Testowanie

Katarzyna Jachim, Maciej Próchniak

22 grudnia 2004 roku

Spis treści

1 Wprowadzenie

Celem tej prezentacji jest przedstawienie ogólnej metodologii testowania i analizy wyników testów, pokazanie kilku wybranych programów ułatwiających testowanie, a także przedstawienie kilku tematów związanych z testowaniem Linuxa, takich jak benchmarki i testowanie jądra.

2 Metodologia

2.1 Po co testować

Pisząc program chcielibyśmy się w jakiś sposób przekonać, czy jest on poprawny. W przypadku większości programów chcielibyśmy również wiedzieć, jaka jest ich wydajność, czyli jak szybko działają, jakie są ich wymagania sprzętowe itp. Często jedyną możliwością aby się o tym przekonać są testy. Rozważa się co prawda formalne dowodzenie poprawności programów, ale nie jest metoda możliwa do zastosowania w typowych przypadkach, przy pisaniu większych systemów. Podobnie jest z wydajnością: należy sobie oczywiście zdawać sprawę, jaka jest złożoność implementowanego przez nas algorytmu, ale czasami sama analiza złożoności nic nam nie da, bo np. pewne skrajne przypadki nie występują "w przyrodzie" lub, co gorsze, są bardziej prawdopodobne niż te "dobre".

2.2 Rodzaje testów

Testy możemy dzielić ze względu na różne cechy.
Podział ze względu na to, co testujemy:

- poprawnościowe (czy system zachowuje się poprawnie dla różnych danych)
- wydajnościowe (jaki jest czas działania systemu dla różnych danych)
- długotrwałego działania (jak system zachowuje się w przypadku dłuższego działania)
- obciążeniowe i przeciążeniowe (dla jakich obciążeń system działa poprawnie, jak zachowuje się dla większych)

Podział ze względu na naszą wiedzę o systemie:

- biała skrzynka - znamy kod, najczęściej w ten sposób testujemy przypadki szczególne, dla których widzać, że coś może nie działać, nie testujemy pełnej funkcjonalności, ale tylko poszczególne fragmenty
- czarna skrzynka - nie znamy kodu, najczęściej w ten sposób testujemy ogólną funkcjonalność, przy testowaniu całego systemu tylko w ten sposób stosunkowo łatwo pominąć jakis przypadek szczególny

2.2.1 Ogólny schemat testowania

1. Testy statyczne - wykonywane jeszcze przed uruchomieniem systemu, są to testy poprawności kodu

    - syntaktycznej - te najczęściej wykonuje za nas kompilator
    - semantycznej - czasami też częściowe zostaną zauważone przez lepszy kompilator (np. odwołanie do zainicjalizowanego wskaźnika), ale niektóre musimy wykonać sami (choćby poprawność nawiasowania)

2. Testy dynamiczne - wykonywane po uruchomieniu programu

    - biała skrzynka
    - czarna skrzynka - i tu mamy dwa rodzaje: testy funkcjonalności, kiedy sprawdzamy, czy w systemie da się wykonać wszystkie zaplanowane działania oraz testy losowe, kiedy sprawdzamy jak system będzie się zachowywał w sytuacji losowej, np. po wciśnięciu losowej kombinacji klawiszy, czy kliknięciu w jakieś miejsce na ekranie

Zazwyczaj testy wykonujemy właśnie w takiej kolejności jak podana powyżej: najpierw testy statyczne, następnie testy typu biała skrzynka i testy funkcjonalności, a na końcu testy losowego zachowania.

2.3 Scenariusze testowe

W przypadku dużego systemu najprawdopodobniej będzie przeprowadzana duża ilość testów typu czarna skrzynka. Takie testy też trzeba przygotować. Polega ono na napisaniu scenariusza testowego, czyli szczegółowej "instrukcji obsługi" dla osoby testującej. W scenariuszu testowym opisujemy kolejne czynności, które należy zrobić, oraz oczekiwane wyniki po ich wykonaniu. Zadaniem osoby testującej jest wykonanie kolejnych kroków opisanych w scenariuszu i zaznaczenie wyników.

3 Narzędzia wspomagające automatyzację testowania

Automatyzacją nazywamy zastępowanie testów manualnych — tzn. wykonywanych przez testerów — przez różnego rodzaju skrypty testowe. Zauważmy, że jeśli chodzi o systemy operacyjne to większość testów jest z konieczności automatyczne — jądro nie wchodzi zwykle w bezpośrednie interakcje z użyrtkownikiem. Automatyzacja ma natomiast duże znaczenie przy testowaniu różnego rodzaju aplikacji biurowych.

Głównymi celami automatyzacji testów są:

- Możliwość wielokrotnego powtarzania testów bez udziału człowieka — przy testowaniu manualnym często jest niemożliwe powtórzenie takich samych sekwencji działań.

- Automatyczna analiza testów — programy testujące często dostarczają po zakończeniu testów szczegółowych raportów o ich przebiegu.
- Możliwość przetestowania wszytkich elementów systemu — przy testowaniu manualnym często nie jest możliwe wykonanie wszytkich scenariuszy testowych — dotyczy to zwłaszcza testów obciążeniowych.
- Efektywność testowania — skrypty testujące wymagają mniejszej ilości czasu i zasobów do przeprowadzenia testów niż testerzy.

Można wyróżnić dwie podstawowe techniki automatyzacji testów:

- Ręcznie pisane skrypty testowe
- Robot User - użycie techniki capture/playback

Pierwsza z tych technik nie wymaga chyba zbyt szczegółowych wyjaśnień. Po zaprojektowaniu testów implementator testów pisze w odpowiednim języku programowania skrypty, które będą symulowały działanie testowanego progamu.

