Automatyczne testowanie jądra Linuksa

Marcin Lis, Michał Pękała, Michał Zydor

wydział Matematyki Informatyki i Mechaniki
Uniwersytetu Warszawskiego

23 grudnia 2008

Składniki

1. Metodologia tworzenia i przeprowadzania testów
   - idea, założenia
   - przeprowadzanie testów
   - metodologia tworzenia testów
2. Testy poprawności i wydajności
   - Memtest86
   - Charles Cazabon's memtester
   - Doug Ledford's Memory Test Script
   - Cerberus Test Control System
   - cpuburn
3. bootchart
4. Linux Test Project
   - idea
   - historia
   - funkcjonalności
   - skuteczność
   - Własne testy

Definicja

„Testowanie oprogramowania jest wykonaniem kodu dla kombinacji danych wejściowych i stanów w celu wykrycia błędów"

Robert V. Binder, Testing Object-Oriented Systems. Models, Patterns, and Tools

Do czego potrzebne?

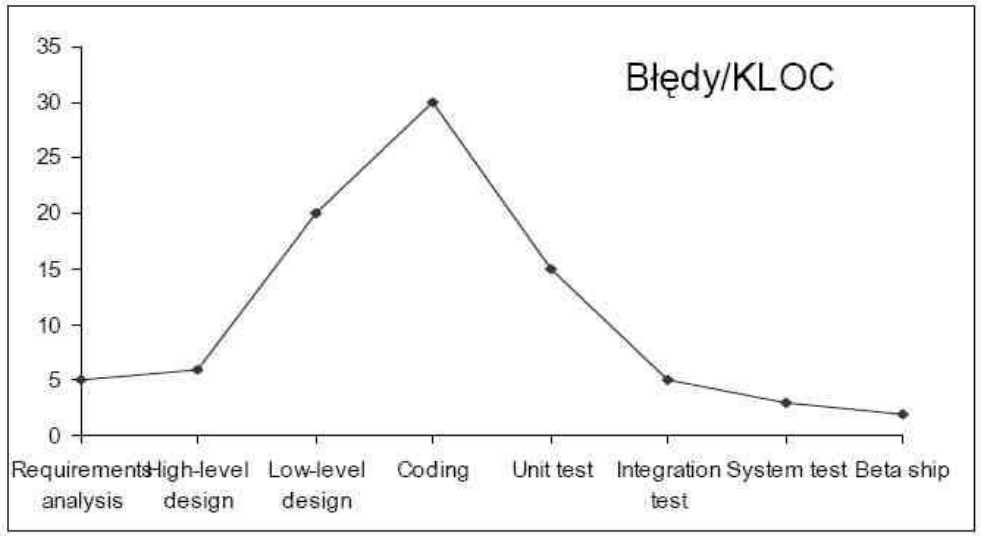
1. tworzenie i rozwój istniejącego oprogramowania,
2. dobór oprogramowania do warunków sprzętowych,
3. konfiguracja oprogramowania - np. dobór bibliotek do wersji jądra systemu.

Podsumowując:

Dla poprawnego i efektywnego działania posiadanego sprzętu i oprogramowania.

Warto przeprowadzać

Dla nowo tworzonych lub stale rozwijanych projektów koszt naprawy błędu rośnie wykładniczo(!) w kolejnych etapach wytwarzania oprogramowania.



W wielu projektach testowanie jest realizowane na samym końcu...

Rodzaje testów

- poprawnościowe

Rodzaje testów

- poprawnościowe
- wydajnościowe

Rodzaje testów

- poprawnościowe
- wydajnościowe
- długotrwałego działania

Rodzaje testów

- poprawnościowe
- wydajnościowe
- długotrwałego działania
- obciążenia-przeciążenia

Różne poziomy

I Testy jednostkowe

1. popularyzowane przez Extreme Programming,
2. każda jednostka/moduł jest testowana oddzielnie (wg specyfikacji),
3. testy pisane przed napisanie modułu,

II Testy integracyjne

1. czy moduły współpracują ze sobą?,
2. dużo trudniej zidentyfikować klasy równoważności,
3. pojawiają się problemy skali,
4. często wykrywane błędy specyfikacji a nie błędy integracji

III Testy systemowe

1. testy funkcjonalności:
   (a) warunki testowe,
   (b) skrypty testowe,
   (c) dane testowe,

Testy systemowe - testy pozafunkcjonalne

- facility testing – Czy system zapewnia wszystkie wymagane funkcje?
- volume testing – Czy system radzi sobie z dużą ilością danych?
- stress testing – Czy system radzi sobie z dużym obciążeniem?
- endurance testing – Czy system zachowuje parametry w długim okresie?
- usability testing – Czy system jest łatwy w użyciu?
- security testing – Czy system wytrzymuje ataki?
- performance testing – Jak dobry jest czas reakcji systemu?
- storage testing – Czy pojawiają się problemy ze składowaniem danych?
- configuration testing – Czy system działa na wszystkich platformach?
- installability testing – Czy da się skutecznie zainstalować system?
- reliability testing – Jak zmienia się niezawodność systemu w czasie?
- recovery testing – Jak skutecznie system podnosi się po awarii?
- serviceability testing – Czy daje się pielęgnować system?
- documentation testing – Czy dokumentacja jest dokładna? Użyteczna?
- operations testing – Czy instrukcje operatorów są poprawne?

Jakie powinny być testy?

Powtarzalne – np. żeby sprawdzić, czy wszystko jest ok po usunięciu nieprawidłowości

Systematyczne – pokrywające cały zakres funkcjonalności, reprezentatywne dla właściwych zadań systemu, losowe testy to za mało

Dokumentowane – dla tworzenia statystyk i podejmowania daleko idących decyzji

Różne podejścia

- czytanie kodu,
  – może dać względnie szybkie rezultaty jeśli kodu mało, a czytających wielu,
- ręczne sprawdzenie czy wszystko działa,
  – ważne, szybkie efekty, dotyczy zazwyczaj tylko bardzo znikomej części funkcjonalności,
- sprawdzenie kompatybilności z szerokim środowiskiem,
  – czasochłonne, nawet jeśli coś jest nie tak, łatwo to przegapić bez znajomości specjalistycznych szczegółów,
- automatyczne testy,
  – w przypadku Linux: tzw. „testy systemu" i „testy aplikacji" (funkcjonalności towarzyszące) - najobszerniejsze, potencjalnie najbardziej miarodajne,

Testy automatyczne - w stronę ideału

1. powtarzalne, systematyczne, dokumentowane
2. pozwalają na dokładną analizę,
3. znacznie szybsze od „ręcznych"
4. kompleksowo pokrywają oprogramowanie, nie tylko „trudne przypadki",

Podstawowe techniki

- dobry kompilator,
- obsługa błędów w kodzie, ostrzeżenia,
- ręczne inicjowanie parametrów,
- debugowanie.

