego zestawu testowego, dobrze jeśli pochodzi z zaufanego źródła. Dodatkowo warto tych standardowych narzędzi używać w standardowy sposób - np. ostrożnie używać optymalizacji podczas kompilowania. Jeśli natomiast sami piszemy benchmark - oczywiście powinien być odporny na niestandardowe sposoby użycia.

Program testujący – kod źródłowy

Kod źródłowy programu testującego mnożenie:

```
int main() {
  long int i, j;
  int k = 45;
  int l = 43;
  for (i = 0; i < 1000000; i++) {
    j = k * l;
  }
  return 0;
}
```

Program testujący – analiza

Pisząc podany wyżej program, można by oczekiwać, że używając polecenia *time* do tego programu zmierzymy jak szybko potrafi mnożyć nasz komputer. Niestety - nawet najprostsza optymalizacja prowadzi nas do kodu, który pętli nie wykona ani razu.

Skompilowany program testujący

Kompilacja poleceniem $gcc - S - O1 - masm = intel$:

```
main:
  lea  ecx, [esp+4]
  and  esp, -16
  push DWORD PTR [ecx-4]
  push ebp
  mov  ebp, esp
  push ecx
  mov  eax, 0
  pop  ecx
  pop  ebp
  lea  esp, [ecx-4]
  ret
```

Wnioski 1

Jak widać nie ma tu żadnego skoku, a nasz milion operacji mnożenia został zastąpiony 10 prostymi linijkami kodu do wykonania sekwencyjnego, w których na dodatek nie ma ani jednego mnożenia.
Istnieje też wiele innych możliwości optymalizacji. Ich listę można obejrzeć wpisując $gcc - -help = optimizers$.

Wnioski 2

W przypadku każdego testu wydajności warto dopisać kilka linijek, które sprawiają, że kod wygląda na użyteczny, np. na końcu zwrócić lub wypisać wynik. W powyższym programie to jednak nie wystarczy, ponieważ kompilator zauważy, że w pętli wykonywane jest cały czas to samo (ang. loop invariant computations).
Na szczęście przy wykonywaniu testów jądra nie jest łatwo popełnić taki błąd, ponieważ kompilator nie optymalizuje wywołań systemowych. Ale uwaga! To się naprawdę zdarza...

Przykład 1

Cytat z angielskiej wikipedii, artykuł "Benchmark (computing)":
Computer manufacturers are known to configure their systems to give unrealistically high performance on benchmark tests that are not replicated in real usage. For instance, during the 1980s some compilers could detect a specific mathematical operation used in a well-known floating-point benchmark and replace the operation with a faster mathematically-equivalent operation.

Przykład 2

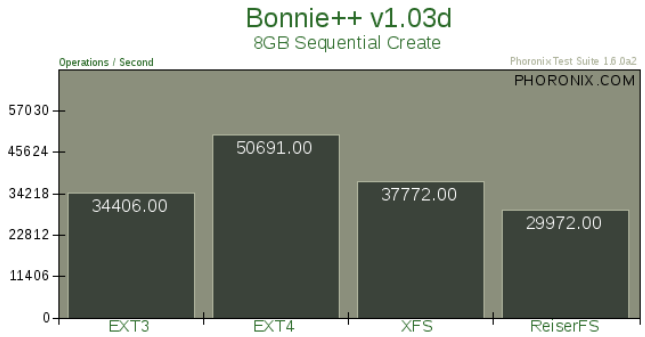
Cytat z angielskiej Wikipedii, artykuł "Dhrystone":
Using Dhrystone as a benchmark has many pitfalls: it features unusual code that is not usually representative of real-life programs. It is also susceptible to compiler optimizations. For example, it does a lot of string copying in an attempt to measure string copying performance. However, the strings in Dhrystone are of known constant length and their starts are aligned on natural boundaries, two characteristics usually absent from real programs. Therefore an optimizer can replace a string copy with a sequence of word moves without any loops, which will be much faster. This optimization therefore overstates system performance, sometimes by more than 30%.