Technika capture/playback polega na rejestrowaniu przy użyciu specjalnych programów poszczególnych kroków użytkownika w czasie interakcji z programem, takich jak kliknięcia myszą lub wciskane klawisze, aby na ich podstawie automatycznie wygenerować skrypty testujące daną funkcjonalność programu. Przykładami progamów realizujących tą technikę są:

- Rational Robot
- Panorama/Playback

Rational Robot jest częścią pakietu Rational i będzie omówiony poniżej.

Inny podział automatycznych testów plega na rozróżnieniu kiedy skrypt testujący korzysta z istniejącego już interfejsu użytkownika, kiedy zaś korzysta z specjalnego interfejsu aplikacji (API).

Każdy z podanych spsobów ma swoje wady i zalety. Głównymi wadami testów opartychna interfejsie użytkownika są bardzo wysokie koszty ich utrzymania oraz problemy przy dokonywaniu zmian. Jest to spowodowane tym, że często nawet niewielkie zmiany w interfejsie powodują konieczność dokonania dużych zmian w skryptach. Z drugiej strony skrypty oparte o API wymagają znajomości specjalnyh technik i narzędzi.

Przykładami narzędzi korzystających przy generowaniu skryptów jedynie z interfejsu użytkownika są ww. Rational Robot i Panorama. Przykładem programu korzystającego ze specjalnego interfejsu na potrzeby testów jest JUnit.

Jednym z głównych celów automatyzacji testów jest zmniejszenie kosztów. Niewłaściwe zaprojektowane testy mogą jednak spowodować, że ich przeprowadzenie i utrzymanie będzie niemal tak pracochłonne jak samo stworzenie oprogramowania. Dlatego też należy unikać kilku podstawowych błędów, które powodują znaczący spadek efektywności testów, a nawet ich nieskuteczność.

- Kodowanie danych testowych w programie. Jest to jeden z podstawowych błędów, który powoduje, że skrypty testowe stają się nieczytelne, zaś ich zmiana powoduje konieczność zmiany kodu samych skryptów.

- Brak jasno określonych celów testów. Aby po przeprowadzeniu testów prawidłowo zinterretować ich wyniki musimy określić jakiego rodzaju wyniki chcemy otrzymać.

- Skomplikowanie poszczególnych testów, w szczególności duża ilość instrukcji warunkowych i rozgałęzień. Powoduje to problemy przy określaniu przyczyn niepowodzenia testów.

- Skrypty wrażliwe na zmiany w programie. Dotyczy to zwłaszcza skryptów działających w oparciu o interfejs użytkownika, który może często ulegać zmianom.

3.1 Rational — zaawansowane narzędzia wspomagające testowanie

Rational to ogromny system wspomagający tworzenie oprogramowania. W jego skład wchodzą również programy wspomagające testowanie, które postaramy się pokrótce omówić. Wszystkie te programy są przeznaczone przede wszystkim do testowania programów w Javie, C++ i VB, jednak niektóre części (np. TestManager'a) można wykorzystać również przy korzystaniu z innego języka (jedynym narzędziem, które wymaga podania od nas języka w którym został napisany kod jest TestFactory). Niestety, nie udało się nam sprawdzić wszystkich możliwości udostępnianych przez ten system, co było spowodowane jego wielkością i stopniem skomplikowania. Wszystkie programy związane z testowaniem spod znaku Rationala utrzymują jedną bazę informacji o testach, dzięki czemu np. możemy przy planowaniu testów przy pomocy TestManager'a korzystać z testów wygenerowanych w Robocie czy TestFactory. Podobnie, po utworzeniu jakiegoś nowego testu możemy go dodać do konkretnego planu testów.

3.1.1 Rational Robot

To narzędzie typu capture&playback, umożliwiające szybkie tworzenie nawet skomplikowanych testów. Co można z jego pomocą zrobić:

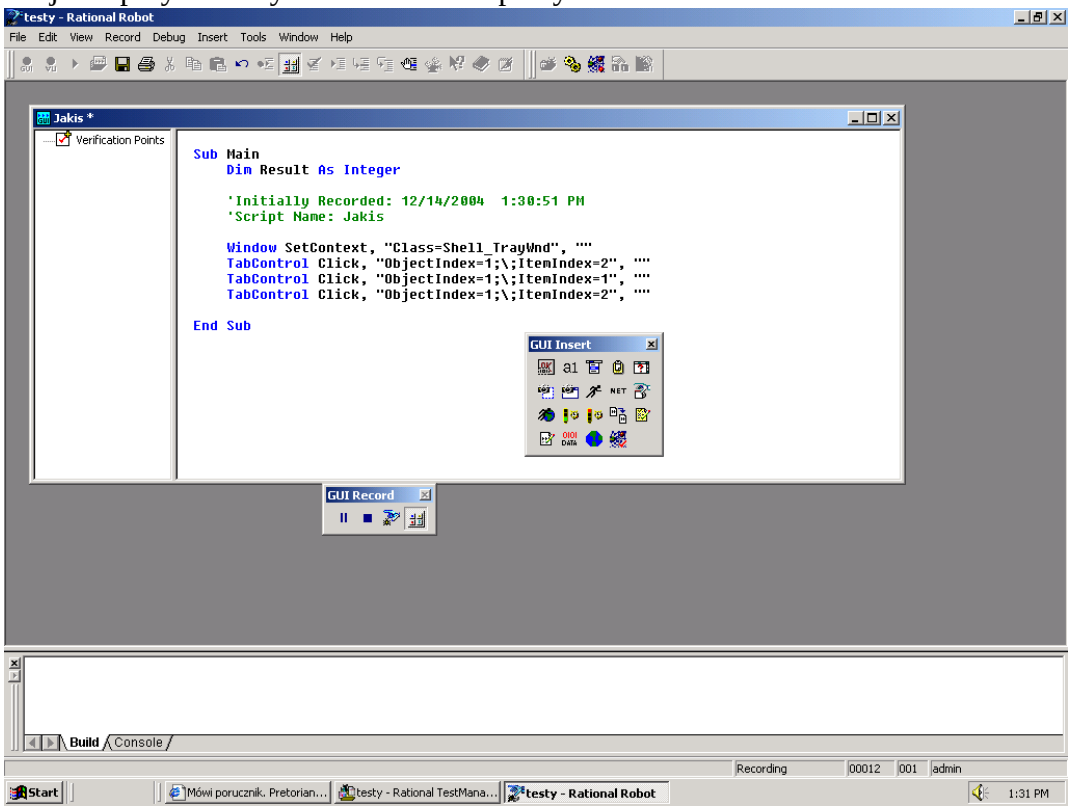
- nagrać testy, a następnie je odtwarzać, sprawdzając stany obiektów w wybranych momentach