Czarna skrzynka - testy funkcjonalności

Generowanie przypadków testowych na podstawie specyfikacji, nieistotny kod programu.

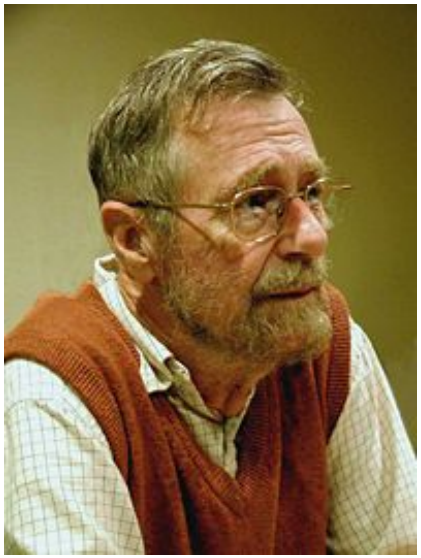
zalety: Unikamy przyjmowania tych samych założeń co programista. Dane testowe są niezależne od implementacji. Wyniki można interpretować bez wnikania w szczegóły implementacyjne.

sugestie: Dla każdego komponentu przypadki testowe: „czego wymaga", „co modyfikuje", „na co wpływa". Dobrze jest określić warunki brzegowe, mogą się przydać testy niepoprawnych danych wejściowych,

Przezroczysta skrzynka - testy strukturalne

Pokrycie kodu testami

- Badanie kodu i testowanie ścieżek w kodzie
  ...ponieważ testowanie czarnej skrzynki nigdy nie gwarantuje, że wykonaliśmy wszystkie fragmenty kodu
- Kompletność ścieżek:
  Zestaw testów pokrywa komplet ścieżek jeśli każda ścieżka w kodzie jest wykonana co najmniej raz w zestawie testów.
- To nie to samo, co stwierdzić, że każda instrukcja kodu jest wykonana!



Edsger Dijkstra

„Testowanie może ujawnić obecność błędów, ale nigdy ich brak”

Testy poprawności i wydajności

Często błędami pojawiającymi się w czasie pracy obarczamy system, oprócz niego jednak najczęstszą przyczyną błędów są problemy z pamięcią wolnego dostępu (cache RAM).

memtest86 - charakterystyka

- Memtest86 to rygorystyczny, niskopoziomowy tester pamięci wolnego dostępu, ściśle związany z architekturą Intel/AMD x86.
- Uruchamia się go jak system operacyjny -ładowanie spośród opcji w GRUB'ie lub z dysku startowego.
- Przeprowadza znacznie dokładniejsze (i bardziej czasochłonne) testy pamięci niż BIOS. Sprawdza wszystkie dostępne sektory pamięci RAM - niezależnie od ich przeznaczenia.
- Można nadzorować działanie programu online.

memtest86 - co zrobić gdy już jest błąd?

Trzeba zorientować się na której kości RAM wystąpił błąd:

1. usunąć kość i sprawdzić - najłatwiej,
2. gdy nie można usuwać kości - rotacja (więcej niż dwie)
3. gdy to nic nie daje - podmienić na inną
4. unikać alokacji - konstrukcja szablonów dla Linuxa (BadRam)

memtest86 - czy wiarygodny?

- Czasami poszukiwanie rozmiaru dostępnej pamięci może powodować problemy, np. z segmentami pamięci zaczynającymi się powyżej 3GB.
- Bywa, że pamięć po prostu nie współpracuje z Athlonem.
- Memtest nie radzi sobie z błędami procesora.

memtester - charakterystyka

- Narzędzie testujące podzespoły pamięci, podobne do memtest86.
- Alokuje tak wiele stron pamięci RAM jak to możliwe, blokuje fizyczną pamięć i wykonuje na niej testy.
- Zastosowanie w systemach unixo-podobnych - proces w trybie użytkownika.
- Łatwa kompilacja i uruchamianie:

  $ memtester <memory> [runs]
  <memory> - rozmiar pamięci do testowania w MB
  runs- liczba powtórzeń testu

- Pojedynczy proces może w linuxie zaalokować co najwyżej połowę fizycznej pamięci, dobrze wykonywać testy równolegle.

memtester - sprawdza się

- gdy architektura inna niż x86 (np. Mac)
- gdy chcemy szybko sprawdzić pamięć, bez restartowania komputera

```
File  Edit  View  Terminal  Tabs  Help
root@scherzo:~/Program_Files/test_suits/memtester-4.0.8# ./memtester 200
memtester version 4.0.8 (32-bit)
Copyright (C) 2007 Charles Cazabon.
Licensed under the GNU General Public License version 2 (only).

pagesize is 4096
pagesizemask is 0xfffff000
want 200MB (209715200 bytes)
got  200MB (209715200 bytes), trying mlock ...locked.
Loop 1/1:
  Stuck Address       : ok
  Random Value        : ok
  Compare XOR         : ok
  Compare SUB         : ok
  Compare MUL         : ok
  Compare DIV         : ok
  Compare OR          : ok
  Compare AND         : ok
  Sequential Increment: ok
  Solid Bits          : ok
  Block Sequential    : ok
  Checkerboard        : ok
  Bit Spread          : ok
  Bit Flip            : testing 135
```

```
File  Edit  View  Terminal  Tabs  Help
[sudo] password for michal:
root@scherzo:~# cd Program_Files/test_suits/memtester-4.0.8/
root@scherzo:~/Program_Files/test_suits/memtester-4.0.8# ./memtester 200 1
memtester version 4.0.8 (32-bit)
Copyright (C) 2007 Charles Cazabon.
Licensed under the GNU General Public License version 2 (only).

pagesize is 4096
pagesizemask is 0xfffff000
want 200MB (209715200 bytes)
got  200MB (209715200 bytes), trying mlock ...locked.
Loop 1/1:
  Stuck Address       : ok
  Random Value        : ok
  Compare XOR         : ok
  Compare SUB         : ok
  Compare MUL         : ok
  Compare DIV         : ok
  Compare OR          : ok
  Compare AND         : ok
  Sequential Increment: ok
  Solid Bits          : ok
  Block Sequential    : ok
  Checkerboard        : testing   1
```

Memory Test Script - charakterystyka

- Bardziej rygorystyczny, testuje pamięć przez kombinację dostępu w trybie DMA i procesora.
- Bada jak zachowują się dane w pamięci RAM podczas ich transferu z maksymalną szybkością (możliwie pełne wykorzystanie przepustowości).
- Skrypt pozwala ocenić czy wbudowany kontroler pamięci (odpowiedzialny za przepływ danych) funkcjonuje prawidłowo.

