Testowanie jądra Linuksa

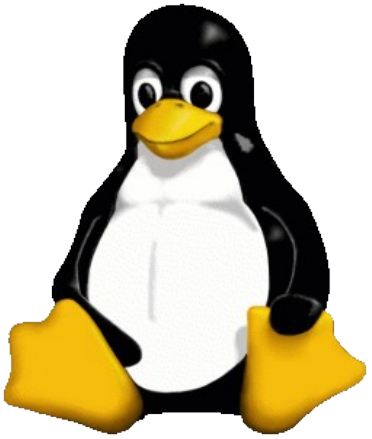
Michał Pilipczuk
Michał Świtakowski
Piotr Wojnarowski

Agenda

- Metodyka testów jądra
- Środowiska i narzędzia używane w praktyce
- Pokrycie kodu
- Prezentacja testów na żywo

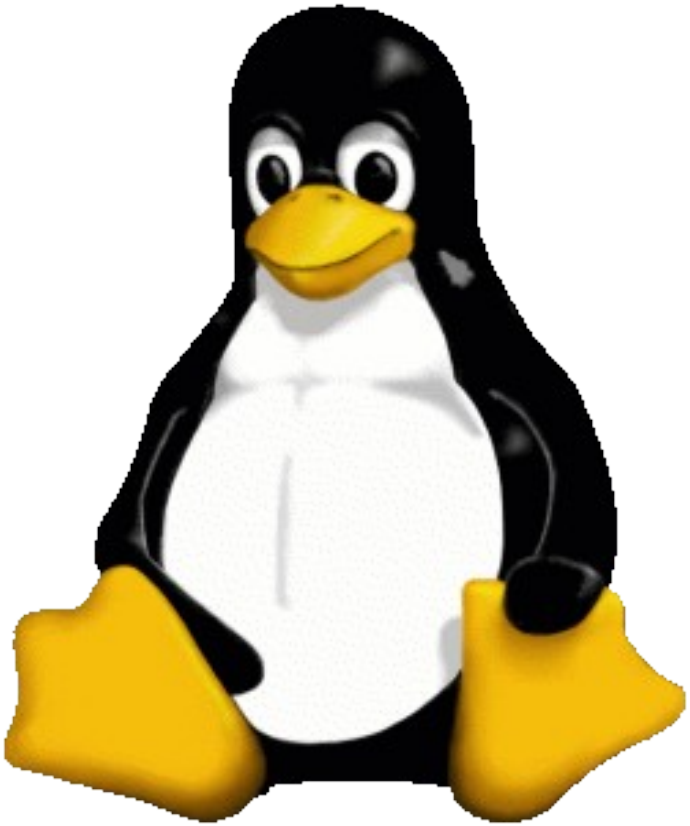
Kiedy Tux był jeszcze młody...

- *„given enough eyeballs, all bugs are shallow”*
  Linus Torvalds
- Oprogramowanie otwarte nie wymaga głębokich testów – wystarczy rzesza użytkowników znajdujących najistotniejsze błędy.
- Sprawdza się przy małych projektach open source.
- Czyli mniej więcej do 2.4.*



Kiedy Tux dorósł...

- Ponad dziesięć milionów linii kodu.
- Odpowiednia liczba funkcji, wywołań systemowych, mutexów...
- Dziesiątki niezarejestrowanych, niedziałających modułów krążących po sieci, działających w zaśmieconym środowisku.
- Więc trzeba było z nim zrobić to...



You Could Learn A Lot From The LTP.

www.linuxtestproject.org

Jak i po co robić testy jądra?

- Jądro działa w trybie dużo bardziej wrażliwym na błędy niż standardowe aplikacje.
- Błędy jądra mają dużo poważniejsze konsekwencje dla stabilności systemu.
- Jądro musi być wytestowane w sposób zorganizowany i metodyczny, wielokrotnie lepiej niż aplikacje działające w trybie użytkownika.
- Tak jak zazwyczaj, testy dzielimy na
  - poprawnościowe
  - wydajnościowe
- W przypadku jądra to rozróżnienie nieco się miesza.