Wykonanie testu

Testy przeprowadza się wielokrotnie - zwykle nie interesują nas bezwzględne wartości zwrócone przez test, ale ich porównywanie. Porównywać można tylko wyniki testów przeprowadzonych w podobnych warunkach - np. po zmianie wartości jednej zmiennej, ten sam test na różnych maszynach, ten sam test na jednej maszynie dla różnych wersji systemu itp.
Poniżej prezentowane są przykłady bazujące na artykułach ze strony [4].

Bonnie

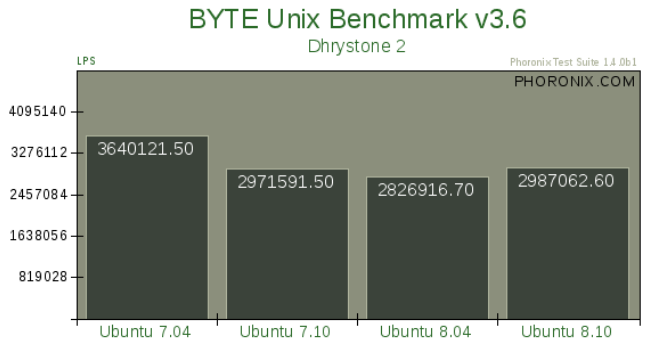
Porównanie różnych systemów plików testem Bonnie, który mierzy czas wykonywania operacji na dysku (patrz [5]).



Bezwzględna ilość operacji, jaką uzyskano dla każdego systemu plików ma znaczenie poboczne - zapewne nie uzyskamy takiego wyniku podczas normalnej pracy z systemem. Istotną informacją jest natomiast, że system EXT4 uzyskał najlepszy wynik.

BYTE Unix Benchmark

Porównanie różnych wersji Ubuntu przy pomocy testu BYTE Unix Benchmark, który mierzy szybkość wykonywania obliczeń, np. FFT, mnożenie macierzy (patrz [1]).



Nawet wiedząc co mierzy ten test ciężko stwierdzić co oznacza otrzymany wynik (w tym przypadku loops per second, ale to nadal nie daje informacji o tym, co faktycznie było mierzone, ponieważ nie znamy treści pętli)...

Czy wyniki testu są wiarygodne?

Niestety często nie są - jak już było napisane łatwo jest popełnić błąd, na etapie tworzenia, kompilacji, wykonania testu, a nawet w momencie interpretacji wyników. W dodatku nie można stwierdzić poprawności lub niepoprawności wyniku porównując go z rzeczywistym działaniem testowanej części. Co gorsze, osoby przeprowadzające testy to nie tylko miłośnicy jądra, pragnący wiarygodnych wyników i polepszenia działania czegoś dzięki tym wynikom. Są to także sprzedawcy, którym zależy na pokazaniu swojego produktu w jak najlepszym świetle. Można nawet spotkać określenie "benchmarketing", oznaczające publikowanie nierzeczywistych wyników w celach marketingowych.

Reguły analizy testów wydajnościowych

Żeby ocenić wiarygodność wyniku testu przeprowadzonego przez inną osobę, należy

- zastanowić się, czy osoba/firma, która go opublikowała, miała interes w publikowaniu zawyżonego/zaniżonego wyniku,
- pamiętać o tym, że dla prawie każdego produktu można znaleźć/stworzyć test, w którym ten produkt będzie miał wyjątkowo dobre wyniki,
- sprawdzić czy przeprowadzony test jest dostępny w sieci.

Ocena testu wykonanego samodzielnie

W przypadku *własnoręcznego* przeprowadzania testu także istnieją pewne wskazówki oraz oznaki, mogące świadczyć o tym, że wynik jest niepoprawny:

- jeśli wyniki kilku testów są rozbieżne, to powinno mieć to jakieś uzasadnienie. Jeśli nie ma - prawdopodobnie wynik jest błędny,
- należy wiedzieć (przynajmniej mniej więcej) co robi dany test,
- nie należy porównywać ze sobą wyników różnych testów, ani liczyć średniej z takich wyników. Bynajmniej nie będzie ona oznaczać średniej wydajności testowanej części systemu w normalnych warunkach...