File Edit View Terminal Tabs Help

```
michal@scherzo:~$ free -m
             total       used       free     shared    buffers     cached
Mem:          1000        986         14          0          2        421
-/+ buffers/cache:        562        438
Swap:         1906        595       1310
michal@scherzo:~$ free -m
             total       used       free     shared    buffers     cached
Mem:          1000        551        449          0          1         27
-/+ buffers/cache:        522        478
Swap:         1906        606       1299
```

Memory Test script -zaczyna się ryzyko

„Be careful with this one, I've been told it has been known to damage cheap hardware, or could even make your computer so hot it catches on fire."

```
File  Edit  View  Terminal  Tabs  Help
        -n <number_of_passes>
                Default: 20
        -m <megs_per_copy>
                Default: 4 * sizeof source .gz files,
                         5 * sizeof source .bz2 files
        -c <number_of_copies>
                Default: physical ram size * 1.5 / megs_per_copy
        -p      Toggles whether or not we will extract and diff all the
                source trees in parallel or serial.
                Default: serial
        -i      Just parse the arguments and say what figures we came
                up with and then exit.
root@scherzo:~/Program_Files/test_suits/test_script# sudo cp linux.tar.bz
2 /tmp/
root@scherzo:~/Program_Files/test_suits/test_script# ./memtest.sh -x -c 1
 -n 2
TEST_DIR:               /tmp
SOURCE_FILE:            linux.tar.bz2
NR_PASSES:              2
MEGS_PER_COPY:          240
NR_COPIES:              1
PARALLEL:               no
COMPRESS_RATIO:         5
COMPRESS_FLAG:          j
COMPRESS_PROG:          /usr/bin/pbzip2
EXTRACT:                no

Creating comparison source...done.
Starting test pass #1: unpacking, comparing, removing, done.
Starting test pass #2: unpacking, comparing, removing,
```

Memory Test Script -test wytrzymałości?

Test Douga Ledforda pozwala też diagnozować problemy związane ze źródłem mocy, jeżeli dodatkowo uruchomione np.:

- operacje na dyskach twardych,
- aplikacje używające karty graficznej,
- napęd CD (np. nagrywanie),
- aktywne łącze internetowe,
- etc.

Wszelkie anomalie w tym stanie jak: spadek szybkości pracy HD, całkowite zablokowanie czynności komputera, błędy w nagrywaniu i inne wynikają z niepoprawnego działania jednostki mocy.

Memory Test Script

```
while [ "$i" -le "$NR_PASSES" ]; do
  echo -n "Starting test pass #$i: "
  j=0
  echo -n "unpacking"
  while [ "$j" -lt "$NR_COPIES" ]; do
    if [ $PARALLEL = yes ]; then
      if [ $EXTRACT = yes ]; then
        mkdir -p memtest-work/$j
        $COMPRESS_PROG -dc $SOURCE_FILE 2>/dev/null | tar -xf - -C memtest-work/$j >/dev/null 2>&1 &
      else
        $COMPRESS_PROG -dc $SOURCE_FILE > memtest-work/$j 2>/dev/null &
      fi
    else
      if [ $EXTRACT = yes ]; then
        mkdir -p memtest-work/$j
        $COMPRESS_PROG -dc $SOURCE_FILE 2>/dev/null | tar -xf - -C memtest-work/$j >/dev/null 2>&1 &
        wait
        if [ $? -gt 128 ]; then
          exit 1
        fi
      else
        $COMPRESS_PROG -dc $SOURCE_FILE > memtest-work/$j 2>/dev/null &
        wait
        if [ $? -gt 128 ]; then
          exit 1
        fi
      fi
    fi
    j=$[ $j + 1 ]
  done
```

Cerberus charakterystyka

„IT IS NOT RECOMMENDED TO RUN THIS SOFTWARE ON A MACHINE USED FOR PRODUCTIVE PURPOSES, OR ON HARDWARE YOU DO NOT WANT TO DESTROY."

Rozpowszechniony pod licencją GNU GPL system do tworzenia i przeprowadzania programistycznych testów wydajności wszystkich sprzętowych komponentów komputera. Używany m.in. do tzw. „wypalania serwerów".

Cerberus - szeroki zakres testów

Procesor

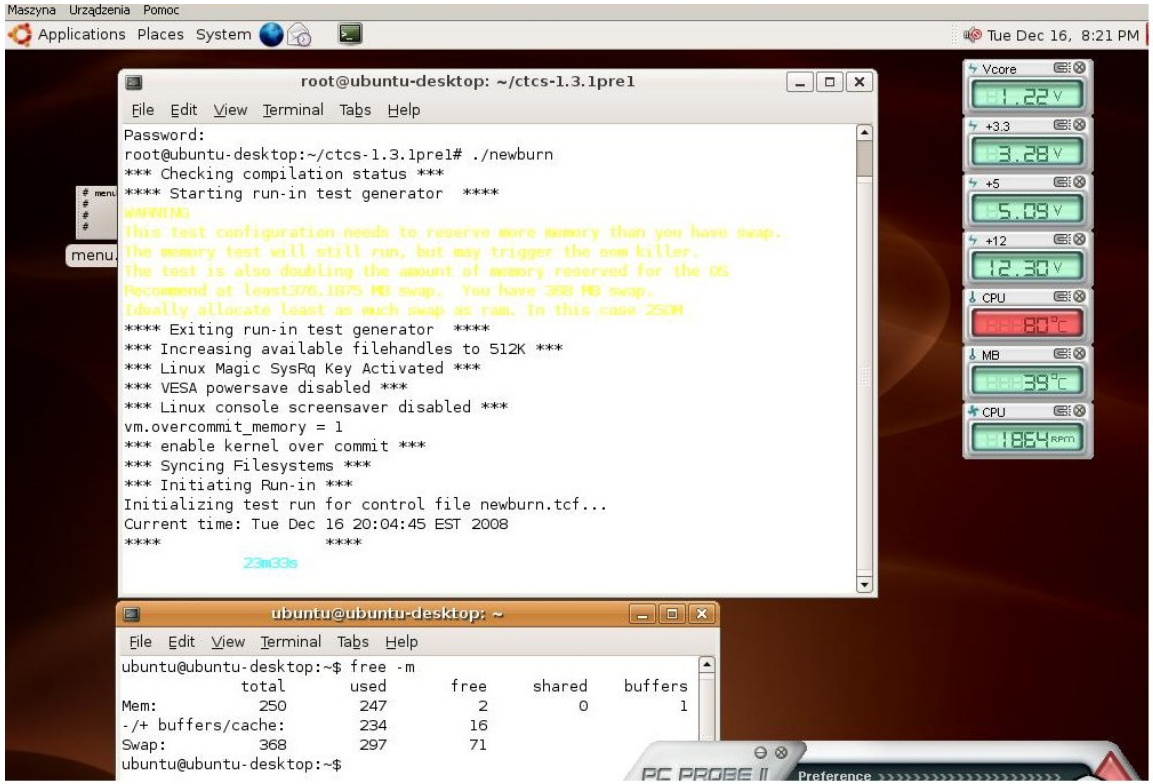
- wielokrotna kompilacja jądra systemu,
- skomplikowane operacje matematyczne - określanie czy liczba jest pierwsza..,
- inne operacje „rozpalające" procesor

Dyski twarde

- wielokrotne zapisywanie, usuwanie, przenoszenie dużych ilości danych na dysku,
- wielokrotne testy poprawności przetwarzanych danych,

Cerberus - skuteczny

Dopiero z jego pomocą można wykrywać pewne krytyczne błędy w systemach plików - wykazały „patologiczne" testy wykonywane dla jądra Linuxa.