Co byśmy chcieli od testu?

- Automatyzacja, parametryzacja, powtarzalność.
- Pewność poprawności i adekwatności wyników oraz zebranych statystyk.
- Dokładne udokumentowanie środowiska, szczególnie:
  - sprzętu
  - wersji modułów, bibliotek dołączanych do jądra
  - metodologii wykonywania testu

Testy poprawnościowe - przykład

- Moduł stats – prosty moduł do zbierania statystyk.
- Ok. 1500 linii, czyli stosunkowo krótki.
- Duża współbieżność, silne korzystanie z kiepsko napisanego podsystemu proc.
- 63 proste funkcje wewnątrz, duża część eksportowanych na zewnątrz.
- Dwa pytania:
  - Jak sprawdzić, że o żadnej funkcji nie zapomnieliśmy?
  - Czy moduł zachowa się poprawnie przy błędach proc-a, dużym obciążeniu systemu, błędach funkcji systemowych?

Testy białej skrzynki

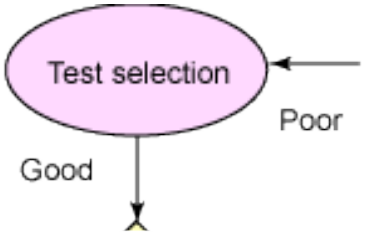
- Staramy się pokryć wszystkie gałęzie kodu.
- A także linie, funkcje, wołania systemowe itp.

LTP GCOV extension - code coverage report

Current view:	directory	Executed lines:	63612 / 256879
Test:	ppc64branch.info	Branches taken:	10110 / 72549
Date:	2004-01-08	Executed functions:	4129 / 12754
Code covered:	22.8 %		

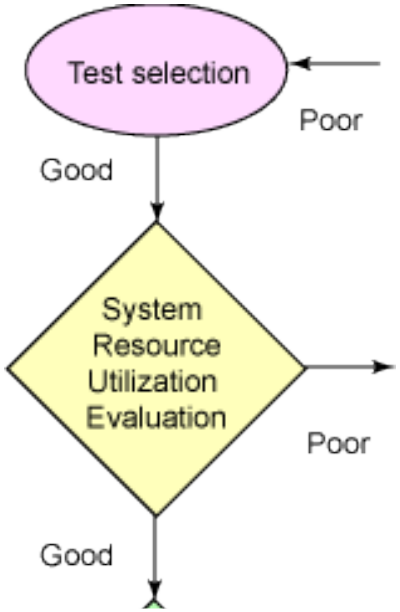
Directory name	Coverage			
	Total	Statement	Branch	Function
arch/ppc64/kernel	36.1 %	38.9 %	23.4 %	44.9 %
arch/ppc64/lib	97.4 %	100.0 %	83.3 %	100.0 %
arch/ppc64/mm	41.0 %	43.7 %	28.0 %	52.3 %
arch/ppc64/oprofile	60.0 %	66.7 %	No branches	50.0 %
arch/ppc64/xmon	0.1 %	0.1 %	0.0 %	0.9 %
drivers/base	35.0 %	37.2 %	22.9 %	40.6 %
drivers/base/power	7.4 %	10.0 %	0.0 %	0.0 %
drivers/block	32.5 %	34.5 %	23.3 %	43.7 %
drivers/cdrom	3.4 %	4.6 %	0.3 %	6.8 %
drivers/char	38.4 %	42.6 %	25.2 %	45.7 %
drivers/gcov	57.9 %	63.0 %	40.4 %	75.0 %
drivers/input	37.8 %	43.9 %	19.3 %	54.1 %
drivers/input/keyboard	40.3 %	43.4 %	32.1 %	47.1 %