- łatwo zmienić kod nagranych testów (wygodne kolorowanie tekstu skryptów)

- testować obiekty nawet jeśli nie są widoczne z poziomu użytkownika

- zachować informacje o przebiegu skryptu; RR współpracuje z innymi narzędziami Rationala, takimi jak Quantify i Purify, które mają na celu pomóc w ocenie poprawności programu

Częścią Robota jest SiteCheck, narzędzie do testowania stron internetowych. Poniżej możemy obejrzeć przykładowy widok w czasie pracy z Rational Robot:



3.1.2 Rational TestManager

Jest to narzędzie do tworzenia i zarządzania testami. Za jego pomocą możemy tworzyć nowe skrypty i scenariusze testowe, dodawać je do już istniejących zbiorów testów, przypisywać je do różnych faz tworzenia systemu (zgodnie z metodologią RUP).

3.1.3 Rational ManualTest

Narzędzie do tworzenia scenariuszy testowych. Pozwala na wygenerowanie ciągu poleceń dla osoby testującej razem z zaznaczeniem, jakiego rodzaju jest to czynność:

- step - pojedyńczy krok, powinien być zazwyczaj opisany jednym zdaniem, jednak jego opis nie może budzić wątpliwości (są to zdania w stylu: "kliknij w przycisk start")

- verification point - pytanie odnośnie bieżącego stanu programu (np. czy system się uruchomił, czy otworzyło się okienko..), odpowiedź na nie powinna być typu tak/nie, aby tester mógł łatwo stwierdzić czy dany test zakończył się sukcesem, czy porażką (sukces = odpowiedź "tak" na wszystkie pytania)

3.1.4 Rational TestFactory

Narzędzie to ma w założeniu zanalizować dostarczony kod i na jego podstawie stworzyć pewne testy. Możemy przy jego pomocy testować programy napisane w Javie, C++ i VB. Możliwości:

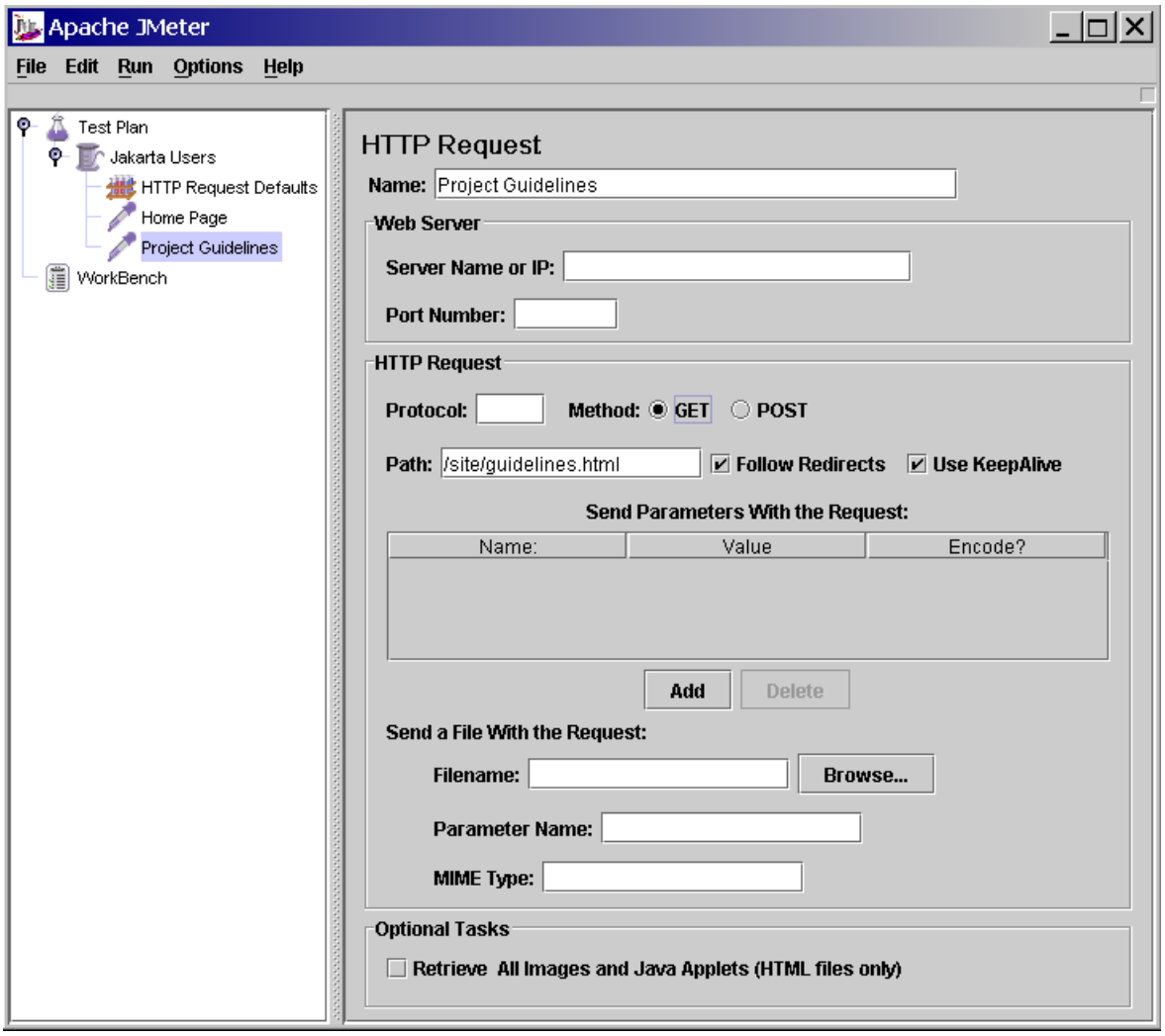
- automatyczne tworzenie i utrzymywanie dokładnej informacji o testowanym systemie
- automatyczne generowanie skryptów na podstawie posiadanych informacji o systemie