Cerberus - zagrożenia

- Nie robi niczego niezwykłego.
- Podczas tworzenia uruchamiany wielokrotnie, bez efektów ubocznych.
- Powinien przejść bezbłędnie o ile system prawidłowo skonfigurowany i stabilny, a sprzęt dobrej jakości.

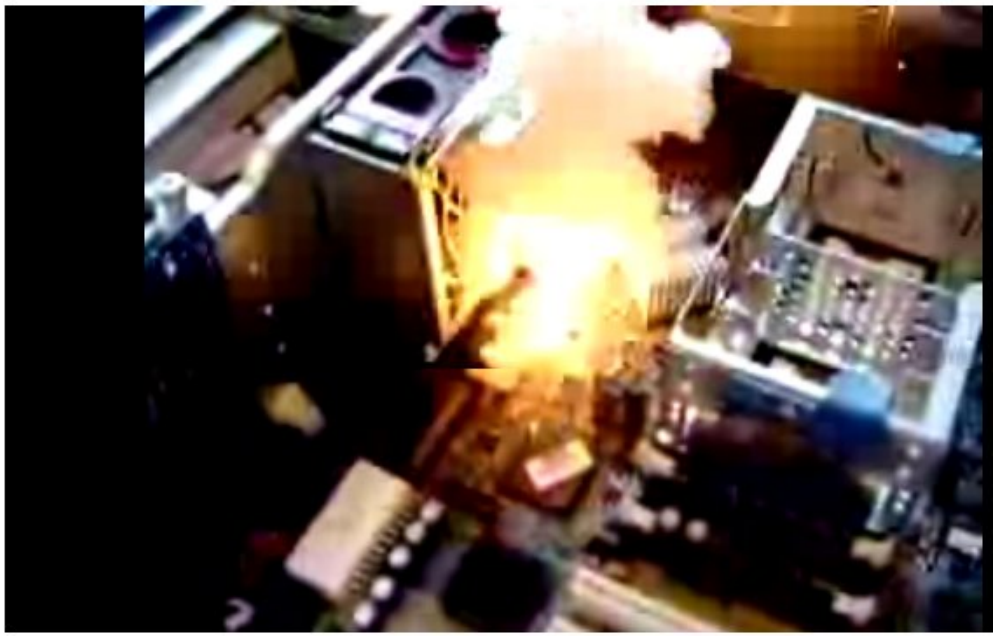
Jeśli system był niestabilny, a sprzęt za słaby lub miał ukryte defekty:

a) utrata danych na dyskach,
b) uszkodzenie oprogramowania,
c) ujawnienie, pogłębienie defektów sprzętowych,
d) poważne uszkodzenia sprzętu,
e) całkowita dewastacja „taniej" elektroniki,
f) (innej niż dostarczana przez VA Linux Systems)

Cerberus - optymistycznie

„If after a week the server is still running (not smoking) and hasn't crashed, it is considered good enough for use as a production machine. Web servers that have survived this process will certainly survive anything you can through at them."

Burning server



cpuburn-charakterystyka

Jest to ekstremalnie rygorystyczny, niskopoziomowy program przeznaczony do testowania zachowania procesora przy maksymalnym obciążeniu. Działa optymalnie na sprzęcie:

- Intel Pentium Pro/II/III, Celeron TM
- Itel P5
- AMD K7 (Athlon/Duron/Thunderbird TM)
- NMD K6

cpuburn-idea

- ⋆ Napisany tak by załadować maksymalnie dużo instrukcji do procesora.
- ⋆ Nie sprawdza wykonania każdej instrukcji.
- ⋆ Celem jest zmuszenie procesora do emisji maksymalnej ilości ciepła.
- ⋆ Co za tym idzie testowane jednocześnie chłodzenie i pobór mocy

Bootchart - charakterystyka

Narzędzie do wizualizacji i analizy procesu uruchamiania systemu Linux. Gromadzi wiele informacji na temat stanu komputera i systemu operacyjnego podczas fazy rozruchu i bezpośrednio po niej:

1. dane o inicjowanych procesach,
2. statystyki aktywności procesora
3. statystyki użycia dysków

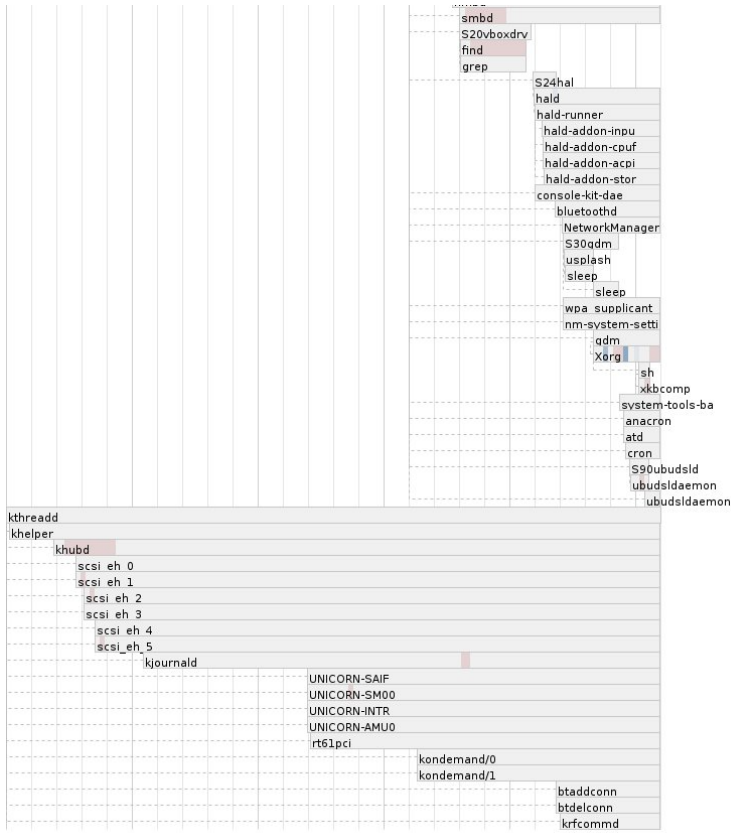
Zebrane przez skrypt inforamcje są zapisywane na dysku (log) i przetwarzane z użyciem aplikacji (Java) dla prezentacji w formie przystępnego wykresu (png/svg/eps).