Projektowanie testu



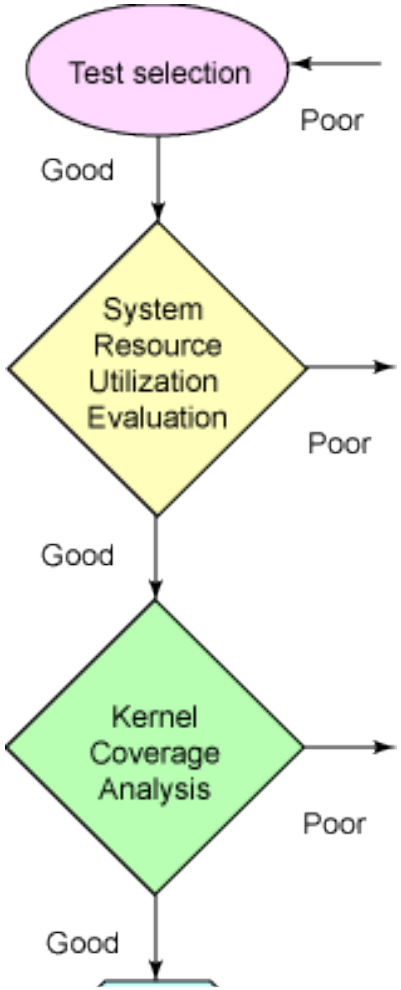
- Przy projektowaniu testu należy pamiętać o rzeczach standardowych, jak:
  - pokrycie wszystkich gałęzi kodu
  - sprawdzenie przypadków brzegowych
  - sprawdzenie zachowania przy błędach systemowych
  - sprawdzenie współbieżności
- Jak i o rzeczach mniej standardowych:
  - Co się dzieje przy dużym obciążeniu zasobów systemowych?

Zasoby systemowe



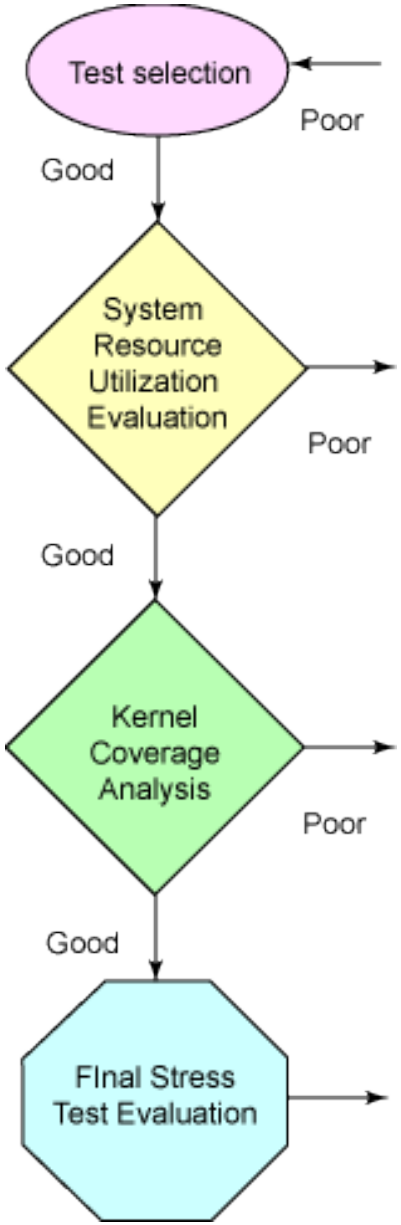
- Staramy się obciążyć kluczowe zasoby systemu:
  - procesor
  - pamięć
  - wejście/wyjście
  - sieć
- Nie wystarczy obciążyć jednej na raz!
- Przesadzenie z obciążeniem jednego zasobu skutkuje brakiem obciążenia innego.
- Obciążenie należy wyważyć metodą prób i błędów.