Ale...

...jeśli jednak uważasz, że wyniki testów są wiarygodne i mogą się komuś przydać, to możesz je opublikować w sieci. Należy jednak podać w jakich dokładnie warunkach był przeprowadzony test.

Przykłady testów wydajnościowych

Istnieje wiele sposobów testowania jądra. Można to robić "domowymi metodami". Najprostszą jest użycie polecenia *time* do zmierzenia np. czasu działania aplikacji lub czasu kompilacji jądra. W ten sposób uzyskujemy informacje o rzeczywistym działaniu. Warto jednak skorzystać z automatycznych testów, ponieważ mierzą dokładniej wydajność naszego systemu.

Bootchart

Bootchart - skrypt shella, który działa w tle podczas uruchamiania systemu i zbiera informacje o wykorzystaniu zasobów; następnie przedstawia wyniki w postaci graficznej. Celem przeprowadzania takiego testu jest pokazanie, jak bardzo sytuacja różni się od idealnej, gdzie wykorzystane jest 100% zasobów, oraz dla których zasobów istnieją najlepsze możliwości optymalizacji. Więcej informacji oraz ładne obrazki na stronie [2].

Phoronix Test Suite

Phoronix Test Suite - duży zestaw benchmarków, np. dla operacji dyskowych, wykonywania obliczeń matematycznych, kompilacji jądra oraz innych aplikacji itd. itd. itd... Istnieje również strona, na której użytkownicy tego zestawu testów mogą publikować swoje wyniki. Więcej informacji na stronie [3].

UnixBench

UnixBench - wersja 4.01 ukazała się w roku 1997 i od tego czasu niewiele się w niej zmieniło. Nadal można nim jednak przetestować jądro systemu, np. wykonywanie skryptów shella (pojedynczo i współbieżnie), wywołania systemowe, użycie pipe'ów. Ten test, jak wiele innych, mierzy ile obrotów można wykonać w ciągu sekundy (lps), dla pętli zawierającej np. zestaw operacji arytmetycznych, wywołań systemowych... Więcej informacji na stronie [4].

Testowanie w praktyce

Prosty test: skompilować, uruchomić i spróbować wystartować cały system z nowym jądrem. Gdy to się uda staramy się przez chwile na nim popracować. W praktyce 90% błędów jest wykrywanych przy budowaniu na wielu architekturach, statycznym weryfikowaniu i uruchamianiu jądra.
W kodzie znajduje się wiele asercji, które sprawdzają poprawność podczas działania.
Przed testowaniem trzeba sprawdzić stabilność samego sprzętu - czy sprawny ram, dysk, procesor?

Autotest

Autotest - projekt open-source (Licencja GPL) używany i rozwijany przez wiele organizacji (m.in. Google, IBM).
Umożliwia kompilacje jądra i automatyczne testowanie na wielu maszynach i różnych konfiguracjach jądra.
Sam nie zawiera testów, ale można bez problemu uruchamiać wiele zewnętrznych testów.

Autotest - cele

Deweloperzy nie mają dostępu do dużej ilości różnych maszyn.
Gdy środowisko testów jest wydzielone na osobne maszyny łatwiej utrzymać niezmienność środowiska.
Możliwość testowania na zaawansowanych komputerach (np. IBM udostępnił 32 rdzeniową maszynę opartą o procesory Xeon 2GHz).
Współpraca z gcov/lcov - analiza pokrycia kodu testami.
Umożliwia śledzenie zmian wydajności (patrz [1]).

Autotest - problemy

Nie wszystkie testy są open-source, ale wg. założeń wystarczą tylko wyniki. Niestety, osoba pisząca testy nie może przewidzieć wszystkich błędów, dlatego pokazywanie samych rezultatów może być mylące.
Aktualnie jest uruchamiane pod Ubuntu 6.06.