Bootchart - dla optymalizacji

Projekt rozwinął się jako odpowiedź na postulaty Owena Taylora:

- Czas rozruchu systemu to około 2 minuty.
- Wymaga aktywności procesora, dysku twardego i czasu uaktywnienia zewnętrznych urządzeń (monitor, połączenie z siecią etc).
- Idealny system powinien wczytać około 100 MB danych (3-4 sek), głównie do tego angażując procesor.
- Żądania do zewnętrznych urządzeń powinny być asynchroniczne.
- W idealnym systemie użytkownik mógłby rozpocząć pracę po około 10 sek. od uruchomienia.
- Dobrze byłoby wiedzieć czym nasz system różni się od takiego idealnego.

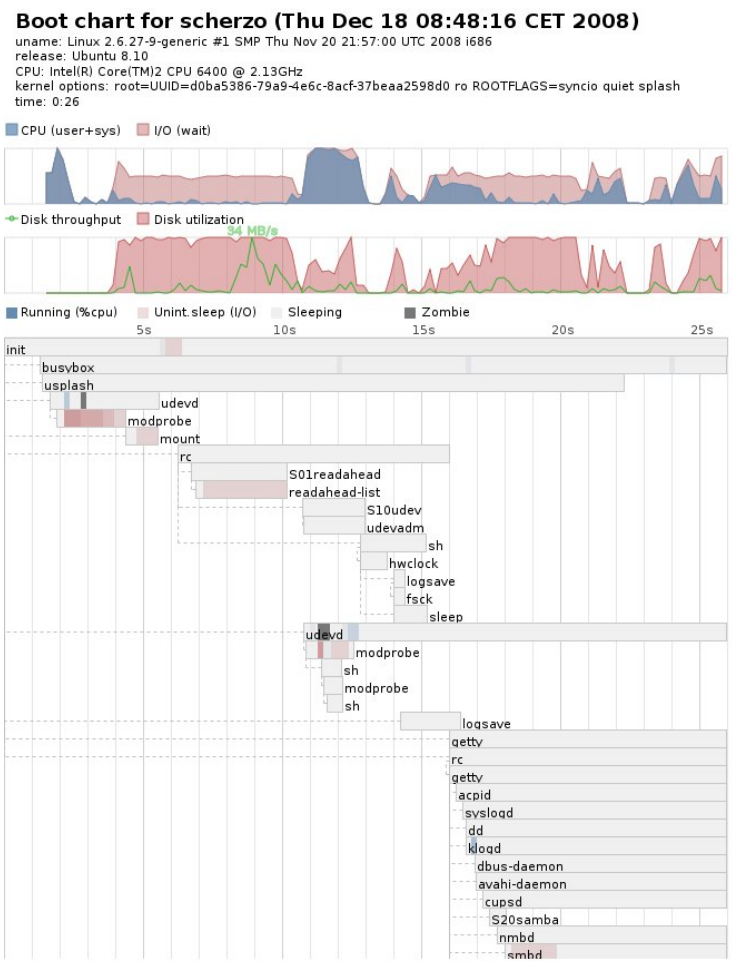
Bootchart - wykres



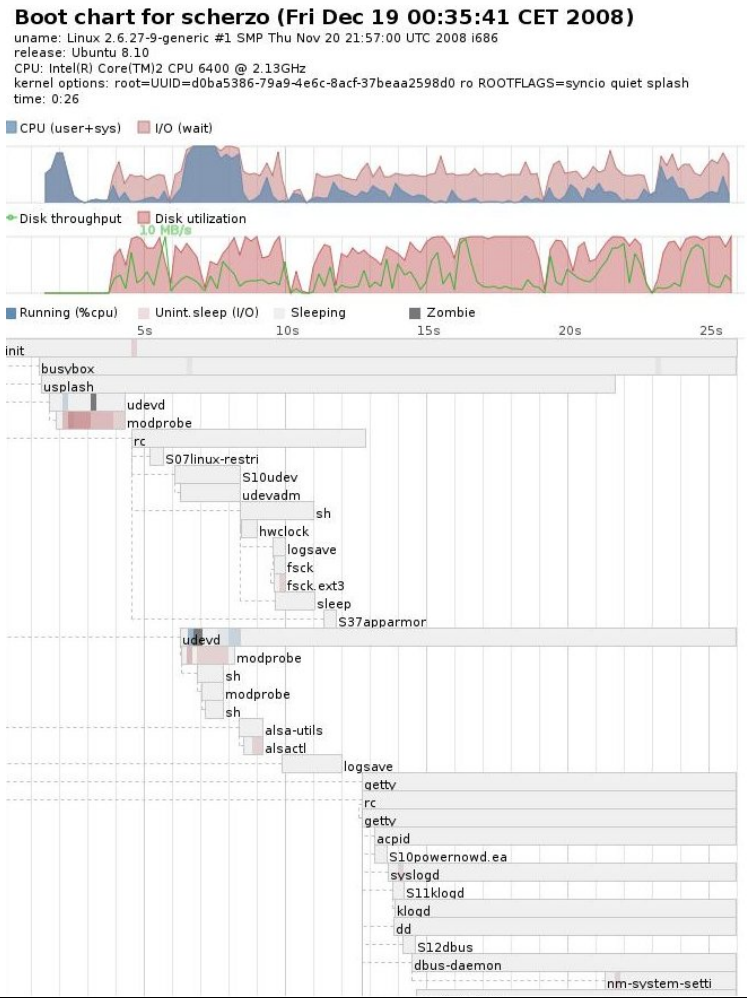
Mając wiedzę można eksperymentować - np z "readahead"

Bootchart - uruchamianie z metodą readahead

z readahead



bez



Linux Test Project

- Linux - rozwój, wiele architektur, ekspansja kodu

Linux Test Project

- Linux - rozwój, wiele architektur, ekspansja kodu
- Cel - stabilność, wydajność, robustness

Linux Test Project

- Linux - rozwój, wiele architektur, ekspansja kodu
- Cel - stabilność, wydajność, robustness
- Ktoś nad tym czuwa?: Linux Test Project (LTP)

Przestroga

Nie należy ignorować etapu testowania gdyż:

- Liczne błędy

Przestroga

Nie należy ignorować etapu testowania gdyż:

- Liczne błędy
- Konieczność częstych update'ów, bug-fiksów

Przestroga

Nie należy ignorować etapu testowania gdyż:

- Liczne błędy
- Konieczność częstych update'ów, bug-fiksów
- W efekcie zniechęcanie użytkowników