3.2 JMeter — narzędzie do testowania usług sieciowych

Jednym z narzędzi pozwalających na automatyzację testów jest JMeter. Zakres jego działania jest niezbyt szeroki, umożliwia on bowiem głownie testowanie wydajności usług sieciowych, zostanie omówiony tutaj gdyż jest (w przeciwieństwie np. do Rationala) niezbyt skomplikowany w obsłudze, oraz łatwy do ściągnięcia i instalacji.

Podstawową funkcją, którą oferują zestawy testów generowane przez JMeter jest generowanie żądań do serwisów sieciowych. JMeter umożliwia jednoczesne generowanie wielu żądań, zarówno stron WWW, jak i usług FTP i zwykłej komunikacji TCP. Zasada działania programu jest dość prosta: najpierw tworzymy plan testów, ustalając kiedy i jakie żądania mają być generowane, następnie uruchamiamy plan testów, i JMeter generuje odpowiednie żądania. Następnie mierzone są czasy odpowiedzi.

Poniżej przedstawiony jest przykładowy widok z działania programu. Po lewej widzimy główną strukturę planu testów, natomiast po prawej — okno w którym można konfigurować poszczególne komponenty planu.



Podstawową jednostką pracy programu jest plan testów, którego strukturę pokrótce omówimy. Plan testów ma strukturę drzewa, składają się nań grupy wątków reprezentowane na diagramie przez szpulki. Reprezentują one poszczególne zadania wykonywane w czasie przeprowadzania testów. Każdej takiej grupie przypisana jest liczba wątków i ile razy mają się one wykonać w czasie jednego testu.

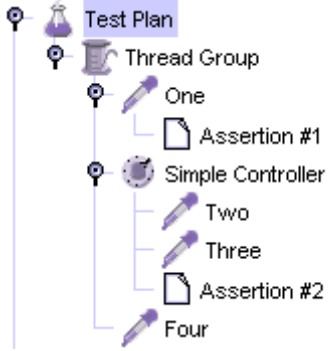
W skład grupy wątków wchodzą takie elementy jak:

- Kontrolery - wątek testowy wykona żądania będące dziećmi kontrolera, jeżeli spełnione są opowiednie warunki ustawione w kontrolerze, np. warunek pętli, warunek logiczny.

- Samplery - podstawowa jednostka działania testów. Każdy sampler odpowiada za generowanie jednego żądania określonego typu, takiego jak: HTTP, FTP, TCP, JDBC i inne.

- Listenery - pozwalają na generowanie tabel i wykresów w których prezentowane są wyniki testów.

Poniżej prezentujemy przykładowe elementy planu, na obrazku widać jak jest realizowana struktura hierarchiczna.



Po utworzeniu i skonfigurowaniu planu testów uruchamiamy go wybierając polecenia Run->Start. W czasie wykonywania testów i po ich zakończeniu tworzone są wykresy i/lub tabelki w których są podane wyniki testów — tj. czasy oczekiwania na odpowiedź a także dane statystyczne takie jak średnia i mediana.

3.2.1 Zdalne testowanie

W wielu sytuacjach jeden komputer nie jest w stanie wygenerować w odpowiednim czasie odpowiedniej ilości żądań do serwera. Wtedy JMeter umożliwia kontrolę wielu instancji swojego

działania na różnych maszynach poprzez jednego kordynatora. Aby to wykorzystać, należy:

- na maszynach-dzieciach uruchomić program jmeter-server
- skonfigurować w koordynatorze plik bin/jmeter.properties — ustawić zmienną **remote_hosts**
- uruchomić JMeter na koordynatorze i zamiast opcji Run->Start wybrać Run->Remote Start

3.2.2 Rozszerzenie działania JMeter

W niektórych przypadkach standardowe elementy planu testów mogą okazać się niewystarczające dla potrzeb użytkownika. JMeter oferuje możliwość dołączania nowych typów obiektów, które mogą być potem wykorzystywane w planie testów na równi ze standardowymi.