Pokrycie kodu



- Sprawdzamy ile kodu pokryły nasze testy.
- Dużo narzędzi udostępnionych przez LTP:
  - gcov
  - lcov
- Statystyka tak naprawdę mocno myląca – przecież nawet 100% pokrycia wcale nie oznacza poprawności np. współbieżności!!!

Ostateczny test obciążenia



- Zalecane co najmniej 24 godziny.
- Utrzymywane przez cały czas monitorowanie obciążenia systemu.
- Służy do testowania między innymi:
  - dziwnych przeplotów,
  - nieoczekiwanych kombinacji zajętości zasobów
  - oraz innych dziwnych sytuacji, które nie występują przy krótkich testach poprawnościowych

Metody monitoringu systemu

- Problem:
  - Jak sprawdzić, jak w rzeczywistości radził sobie testowany kawałek kodu?
- Rozwiązanie:
  - Monitoring systemu przez:
    - Wewnętrzne logi aplikacji testującej
    - Zewnętrzne programy monitorujące system:
      - `top`
      - `sar`

top

- Monitoruje jedynie procesy
- Do LTP jest dołączony ulepszony `top`:
  - sprawdzanie CPU
  - sprawdzanie pamięci
  - sprawdzanie swap
  - snapshoty co określony czas

```
top - 13:05:27 up  3:52,  3 users,  load average: 0.20, 0.14, 0.10
Tasks: 149 total,   1 running, 148 sleeping,   0 stopped,   0 zombie
Cpu(s):  1.3%us,  0.5%sy,  0.0%ni, 98.2%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1032904k total,   952356k used,    80548k free,    86848k buffers
Swap:  2031608k total,        8k used,  2031600k free,   353492k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3295 michal    20   0  234m  99m  27m S    2  9.8   5:38.34 firefox-bin
 3007 root      20   0  426m 115m  22m S    1 11.5   6:12.36 X
 3798 michal    20   0  343m 133m  58m S    1 13.2   3:18.90 simpress.bin
 3394 michal    20   0 93060  19m  11m S    0  2.0   0:01.20 gnome-terminal
 3457 michal    20   0 42156  15m  10m S    0  1.5   0:03.68 file-roller
 4923 michal    20   0  2272 1032  784 R    0  0.1   0:00.02 top
    1 root      20   0  2112  652  560 S    0  0.1   0:00.95 init
    2 root      15  -5     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      RT  -5     0    0    0 S    0  0.0   0:00.00 migration/0
    4 root      15  -5     0    0    0 S    0  0.0   0:00.18 ksoftirqd/0
    5 root      RT  -5     0    0    0 S    0  0.0   0:00.00 watchdog/0
    6 root      RT  -5     0    0    0 S    0  0.0   0:00.02 migration/1
    7 root      15  -5     0    0    0 S    0  0.0   0:00.27 ksoftirqd/1
    8 root      RT  -5     0    0    0 S    0  0.0   0:00.00 watchdog/1
    9 root      15  -5     0    0    0 S    0  0.0   0:00.05 events/0
   10 root      15  -5     0    0    0 S    0  0.0   0:00.19 events/1
   11 root      15  -5     0    0    0 S    0  0.0   0:00.00 khelper
   64 root      15  -5     0    0    0 S    0  0.0   0:00.04 kblockd/0
   65 root      15  -5     0    0    0 S    0  0.0   0:00.08 kblockd/1
   67 root      15  -5     0    0    0 S    0  0.0   0:00.00 kacpid
   68 root      15  -5     0    0    0 S    0  0.0   0:00.00 kacpi_notify
  156 root      15  -5     0    0    0 S    0  0.0   0:00.00 cqueue
  158 root      15  -5     0    0    0 S    0  0.0   0:00.00 ksuspend_usbd
```