Początki

1. Zapoczątkowane przez SGI - 2001 rok
2. Założenia: udostępnić zestaw testów do walidacji stabilności wydajności i robustness jądra Linuksa
3. Udostępnienie narzędzi do automatyzacji przeprowadzania i tworzenia testów
4. Kernel 2.3 - 100 testów

Obecnie

1. Kernel 2.6 - ponad 3000 testów
2. Testowanie różnych funkcjonalności jądra: syscall, filesystem, ipc, timers, scheduler, modules ...
3. 95% kodu w C, reszta w Shellu i Perlu
4. Jedno z najważniejszych narzędzi stosowanych przez deweloperów, testerów i ludzi mających wpływ na rozwój Linuksa
5. począwszy od 2007 roku dynamiczny rozwój, rewizja dotychczasowych testów

Wynik w postaci HTML

PASSED	FAILED	WARNING	BROKEN	RETIRED	CONFIG-ERROR

Click Here for Detailed Report
Click Here for Summary Report

Detailed Report

No	Test-Name	Command-Line	Test-Output	Termination-id
1	abort01	ulimit -c 1024;abort01	abort01 1 PASS : Test passed	0
93	faccessat01	faccessat01	faccessat01 4 FAIL : faccessat01 failed, errno=20 : Not a directory	1
94	fallocate01	fallocate01	fallocate01 0 WARN : System doesn't support execution of the test	0
183	ftruncate04	ftruncate04	ftruncate04 1 CONF : The filesystem where /tmp is mounted does not support mandatory locks. Cannot run this test.	0

Summary Report

Test Summary	Run reported some Tests FAIL
LTP Version	LTP-20080331
Start Time	Mon Mar 31 12:14:29 CDT 2008
End Time	Mon Mar 31 13:05:21 CDT 2008
Log Result	/root/subrata/ltp/ltp-full-20080331/results
Output/Failed Result	/root/subrata/ltp/ltp-full-20080331/output
Total Tests	860
Total Failures	3
Kernel Version	2.6.21.3
Machine Architecture	i386
Hostname	sniff

Dodatkowo ...

- Współbieżne wykonywanie testów z kontrolą outputu

Dodatkowo ...

- Współbieżne wykonywanie testów z kontrolą outputu
- Zapuszczanie testów na określony czas lub/i określoną liczbę razy

Dodatkowo ...

- Współbieżne wykonywanie testów z kontrolą outputu
- Zapuszczanie testów na określony czas lub/i określoną liczbę razy
- Wysyłanie wyników pocztą elektroniczną

Dodatkowo ...

- Współbieżne wykonywanie testów z kontrolą outputu
- Zapuszczanie testów na określony czas lub/i określoną liczbę razy
- Wysyłanie wyników pocztą elektroniczną
- Generowanie obciążenia w trakcie testów, ograniczeń co do zużycia zasobów

Pokrycie kodu jądra

Directory	Coverage	
fs	49.8%	10135/20367 lines
include/asm	49.4%	595/1204 lines
include/linux	58.7%	2239/3812 lines
include/net	56.2%	990/1762 lines
ipc	52.8%	1442/2729 lines
kernel	38.2%	9880/25837 lines
lib	42.2%	2105/4992 lines
mm	51.5%	6899/13396 lines
net	65.4%	630/964 lines
security	51.9%	666/1283 lines

Table 2: 2.6.24 kernel code coverage using December 2006 LTP

Directory	Coverage	
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

Table 3: 2.6.24 kernel code coverage using March 2008 LTP

Własne testy

Schemat pliku z testem (napisanego w C)

- Nagłówki, własne makra

Własne testy

Schemat pliku z testem (napisanego w C)

- Nagłówki, własne makra
- main(), sprawdzenie czy test może zostać przeprowadzony

Własne testy

Schemat pliku z testem (napisanego w C)

- Nagłówki, własne makra
- main(), sprawdzenie czy test może zostać przeprowadzony
- setup() - alokowanie zasobów

Własne testy

Schemat pliku z testem (napisanego w C)

- Nagłówki, własne makra
- main(), sprawdzenie czy test może zostać przeprowadzony
- setup() - alokowanie zasobów
- wykonanie testu w pętli (ewentualne błędy)

Własne testy

Schemat pliku z testem (napisanego w C)

- Nagłówki, własne makra
- main(), sprawdzenie czy test może zostać przeprowadzony
- setup() - alokowanie zasobów
- wykonanie testu w pętli (ewentualne błędy)
- zaraportowanie wyniku - PASS/FAIL

Własne testy

Schemat pliku z testem (napisanego w C)

- Nagłówki, własne makra
- main(), sprawdzenie czy test może zostać przeprowadzony
- setup() - alokowanie zasobów
- wykonanie testu w pętli (ewentualne błędy)
- zaraportowanie wyniku - PASS/FAIL
- wypisanie danych do plików log/output

Własne testy

Schemat pliku z testem (napisanego w C)