Aby dołączyć nowy typ obiektu musimy napisać własną klasę w Javie, implementującą określony interfejs — wszystko jest dokładnie opisane w manualu. Następnie spakowany plik .jar umieszczamy w odpowiednim katalogu i po odpowiedniej konfiguracji programu możemy używać nowego typu obiektu.

4 Testowanie systemów operacyjnych i innych programów w linuxie

Przejdziemy teraz do opisu problemów jakie powstają przy testowaniu systemów operacyjnych i ich części oraz do omówienia niektórych narzędzi pomocnych w testowaniu i analizowaniu programów pisanych w linuxie.

4.1 Debugger

Jednym z podstawowych narzędzi służących do wykrywania błędów w programch pisanych w C jest GDB. Nie będziemy tu omawiać szczegółowo jego działania, min. dlatego że było to już wcześniej wspominane. Przypomnijmy jednak podstawowe komendy jakie można wydawać:

- run - uruchamia program który testujemy
- breakpoint - ustawia punkt kontrolny na jakiejś funkcji lub symbolu
- c - kontynuacja zawiszonego programu
- next - przechodzenie instrukcja po instrukcji, bez zagłębiania w wywołania funkcji
- step - jw. ale zagłębiając się w wywołania.

Nie zawsze jest tak, że możemy zatrzymać debugowany program aby obejrzeć jego działanie, a potem przywrócić jego działanie bez żadnych skutków ubocznych - jeden z przykładów to oczywiście systemy operacyjne. GDB oferuje tu możliwość zdalnego debugowania. Oznacza to, że na jednej maszynie działa program testowany, zaś program gdbserver wysyła odpowiednie informacje o jego działaniu do drugiej maszyny , na której działa normalny gdb. W ten sposób przetwarzanie informacji odbywa się na maszynie testującej.

4.2 KDB — debugowanie jądra

Jednym z problemów przy debugowaniu jądra jest niemożność standardowego użycia debuggera. Wynika to z oczywistego faktu, że wykonanie pewnych fragmentów jądra nie może być wstrzymane przez GDB, aby dokonać odpowiednich operacji. Istnieje kilka rozwiązań tego problemu:

- użycie UML — omawiane w ramach poprzedniej prezentacji
- użycie zdalnego debugowania — patrz poprzedni punkt
- użycie KDB

KDB jest programem, który umożliwia debugowanie jądra. Aby go użyć należy wykonać następujące czynności:

- ściągnięcie i zainstalowanie patcha na jądro — np. z http://oss.sgi.com/projects/kdb
- przy konfiguracji możemy ustalić — za pomocą flagi **CONFIG_KDB_OFF** czy KDB będzie aktywny domyślnie, można to też ustawić przy bootowaniu przez ustawienie opcji **kdb=on**
- gdy KDB jest aktywny jego uruchomienie następuje przy każdym kernel panic.
- możemy także uruchomić KDB ręcznie, przez wciśnięcie klawisza **Pause**

KDB oferuje z grubsza podobne funkcje jak zwykly debugger. Do najważniejszych poleceń należą:

1. Zarządzanie pamięcią: md/mm — wyświetla/ustawia zawartość komórki pamięci pod podanym adresem

   Zarządzanie rejestrami procesora: rd/mm — wyświetla/ustawia zawartość poszczególnych rejestrów

2. Ustawianie punktów kontrolnych: bp/bc — ustawianie i zwalnianie breakpointów na adresie/symbolu, be/bd — aktywacja/deaktywacja breakpointu, bl — lista wszystkich

3. Inne: id — pokazuje zdeasemblowane instrukcje, poczynając od danego adresu, ss — wykonanie jednej instrukcji

4.3 Gcov+lcov — badanie skuteczności testów

Wspomnieliśmy już wcześniej, że jedną z podstawowych cech jakie powinien mieć dobry zestaw testów jest jego kompletność — tzn. powinniśmy przetestować wszystkie wariany działania programu. Jednym ze wskaźników, które pozwalają to ocenić jest fakt, czy w czasie testów były wykonane wszytskie rozgałęzienia i linie kodu w programie.

4.3.1 gcov

Gcov jest prostym programem, załączonym do większości dystrybucji Linuxa, który pozwala na badanie które linie kodu i ile razy były wykonane. Program współpracuje z językami programowania:

- C
- C++
- Modula2

Aby gcov mógł działać musimy wykonać następujące kroki:

- kompilacja programu. Aby móc korzystać z gcov program musi być skompilowany z flagami **-fprofile-arcs** oraz **-ftest-coverage**
- w katalogu, w którym kompilowaliśmy tworzone są pliki z rozszerzeniami .da, .bb i .bbg. Gcov przechowuje w nich informacje o wykonanych liniach.
- Uruchamiamy program dowolną ilość razy.
- Po zakończeniu testów wywołujemy **gcov plik.c**. Utworzony zostaje plik o rozszerzeniu .gcov w którym podane są informacje o statystykach wykonania programu.

Przykładowy wynik dziaąlania gcov na programie hello.c:

```
-:    0:Source:hello.c
-:    0:Object:hello.bb
-:    1:#include <stdio.h>
-:    2:
-:    3:#define text "Hello"
-:    4:
-:    5:int main()
2:    6:{
2:    7:  printf ("%s world\n", text);
2:    8:  return 0;
-:    9:}
```

Poszczególne kolumny wydruku mają następujące znaczenie: ile razy wykonano poszczególne instrukcje, numer kolejnej linii.