sar

```
Linux 2.6.27.3 (harry.dom)      07.12.2008

00:00:01        CPU     %user     %nice   %system   %iowait    %steal     %idle
00:10:01        all     18,25      0,33      1,22      0,74      0,00     79,45
00:20:01        all     28,69      0,00      2,12      0,04      0,00     69,16
00:30:01        all     19,14      0,00      1,26      0,05      0,00     79,55
Średnia:        all     22,02      0,11      1,53      0,28      0,00     76,06

09:12:59          LINUX RESTART

09:20:01        CPU     %user     %nice   %system   %iowait    %steal     %idle
09:30:01        all      3,55      0,00      0,66      0,10      0,00     95,69
09:40:01        all      7,56      0,00      0,57      0,28      0,00     91,59
09:50:01        all      1,22      0,00      0,34      0,00      0,00     98,43
10:00:01        all      0,78      0,00      0,26      0,06      0,00     98,90
10:10:01        all      1,26      0,00      0,35      0,01      0,00     98,38
10:20:01        all      3,00      0,32      0,57      0,01      0,00     96,10
10:30:01        all      1,74      0,00      0,39      0,00      0,00     97,88
10:40:01        all      4,17      0,00      0,81      0,86      0,00     94,17
10:50:01        all      0,86      0,00      0,22      0,02      0,00     98,90
11:00:01        all      2,89      0,24      0,52      0,01      0,00     96,33
11:10:01        all      3,61      1,57      0,91      4,47      0,00     89,45
11:20:01        all      3,60      0,67      0,75      0,04      0,00     94,95
11:30:01        all      1,66      0,12      0,39      0,00      0,00     97,83
11:40:01        all      2,77      0,00      0,57      0,14      0,00     96,52
11:50:01        all      5,94      0,00      0,90      0,13      0,00     93,03
12:00:01        all      3,46      0,00      0,65      0,05      0,00     95,83
12:10:01        all      1,96      0,00      0,42      0,01      0,00     97,60
12:20:01        all      2,86      0,32      0,61      0,06      0,00     96,16
12:30:01        all      3,97      0,00      0,68      0,02      0,00     95,32
12:40:01        all      6,16      0,00      1,05      0,03      0,00     92,76
12:50:01        all      3,77      0,00      0,70      0,02      0,00     95,50
13:00:01        all      3,48      0,00      0,71      0,01      0,00     95,79
13:10:01        all      7,09      0,00      1,03      0,32      0,00     91,56
13:20:01        all      3,99      0,34      0,77      0,03      0,00     94,87
Średnia:        all      3,39      0,15      0,62      0,28      0,00     95,56
```

- Zbiera, zapisuje i zapamiętuje statystyki o systemie
- Co określony czas zrzut wszystkiego co się da
- Można z niego wycisnąć wszystko, `man sar` zrzucony do pliku ma 726 linii.

Testy wydajnościowe

- W gruncie rzeczy podobne do poprawnościowych, tylko z dłuższym okresem czasu.
- Puszczamy test na ustalonej maszynie na długi okres czasu – np. miesiąc i oglądamy co określony czas statystyki wydajności wykonywanych testów.
- Uwaga na narzut analizy pokrycia i metod pomiarowych!!!
- Musimy zadbać o:
  - odpowiednią dokumentację środowiska,
  - obciążenie zasobów systemowych,
  - odpowiedni monitoring wydajności w czasie testu.

Case study – testy IBM

- W 2003 roku laboratoria IBM przeprowadziły kompleksowe, długoterminowe testy jądra 2.4.19 używając narzędzi LTP.
- Cel: sprawdzenie niezawodności oraz wydajności jądra Linuxa.
- Hardware:
  - 2 procesory POWER4+ 1.2GHz
  - 8GB pamięci RAM
  - oraz wiele innych dokładnych danych
- Software:
  - SuSE SLES 8 z Service Pack 1
  - Jądro 2.4.19-ul1-ppc64-SMP

Metodyka testu

- Trzy testy: 30, 60 i 90-dniowy.
- Wykorzystanie standardowych testów LTP oraz własnych testów wydajnościowych
- Co 10 sekund zrzut `top`, co określony czas przegląd statystyk wykonania testów
- Przez cały czas bardzo duże (99%) obciążenie CPU oraz duże (80-90%) obciążenie pamięci.
- W teście 60-dniowym testowanie obciążenia pamięci swap.