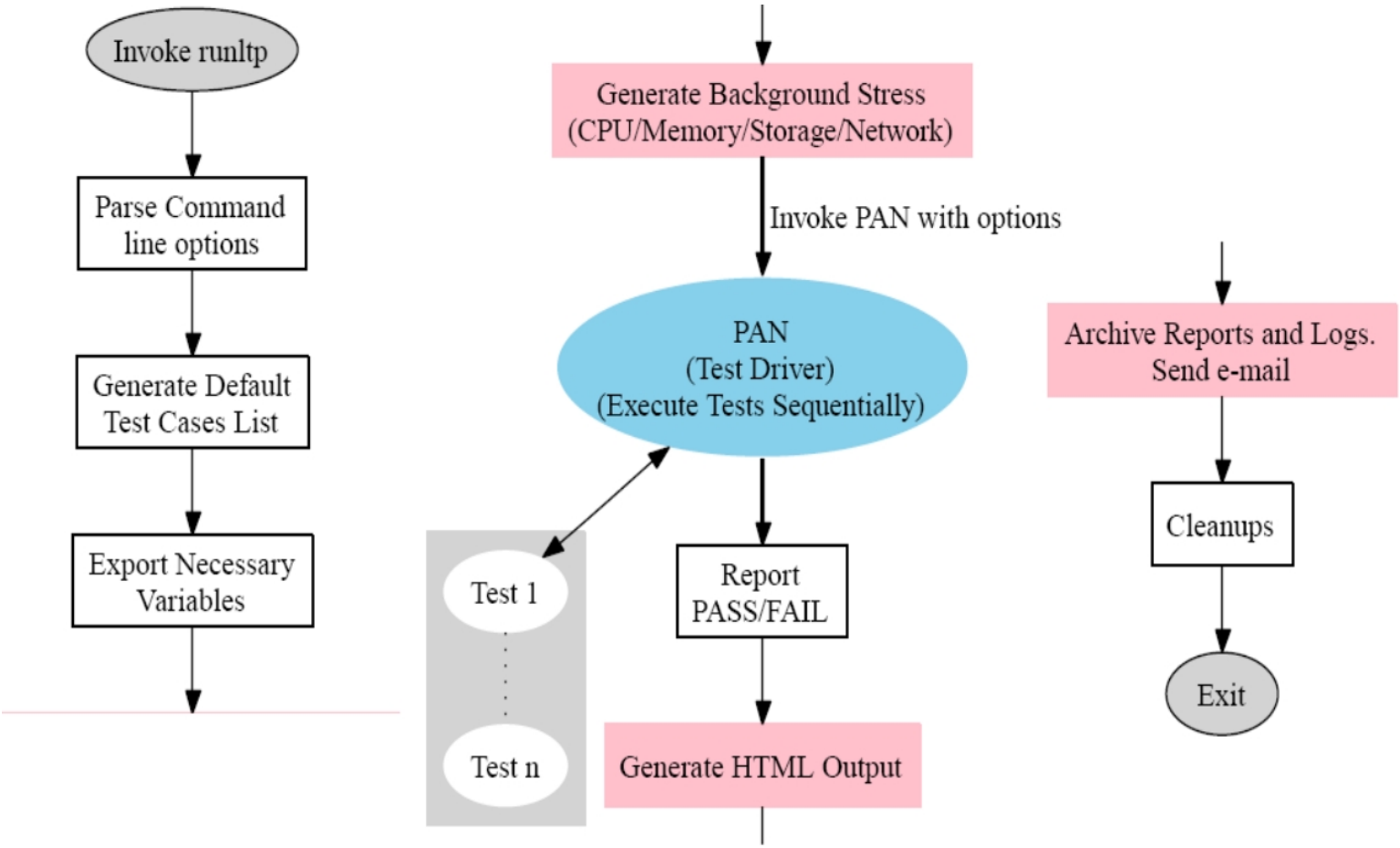
- Nagłówki, własne makra
- main(), sprawdzenie czy test może zostać przeprowadzony
- setup() - alokowanie zasobów
- wykonanie testu w pętli (ewentualne błędy)
- zaraportowanie wyniku - PASS/FAIL
- wypisanie danych do plików log/output
- cleanup() - zwolnienie zasobów

Własne testy

Schemat pliku z testem (napisanego w C)

- Nagłówki, własne makra
- main(), sprawdzenie czy test może zostać przeprowadzony
- setup() - alokowanie zasobów
- wykonanie testu w pętli (ewentualne błędy)
- zaraportowanie wyniku - PASS/FAIL
- wypisanie danych do plików log/output
- cleanup() - zwolnienie zasobów
- exit() - kod powrotu - PASS/FAIL

Runltp


Invoke runltp
Parse Command line options
Generate Default Test Cases List
Export Necessary Variables
Generate Background Stress (CPU/Memory/Storage/Network)
Invoke PAN with options
PAN (Test Driver) (Execute Tests Sequentially)
Test 1
Test n
Report PASS/FAIL
Generate HTML Output
Archive Reports and Logs. Send e-mail
Cleanups
Exit

co to jest?

Jest to proces

- znajdowania tych miejsc w kodzie, które nie były wykonane podczas przeprowadzania testów.

co to jest?

Jest to proces

- znajdowania tych miejsc w kodzie, które nie były wykonane podczas przeprowadzania testów.
- tworzenia nowych testów, które te miejsca odwiedzą.

co to jest?

Jest to proces

- znajdowania tych miejsc w kodzie, które nie były wykonane podczas przeprowadzania testów.
- tworzenia nowych testów, które te miejsca odwiedzą.
- określania odpowiedniej miary pokrycia, która w sposób pośredni jest miarą jakości.

co to jest?

Jest to proces

- znajdowania tych miejsc w kodzie, które nie były wykonane podczas przeprowadzania testów.
- tworzenia nowych testów, które te miejsca odwiedzą.
- określania odpowiedniej miary pokrycia, która w sposób pośredni jest miarą jakości.
- znajdowanie redundantnych testów, które nie zwiększają pokrycia kodu.

do czego to służy?

- Zapewnia jakość testów oprogramowania.

do czego to służy?

- Zapewnia jakość testów oprogramowania.

Metanarzędzie - testowanie testów.

do czego to służy?

- Zapewnia jakość testów oprogramowania.

Metanarzędzie - testowanie testów.

- Pośrednio zapewnia jakość samego oprogramowania.

co zakłada?

- Błędy dotyczą przepływu sterowania i poprzez zmianę tego przepływu możemy je odnaleźć. Np. ktoś napisał *if* (c), zamiast *if* $(!c)$.

co zakłada?

- Błędy dotyczą przepływu sterowania i poprzez zmianę tego przepływu możemy je odnaleźć. Np. ktoś napisał *if* (*c*), zamiast *if* (!*c*).
- Nie ma niedostępnych fragmentów kodu.

pokrycie instrukcji

- Jeśli piszemy intrukcje w osobnych liniach to to samo co:
  - pokrycie linii
  - pokrycie segmentu
- Nie wymaga przetwarzania kodu źródłowego.
- Ma swoje wady:

```
int* p = NULL;
if (condition)
p = &variable;
*p = 123;
```

pokrycie decyzji

- Pokazuje czy wyrażenia logiczne w strukturach takich, jak *if*, *while* wyliczyły się zarówno do *true* i *false*
- Udostępnia pokrycie przypadków struktury *switch* oraz exception handlerów.
- Inne nazwy to:
  - pokrycie gałęzi
  - pokrycie wszysktich krawędzi
- Ma swoje wady:

```
if (condition1 && (condition2 || function1()))
  statement1;
else
  statement2;
```

pokrycie warunków

- Sprawdza pokrycie wszystkich podwarunków w instrukcjach warunkowych.
- Bardziej wrażliwa na przepływ sterowania.
- Inne nazwy to:
  - pokrycie gałęzi
  - pokrycie wszysktich krawędzi
- Pełne pokrycie warunków nie implikuje pełnego pokrycia decyzji.
- Ma swoje wady:

```
bool f(bool e)  return false;
bool a[2] =  false, false ;
if (f(a && b)) ...
if (a[int(a && b)]) ...
if ((a && b) ? false : false) ...
```

wielokrotne pokrycie warunków

- Sprawdza pokrycie wszystkich kombinacji podwarunków logiczych (z dokładnością do leniwości).
- Nie obejmuje pokrycia decyzji.

```
a && b && (c || (d && e))
```

	a	b	c	d	e
1.	F				
2.	T	F			
3.	T	T	F	F	
4.	T	T	F	T	F
5.	T	T	F	T	T
6.	T	T	T		

pokrycie ścieżek

- Sprawdza czy każda możliwa ścieżka została wykonana.
- Trudności z oszacowaniem prawdziwej liczby ścieżek.

```
if (success)
  statement1;
statement2;
if (success)
  statement3;
```

- Pętle - nieskończona liczba ścieżek.

inne

- pokrycie funkji
- pokrycie wywołań
- pokrycie tablic
- race coverage

co to jest GCOV?