4.3.2 Gcov i testowanie jądra

Gcov umożliwia także analizowanie wykonania kodu jądra. Aby wykorzystać tę funkcjonalność należy najpierw zainstalować moduł **gcov-kernel**, dostępny pod adresem: **http://sourceforge.net/projects/lse** i uruchomić go.

4.3.3 Lcov - generowanie wydruków dla gcov'a

Lcov jest prostym rozszerzeniem gcov, w zasadzie jest prostu zbiorem skryptów, generujących wykresy obrazujące wyniki działania gcov w formacie html lub png.

Lcov może być ściągnięty ze strony:

Po zainstalowaniu Lcov może być użyty według następującego schematu:

- Kompilujemy program z odpowiednimi flagami, a następnie uruchamiamy go żądaną ilość razy

- Wykonujemy polecenie:

```
lcov --capture --output-file nazwa.info --test-name " nazwa testu"
```

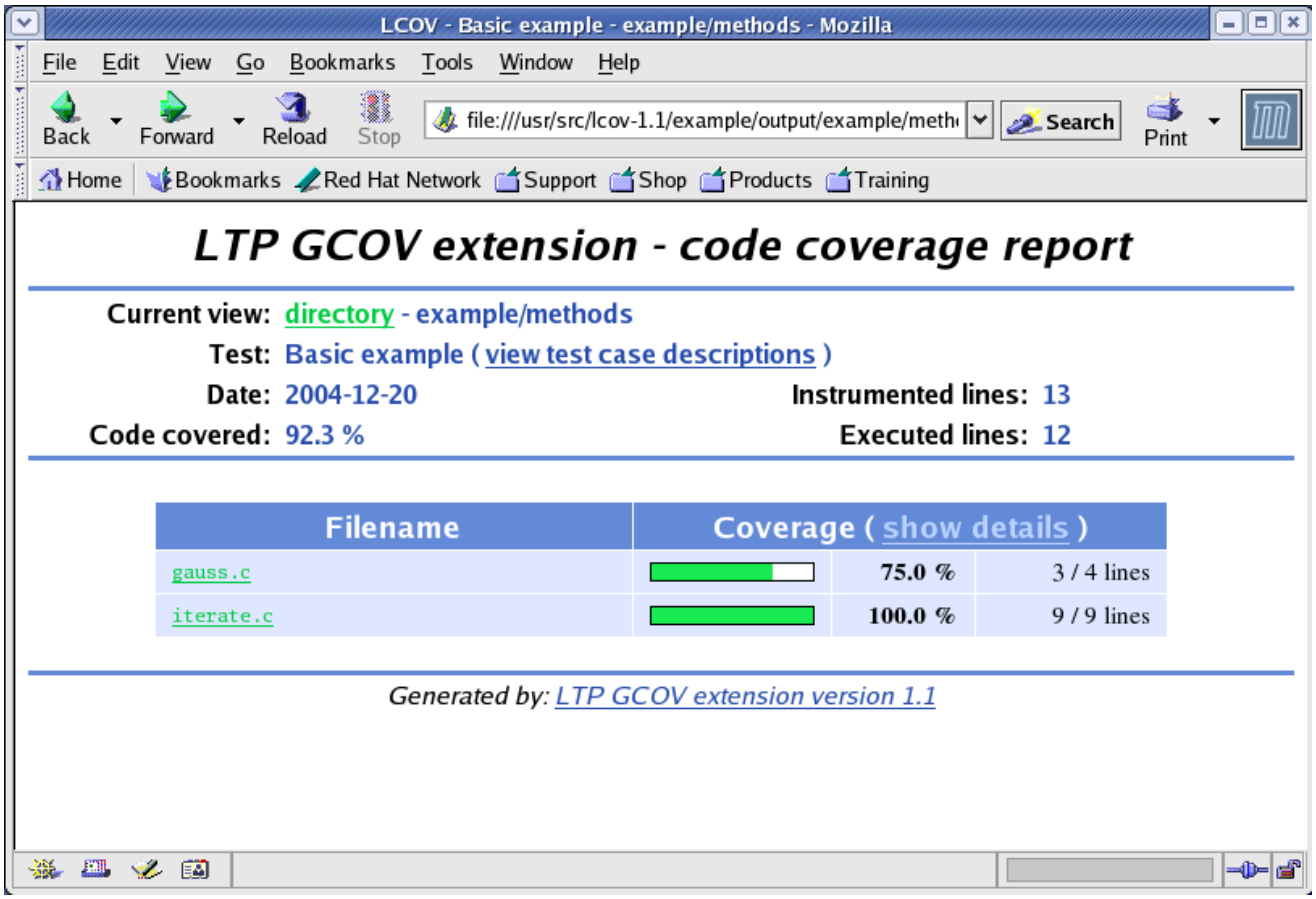
  W jego wyniku otrzymamy plik **nazwa.info**, w którym będą informacje na temat statystyk działania programu.

- Aby wygenerować z niego np. pliki html wywołujemy polecenie:

```
genhtml nazwa.info --title "Przyklad" --show-details
```

  Ostatnia opcja powoduje pokazanie większej liczby informacji.

Poniżej przykładowy wynik działania lcov:



4.4 Top — narzędzie do analizowania zadań systemu

W czasie testów często chcemy wiedzieć jakie zadania wykonuje w danym momencie nasz system. Jednym z narzędzi, pozwalających na śledzenie w czasie rzeczywistym stanu systemu operacyjnego — tzn. jakie procesy aktualnie działają, jakie jest zużycie pamięci jest **top**. Program jest dołączony do większości dystrybucji Linuxa.