Testowane aspekty

- We wstępnym, 24-godzinnym teście standardowym LTP testowano:
  - Obciążanie systemu plików.
  - Obciążanie dysku.
  - Obciążanie systemu zarządzania pamięcią.
  - Obciążanie IPC (pipe'y, semafory).
  - Działanie schedulera.
  - Poprawność funkcjonalności komend.
  - Poprawność funkcjonalności wołań systemowych.

Testowane aspekty c.d.

- A następnie uruchomiono duży, własny test IBM, ponadto sprawdzający:
  - Obciążanie NFS-a
  - Obliczenia zmiennoprzecinkowe
  - Wątki
  - Sieć
- Ogółem przez 6 miesięcy system bez przerwy działał na pełnych obrotach wykonując współbieżnie testy obciążenia jego zasobów.
- A jakie wyniki?

Wyniki

- 0 błędów krytycznych (panic, Oops).
- Ok. 95% poprawnych wyników, niepoprawne z powodu braku zasobów.
- Wszystkie procesy zmieściły się w oczekiwanym czasie działania.
- Wydajność systemu nie spadała wraz z testami.
- Wszystko to udokumentowane, sprawdzone i powtarzalne.

Podsumowanie

- Pomysł formalnej weryfikacji
  - Na szczęście jeszcze nikt na niego nie wpadł.
  - Problem z formalizacją abstrakcji sprzętowej.
  - Problem z formalizacją abstrakcji współbieżności.
  - 10.000.000 linii kodu?!
- A tak na serio:
  - Myślmy pragmatycznie

Ciekawostki

- LTP po raz pierwszy użyto w szerszym zakresie przy rozwijaniu jąder 2.5.*. Użyto wtedy tzw. nocnych testów regresyjnych.
- Oprócz standardu oferowanego przez LTP są jeszcze inne programy testujące. Najciekawsze są te testujące pamięć:
  - Cerberus Test Control System w przypadku sprzętu kiepskiej jakości może doprowadzić do jego spalenia. Cpuburn jest nawet napisany w asemblerze, żeby zmaksymalizować wydzielanie ciepła.

Przykładowe testy

- Testy pamięci:
  - Idealny test: wpisać do komórki 0, do wszystkich pobliskich 1, sprawdzić, czy nadal jest 0. Wymaga precyzyjnej wiedzy o działaniu hardware'u.
  - Memtest86: bootuje zamiast Linuxa, długie testy. Algorytm „modulo-x”: wypełnić co 20 pozycję wzorcem, resztę uzupełnieniem wzorca, sprawdzić co 20, czy nadal pasują.
  - Memory test script: bardzo rygorystyczny, łączący odwołania do pamięci przez CPU i DMA (obchodzące CPU), ponoć może spalić PC
  - Memtester

Przykładowe testy 2

- Volanomark- mierzy wydajność serwera. Tworzy grupy po 20 klientów i sprawdza, jak szybko wysyłają wiadomość do całej grupy.
- VM regress- test pamięci wirtualnej
- Fileio- testy wydajnościowe I/O. Tworzy zadaną ilość plików zadanego rozmiaru i uruchamia wątki, które z nich korzystają

Linux Test Project

- Pierwszy systematyczny zestaw testów Linuxa, stworzone przez Silicon Graphics Inc.
- Sprawdza system plików, operacje I/O dysku, zarządzanie pamięcią, ipc, sterowniki, planistę (scheduler), zawołania systemowe, łączenie z siecią
- Dawniej (jądro 2.3) - 100 testów, teraz 3000