- Narzędzie w ramach projektu LTP.
- Współpracuje tylko z kompilatorem gcc.
- Pokrycie linii + procentowe pokrycie gałęzi.

jak on działa?

GCOV działa w czterech fazach:

- instrumentacja kodu

jak on działa?

GCOV działa w czterech fazach:

- instrumentacja kodu
- zbieranie danych w czasie wykonywania kodu

jak on działa?

GCOV działa w czterech fazach:

- instrumentacja kodu
- zbieranie danych w czasie wykonywania kodu
- ekstrakcja danych w czasie zakończenia wykonywania programu (a co gdy monitorujemy system?)

jak on działa?

GCOV działa w czterech fazach:

- instrumentacja kodu
- zbieranie danych w czasie wykonywania kodu
- ekstrakcja danych w czasie zakończenia wykonywania programu (a co gdy monitorujemy system?)
- analiza pokrycia kodu i prezentacja post-mortem

co to jest instrumentacja?

Kompilujemy program z opcją $-fprofile-args\ -ftest-coverage$
Do kodu dodawane są instrukcje, które zliczają pokrycie kodu:

- dla każdej funkcji tworzony jest graf przepływu sterowania,
- minimalizowana jest liczba krawędzi grafu, które muszą zostać zinstrumentalizowane (dopełnienie drzewa rozpinającego),
- cała informacja jest odtwarzana z tych krawędzi,
- każdy zinstrumantelizowany blok podstawowy programu dostaje swój numer ID,
- kompilator alokuje dla każdej funkcji tablicę zliczającą oraz dodaje do każdego takiego bloku instrukcje zwiększające liczniki
- tworzony są odpowiednie pliki .o

co to jest instrumentacja?

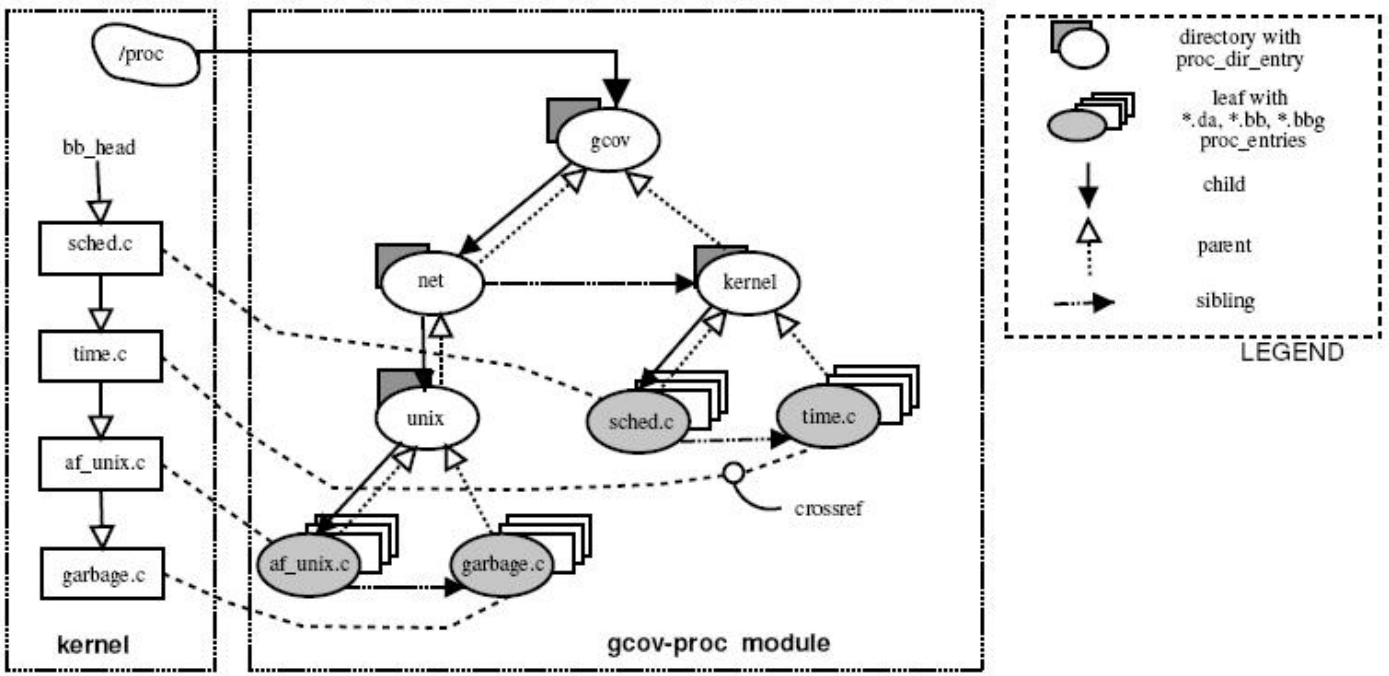
file1.c

```
→ static ulong counts[numbbs];
→ static struct bbobj =
→         { numbbs, &counts,"file1.c"};
→ static void _GLOBAL_.I.fooBarGCOV()
→         { __bb_init_func(&bbobj); }

void fooBar(void)
{
→   counts[i]++;
    <bb-i>
    if (condition) {
→      counts[j]++;
       <bb-j>
    } else {
       <bb-k>
    }
}

→ SECTION(".ctors"')
→ { &_GLOBAL_.I.fooBarGCOV }
```

a co w przypadku jądra?

Podstawowy problem: jądro nie kończy swojego działania.

Rozwiązanie - moduł, podsystem w stystemie proc + patch na jądro.



Inne problemy

Zasada nieoznaczoności Heisenberga!

Pomiar zmienia mierzoną wielkość.

- Kod jądra jest średnio pokryty w 14% przez działalność gcova.
- Pomiar nie jest atomowy w czasie - zebranie statystyk z całego jądra trwa długo.
- Problem z architekturami wieloprocesorowymi.

co to jest?

Nakładka na gcov umożliwiająca generowanie czytelnych dla ludzi plików html.

Bibligrafia

http://ltp.sourceforge.net/
http://linuxquality.sunsite.dk/articles/testsuites/
http://www.linuxbase.org/test/
http://kernel-perf.sourceforge.net/about$_t$*ests.php*
http://www.bootchart.org/
http://www.kernelbench.org/
http://www.ibm.com/developerworks/linux/library/l-stress/index.html
http://www.anchor.com.au/hosting/dedicated/Server-burnin-hardware-testing"
http://linux.softpedia.com/get/System/Diagnostics/cpuburn-1407.shtml