Top może dla każdego z działających procesów podać takie informacje jak:

- PID
- PPID — numer rodzica
- RUSER — prawdziwa nazwa właściciela
- UID — numer właściciela
- PR — priorytet
- %CPU — jaki procent czasu procesora zużył proces od ostatniego pomiaru
- %MEM — jaki procent pamięci fizycznej miał proces od ostatniego pomiaru

Oprócz tego wyświetlane są takie informacje jak ilość wykonywanych zadań, jak wiele czasu procesora jest wykorzystywane, ile pamięci fizycznej i swapa jest zajęte.

4.5 Benchmarki

Benchmarki są testami wydajności systemu operacyjnego. Pełnią one kluczową rolę przy projektowaniu poszczególnych elementów systemu.

Benchmarki możemy podzielić na dwie grupy:

- syntetyczne
- aplikacyjne

Benchmarki syntetyczne obejmują jeden, specyficzny fragment funkcjonalności systemu. Przykładem jest tu badanie opóźnienia przy błędzie braku strony. Zaletą takich benchmarków jest konkretność informacji: możemy dokładnie określić co ma wpływ na taki wynik testu. Z drugiej strony nie mamy informacji o wpływie testowanej funkcji na działanie całego systemu — jeżeli badana funkcja jest wywoływana relatywnie rzadko, to jej efektywność będzie miała niewielki wpływ na wydajność całego systemu.

Benchmarki aplikacyjne obejmują sobą działanie całej aplikacji, a nawet kilku. Pozwalają one na przetestowanie działania całego programu — jednak nie pozwalają one na szczegółową analizę progrmu — która jego część generuje największe obciążenia czasowe lub pamięciowe.

4.5.1 Projektowanie i przeprowadzanie benchmarków

Przy wykonywaniu i projektowaniu testów wydajnościowych konieczne jest pamiętanie o kilku regułach, których zachowanie jest niezbędne aby wyniki testu były miarodajne.

Aby zestaw testów był miarodajny muszą być spełnione określone warunki dla czterech faz przygotowania testów:

- wybór testów
- analiza wykorzystania zasobów systemowych
- analiza zakres testów
- wykonanie zestawu testów

Zestaw testów powinien sprawdzać efektywność działania takich zasobów systemowych jak:

- CPU
- pamięć
- I/O
- Sieć

Poprawnie skonstruowany zestaw testów powinien obejmować możliwie dokładnie fragment kodu, którego dotyczy. Przy sprawdzaniu tego pomocne są narzędzia takie jak gcov, omówione w poprzednim punkcie.

Końcową fazą przygotowania planu testów jest uruchomienie go na jądrze, o którym wiadomo że jest stabilne. W ten sposób możemy uzyskać informację o tym jakie wyniki daje nasz zestaw na standardowej konfiguracji i następnie porównać wyniki.

Przy projektowaniu benchmarków warto pamiętać też o nastęujących zasadach:

- Określenie celu testów — co chcemy tak naprawdę zmierzyć
- Użycie standardowych narzędzi — stabilnej wersji jądra, standardowego kompilatora itd.
- Załączanie kompletnego opisu użytej kofiguracji.

Przy analizie wszelkich testów wydajnościowych należy być szczególnie ostrożnym, z uwagi na to że próbujemy zwykle na podstawie danych pochodzących z wyników testów wnioskować o ogólnych regułach zachowania systemu. Poniżej przedstawiamy niektóre typowe błędy, jakie można popełnić:

- Nie uwzględnienie istotnych czynników (np. nie tylko rodzaju danych, ale i ich rozmiaru)
- Założenie, że zależności wyników testu od rozmiaru danych są np. liniowe lub kwadratowe, a następnie aproksymowanie działania systemu dla innych danych przez odpowiednie funkcje. W rzeczywistości w pewnych systuacjach (takich jak wejście/wyjście) narzut czasowy rośnie szybciej.
- Aproksymacja na podstawie zbyt małej próbki. Nie możemy wnioskować o zachowaniu systemu na podstawie jedynie fragmentarycznych danych, ponieważ wiele istotnych zjawisk, związanych np. ze współbieżnością pojawia się nieregularnie.

4.5.2 Lmbench

Lmbench jest zestawem mikro-benchmarków, czyli benchmarków testujących podstawowe elemnty i wielkości związane z systemem. Ważną jego cechą jest możliwość łatwego rozszerzenia o benchmarki stworzone przez użytkownika.

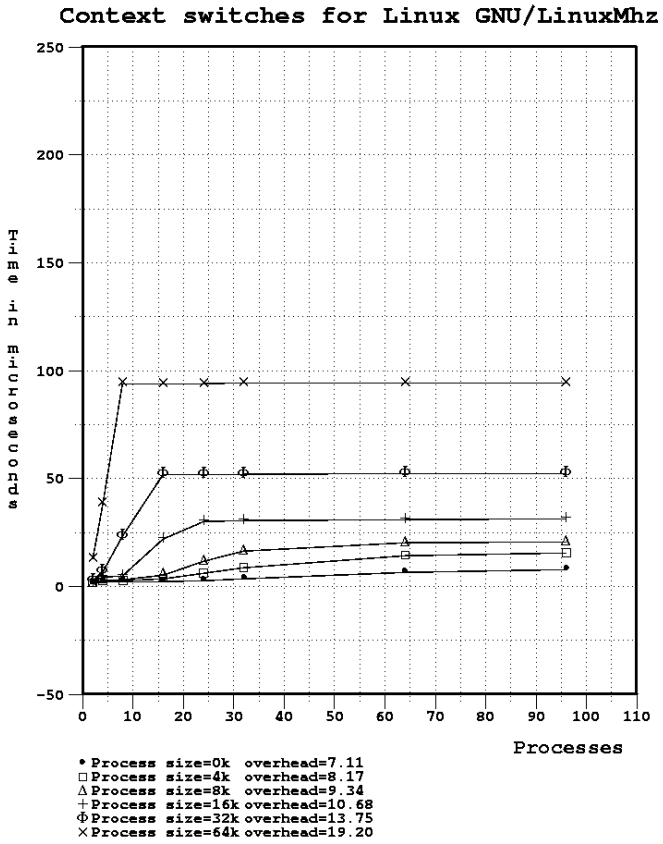
Pakiet można ściągnąć ze strony: **http://www.bitmover.com/lmbench**, obecnie jest to wersja 3.04. Pakiet składa się z trzech głównych części — część mierząca upływ czasu, kod samych benchmarków oraz różnego rodzaju skrypty pozwalające uzskać wyniki działania skryptów w przystępnej formie.