Directory	Coverage
fs	52.9%
include/asm	50.9%
include/linux	60.0%
include/net	57.6%
ipc	56.4%
kernel	39.1%
lib	43.2%
mm	52.7%
net	65.7%
security	51.9%

Metodologia LTP

- Testy zdrowia:
  - Sprawdzają niezawodność jądra
  - Powinny działać poniżej 24h
  - Testowanie wszystkich patchy i wydań jądra
  - Większość z nich szybkie testy z oczywistym wynikiem
  - Wyniki Pass lub Fail
- Testy obciążeniowe:
  - Większe obciążenie
  - Poniżej 6h
  - Mają wyłapać „zawieszanie” i „wywalanie” się systemu
- Wytrzymałościowe
  - Od 6h do tygodni

Linux Standard Base Tests

- LSB - wspólne standardy różnych dystrybucji Linuxa, większa kompatybilność oprogramowania
- Liczne testy sprawdzające zgodność z LSB:
  - Czy pakiety są zgodne z LSB
  - Czy używają tylko zgodnych symboli
  - Czy dostępne są wszystkie wymagane polecenia
  - Czy w bibliotekach są wymagane symbole

Bootchart - zanim Linux się uruchomi

- „The challenge is to create a single poster showing graphically what is going on during the boot, what is the utilization of resources, how the current boot differs from the ideal world of 100% disk and CPU utilization, and thus, where are the opportunities for optimization.”- *Owen Taylor, Fedora mailing list, Listopad 2004*
- Skrypt działający w fazie *init,* zbierający informacje z /proc + aplikacja w Javie lub webowa, generująca graficzne przedstawienie wyników
- Pokazuje, w jakim czasie jaki proces był w jakim stanie, oraz statystyki zużycia zasobów systemowych (CPU, dysk)

Bootchart, jak działa

- Bootchart uruchamiany zamiast procesu init (należy poprawić konfigurację Grub/Lilo, RPM spróbuje zrobić to automatycznie)
- Uruchamia loggera w tle, po czym uruchamia init
- Pobiera z /proc/stat, /proc/[PID]/stat, /proc/diskstat dane o procesach i zasobach
- Dane zapisuje w wirtualnej pamięci /tmp
- Wykrywa wywołanie konkretnych procesów, np. kdm_greet, co oznacza koniec bootowania - zapisuje logi
- Może „przegapić” wyjątkowo krótko żyjące procesy, jeśli to ważne, używać „zliczania procesów”- CONFIG_BSD_PROCESS_ACCT_V3 + psacct/acct

Kwestia pokrycia

- Pokrycie - jaka część kodu została wykonana, gdzie optymalizować
- Dobry test powinien dawać jak największe pokrycie
- Metody analizy pokrycia:
  - Pokrycie poleceń (statement coverage)
  - Pokrycie gałęzi kodu (branch coverage)
  - Pokrycie warunków (condition cov.)
  - Pokrycie wejść i wyjść (exit/entry cov.)

Metody

- Pokrycie poleceń - ile razy były wykonane bloki kodu (blok - część kodu między rozgałęzieniami)
- Nie zawsze skuteczne:

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

- Pokrycie gałęzi- które spośród możliwych ścieżkę były wybierane i jak często. Czasem potrzebne jest też analizowanie złożonych warunków:

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

Analiza pokrycia za pomocą gcov

- Współpracuje z gcc
- Tylko pokrycie poleceń, gałęzi i wywołań funkcji
- Używamy gcc z opcjami '-fprofile-arcs -ftest-coverage'
- Dodatkowe liczniki, graf przepływów, powstaje pliki .gcno, .gcda
- Pliki nagłówkowe zawierające kod także będą sprawdzone
- Możemy wywoływać kilka razy, jeśli nie usuniemy plików .gcda, wyniki będą się kumulować
- Uruchamiany program, statystyki się liczą
  - 90.00% of 10 source lines executed in file tmp.c
  - 80.00% of 5 branches executed in file tmp.c
  - 80.00% of 5 branches taken at least once in file tmp.c
  - 50.00% of 2 calls executed in file tmp.c
- Plik .gcov

```
        -:    0:Source:tmp.c
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:int main (void)
function main called 1 returned 1 blocks executed 75%
        1:    4:{
        1:    5:  int i, total;
        -:    6:
        1:    7:  total = 0;
        -:    8:
       11:    9:  for (i = 0; i < 10; i++)
branch  0 taken 91% (fallthrough)
branch  1 taken 9%
       10:   10:    total += i;
        -:   11:
        1:   12:  if (total != 45)
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
    #####:   13:    printf ("Failure\n");
call    0 never executed
        -:   14:  else
        1:   15:    printf ("Success\n");
call    0 called 1 returned 100%
        1:   16:  return 0;
        -:   17:}
```

gcov a testy jądra

- Różnice (kernel się nie kończy, nie linkuje bibliotek gcc, których potrzebuje gcov)
- Gcov-kernel patch od LTP, umożliwia badanie pokrycia testów jądra:
  - Tworzenie /proc/gcov
  - Umożliwia badanie modułów (w jądrze 2.4 wymagał dodatkowego patcha do modutils)
  - Moduł skompilowany z odpowiednimi flagami, domyślnie dane usuwane przy odładowaniu
  - Plik /proc/gcov/vmlinux

lcov

- Graficzna nakładka na gcov, skrypty Perl zbierające wyniki gcov do pliku .info i generujące HTML lub .png
- Manipulowanie .info: extract / remove PATTERN, łączenie.
- Ułatwia nawigację - drzewo HTML dla plików i katalogów
- Stworzone z myślą o kernelu, ale można też używać z innymi programami (gcc)

Problemy z wynikami pokrycia

- Równoległe wykonanie na różnych procesorach - aktualizacja nie jest atomowa, mogą wystąpić niespójności w wynikach, wówczas gcov może podać ujemne wyniki (inne źródła - zaburzenia do 3%)
- Pokrycie „na żywo” - także może spowodować niespójność i ujemne wyniki
- Pierwsze linie funkcji „inline” liczone są czasem dwa razy
- Makra, optymalizacja i linijki zawierające kilka poleceń

Narzuty

- Maszyna testowa - Pentium® II 400MHz, 256 MB RAM, Red Hat 7.1 (Linux 2.5.50) z gcc 2.96 (2003 rok)
- 1,3 MB pamięci na liczniki (dla gcc 3.1 - 2,6 MB)
- 3% dłuższe wykonanie, głównie w przestrzeni jądra
- 14% kodu jądra jest wywoływane w celu zebrania informacji

Testy na żywo

- Analiza wyników z Bootchart na maszynie wirtualnej i prawdziwym komputerze
- Readahead – jak działa i co właściwie daje
- Wyścig w `remove_proc_entry`, czyli dlaczego nie można zaniedbać żadnego przypadku
- Wdrożenie własnego testu do LTP

Bibliografia

- http://en.wikipedia.org/wiki/Code_coverage
- http://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov
- http://ltp.sourceforge.net/documentation/how-to/UsingCodeCoverage.pdf
- http://ltp.sourceforge.net/coverage/
- P. Larson, N. Hinds, H. Franke, and R. Ravindran. Improving the linux test project with kernel code overage analysis. In 2003 Ottawa Linux Symposium, July 2003.
- Subrata Modak, Balbir Singh, Building a Robust Linux kernel piggybacking The Linux Test Project
- http://www.linuxbase.org/test/
- http://www.ibm.com/developerworks/linux/library/
- http://ltp.sourceforge.net/documentation/how-to/ltp.php
- http://dailypackage.fedorabook.com/index.php?/archives/59-Wednesday-Why-Readahead.html
- http://www.bootchart.org/docs.html