Zewnętrzne narzędzia do Autotest 1

aiostress uruchamia dużo operacji wejścia/wyjścia na dysku (tworzenie, czytanie i zapisywanie plików),

bonnie testowanie poprawności i wydajności operacji na dużych plikach (np >2GB),

tbench, dbench, smbtorture oryginalnie zestaw do testowania serwerów Samba, bo mało kto może przeprowadzać testy na serwerze, który używa 60-120 klientów,

tbench wywołuje sztuczny ruch TCP podobny do generowanego przez Samba,

Zewnętrzne narzędzia do Autotest 1

dbench wywołuje obciążenie systemu plików podobne do generowanych przez serwer samby, często dbench jest używany do porównywania systemów plików,

smbtorture obciążenie prawdziwego serwera przez prawdziwy klient (tylko operacje sztuczne),

fsx testowanie poprawności systemów plików,

reaim odświeżona wersja testów AIM - testy wydajności dla systemu z wieloma użytkownikami/procesami,

kernbench testowanie kompilacji jadra bez użycia dysku (>2GB RAM'u, ilość procesów kompilacji = 4*ilość rdzeni), możliwość testowania schedulera, zarządzania pamięcią.

Inne narzędzia do Autotest

Inne ciekawe testy zewnętrzne w Autotest: *cpu_hotplug*, *cyclictest*, *dbt2*, *fio*, *fs_mark*, *interbench*, *iozone*, *isic*, *libhugetlbfs*, *lmbench*, *ltp*, *netperf2*, *pi_tests*, *pktgen*, *rmaptest*, *scrashme*, *selftest*, *sleeptest*, *sparse*, *stress*, *tiobench*, *unixbench*, *xmtest*.
Profilery: *catprofile*, *lockmeter*, *oprofile*, *readprofile*.

LTP

- Ponad 3000 małych testów,
- testowanie systemu plików, I/O dysku, zarządzania pamięcią, IPC, scheduler'a,
- poprawność *syscall*.

Pokrycie kodu jądra przez LTP

2.6.24 kernel code coverage using March 2008 LTP

Directory	Coverage %	Coverage lines
`fs`	52.9%	10778/20367 lines
`include/asm`	50.9%	613/1204 lines
`include/linux`	60.0%	2283/3812 lines
`include/net`	57.6%	1015/1762 lines
`ipc`	56.4%	1539/2729 lines
`kernel`	39.1%	10097/25837 lines
`lib`	43.2%	2159/4992 lines
`mm`	52.7%	7066/13396 lines
`net`	65.7%	633/964 lines
`security`	51.9%	666/1283 lines

KLive, gcov, GIT

KLive jest narzędziem, dzięki któremu deweloperzy Linuksa mogą się dowiedzieć jak długo dane drzewo było testowane.
Deweloperzy nie wiedzą ile osób i jak długo testowało konkretne wydanie. Czasami zdarza się sytuacja, że wszystko działa jak należy i nikt nie zgłasza żadnych błędów - opiekun drzewa nie ma żadnej pewności, czy ktokolwiek testował dane wydanie.

gcov/lcov używane do analizowania pokrycia kodu testami.

GIT posiada narzędzia do binarnego wyszukiwania wersji w której pojawił się błąd.

Własny test

Literatura 1

- Wykres zmian wydajności różnych wersji jądra
  http://test.kernel.org/ols2006
- Strona projektu LTP
  http://ltp.sourceforge.net/
- http://www.linuxbase.org/test/
- http://www.phoronix.com/scan.php? page=phoronix_articles
- http://www.phoronix.com/scan.php?
  page=article&item=ext4_benchmarks&num=1

Literatura 2

- http://www.phoronix.com/scan.php?page=article&item=ubuntu_bench_2008&num=1
- http://www.bootchart.org/
- http://phoronix-test-suite.com/
- http://ftp.tux.org/pub/benchmarks/System/unixbench/
- http://tldp.org/HOWTO/Benchmarking-HOWTO.html
- Strony manuala dotyczące programu GCOV http://linuxcommand.org/man_pages/gcov1.html