Twórcy pakietu położyli główny nacisk na testowanie następujących elementów:

- Przepustowość: testy mierzące czas odczytu, zapisu, kopiowania z i do pamięci
- Czas reakcji w takich sytuacjach jak: przełączanie kontekstu, ustanawianie połączenia TCP/IP, kasowanie i tworzenie małych plików.

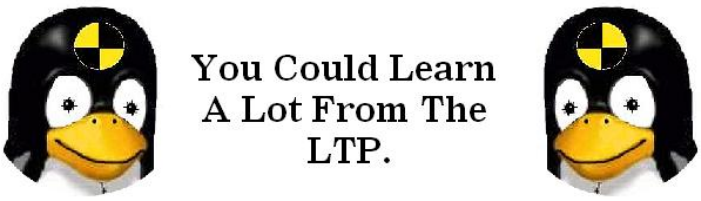
Skrypty dołączane do benchmarków pozwalają min. na generowanie wykresów obrazujących wyniki w postaci plików EPS.

Poniżej pokazujemy przykładowy wydruk uzyskany za pomocą skryptu **graph** zawartego w pakiecie. Na wykresie możemy przeanalizować jak zmienia się czas przełączania kontekstu w zależności od liczby procesów i ich rozmiaru.



Ciekawą opcją jest możliwość łatwego dołączania własnych benchmarków (pisanych w C). Lmbench udostępnia funkcje korzystające z części mierzącej upływ czasu, opis w pliku **bench.h**

4.6 Linux Test Project



Linux Test Project (LTP) jest obszernym zestawem testów, pozwalającym na badanie stabilności, odporności i funkcjonalności Linuxa. Dość pobieżnie potraktowane są za to kwestie wydajności systemu. Aktualne wersje LTP można ściągnąć ze strony **http://ltp.sourceforge.net**.

Z LTP można korzystać na dwa podstawowe sposoby:

- uruchamianie standardowych testów
- pisanie własnych testów z pomocą narzędzi oferowanych przez LTP

Aby uruchomić standardowy zestaw testów, wystarczy — po zainstalowaniu LTP — uruchomić jeden ze skryptów **runltp** lub **runalltests**.

Aby skorzystać z możliwości dołączania własnych programów testowych należy najpierw poznać z grubsza konstrukcję zestawu testów w LTP:

- Podstawową jednostką jest przypadek testowy. Zawiera on zwykle jedną akcję i jego zadaniem jest weryfikacja jej poprawności.
- Program testujący jest wykonywalnym programem, który zawiera jeden lub kilka przypadków testowych. Może być uruchamiany z opcjami.
- Do opisu zestawu testów używamy tagów testowych, wiążących jednoznacznie program testowy i odpowiedni zestaw opcji.

Do przeprowadznia testów LTP udostępnia program do przeprowadzania i kontroli testów, nazywany **pan**. Pobiera on zestaw tagów testpwych i uruchamia je losowo, z podanym zestawem opcji, oraz przechwytuje wyniki testów. Pan utrzymuje plik o rozszerzeniu **.zoo**, w którym przechowuje informacje o aktualnie wykonywanych testach.

LTP udostępnia też zestaw funkcji, które mogą być użyte przy pisaniu własnych testów, ich nagłówki są zawarte w plikach **test.h** i **usctest.h**.

Po stworzeniu programu testowego możemy go uruchomić przy pomocy **pan**:

```
pan -a ltp.zoo -n tut moj_prog 3
```

Pierwsza opcja określa plik w jakim program zapisuje dane o programie testowym, zaś druga — to nazwa pod jaką proces programu będzie znany innym narzędziom.

5 Podsumowanie

Krótka prezentacja niektórych aspektów testowania oraz przykładowych narzędzi służących do projktowania i przeprowadzania testów nie wyczerpuje oczywiście nawet w części tematu. Pokazuje ona jednak, że w procesie tworzenia oprogramowania testowanie jest fazą, której nie można pominąć ani zaniedbać. Znajomość prawidłowych technik i narzędzi może zaoszczędzić wiele czasu i funduszy, a często decyduje o powodzeniu całego projektu. Jest to szczególnie ważne w przypadku systemów operacyjnych, gdzie stabilne działanie jest kluczowym czynnikiem. Poniżej przedstawiamy odnośniki do stron z których korzystaliśmy i na których można znaleźć o wiele więcej informacji na temat testów:

- http://jakarta.apache.org/jmeter
- http://www.bitmover.com/lmbench
- http://ltp.sourceforge.net
- http://oss.sgi.com/projects/kdb
- http://www-306.ibm.com/software/rational
- http://www.tldp.org/HOWTO/Benchmarking-HOWTO.html
- http://ltp.sourceforge.net/coverage/lcov.php