Podręcznik testera jądra Linux

(wersja 0.2)

Michał Piotrowski
michal.k.k.piotrowski@gmail.com
Linux Testers Group

Współautorzy:
Maciej Rutecki
maciej.rutecki@gmail.com
Unixy.pl
Rafał J. Wysocki
rjw@sisk.pl

Ludzie, którzy wpłynęli
na kształt podręcznika:

Bartłomiej Żołnierkiewicz
bzolnier@gmail.com
Jarek Popławski
jarkao2@o2.pl
Mariusz Kozłowski
m.kozlowski@tuxland.pl

Spis treści

Wprowadzenie

Testowanie oprogramowania odgrywa bardzo ważną rolę w procesie jego tworzenia. Niniejszy podręcznik ma za zadanie przybliżyć Czytelnikowi proces testowania jądra systemu Linux. Jest podzielony na kilka krótkich rozdziałów omawiających ważniejsze zagadnienia, które powinny być znane każdemu dobremu testerowi. Nie opisuję w nim dogłębnie wszystkich problemów na jakie można natrafić podczas testowania aby nie zanudzać Czytelnika, oraz aby zostawić pole do własnych odkryć.

Dlaczego warto się zająć testowaniem jądra Linux?

> „Otóż moim zdaniem powinniśmy je testować, jeżeli chcemy mieć gwarancję, że kolejne wersje jądra będą prawidłowo działać na naszym sprzęcie i będą prawidłowo robić to, co jest nam potrzebne. Innymi słowy, jeżeli chcemy używać Linuksa „na poważnie", to warto poświęcić trochę czasu na sprawdzanie, czy w kolejnej wersji nie „szykuje się" coś, co nam popsuje szyki, zwłaszcza, że obecnie jądro dość znacznie zmienia się między „stabilnymi" wersjami (z czego większość użytkowników nawet nie zdaje sobie sprawy). W gruncie rzeczy na tym polega odpowiedzialność użytkownika Open Source: należy sprawdzać nowe wersje i informować o problemach. Jeżeli tego nie robimy, to później nie mamy prawa narzekać, że coś przestało działać.
>
> Oczywiście z naszego testowania skorzystają także inni użytkownicy, ale raczej nie warto liczyć na to, że ktoś za nas sprawdzi czy na naszym sprzęcie nowa wersja jądra będzie działać prawidłowo, czy nie. :-)"
> – Rafał J. Wysocki

Niestety wśród użytkowników Linuksa pokutuje mit mówiący, że ludzie chcący testować jądro systemu muszą umieć programować. Takie stwierdzenie jest równie prawdziwe jak stwierdzenie, że ludzie oblatujący nowe modele samolotów muszą umieć je projektować. Umiejętność programowania jest bardzo przydatna podczas wykonywania testów, pozwala lepiej ocenić sytuację, oraz pozwala poznać sposób działania jądra systemu, jednak brak tej umiejętności nie dyskwalifikuje nas jako testerów.

Powinniśmy posiadać wiedzę o budowie i zasadzie działania systemu operacyjnego. Polecam lekturę „*Budowa systemu operacyjnego Linux 2.0*" `http://rainbow.mimuw.edu.pl/SO/Linux/` (opracowanie jest przestarzałe, jednak jako wprowadzenie nadaje się bardzo dobrze), „*Linux Device Drivers*" Jonathana Corbeta, Alessandro Rubiniego i Grega Kroah-Hartmana `http://lwn.net/Kernel/LDD3/` i „*Linux Kernel Development*" Roberta Love `http://rlove.org/kernel_book/` (na stronie KernelNewbies można znaleźć listę wszystkich

ciekawych książek `http://kernelnewbies.org/KernelBooks`). Bardzo przydatne artykuły przybliżające działanie niektórych komponentów jądra Linux można znaleźć na stronach Linux Weekly News `http://lwn.net/Kernel/Index/` (ang.). Powinniśmy jednak pamiętać, że żadna nawet najlepsza książka czy artykuł nie zastąpią nam lektury kodu źródłowego jądra systemu jedynej zawsze aktualnej dokumentacji.

Kolejny często spotykany mit mówi: „używając rozwojowych wersji systemu narażamy się na utratę danych". Jest to jak najbardziej prawdziwe stwierdzenie, tak samo jak „używając noża kuchennego narażamy się na utratę palców". Aby zaradzić utracie danych, najlepiej jest wykonywać testy na systemie przeznaczonym do tego celu, zainstalowanym na oddzielnej partycji lub dysku, na którym nie montujemy żadnych partycji z systemu stabilnego. Dobrą praktyką jest też wykonywanie regularnych kopii zapasowych naszych danych (to jest bardzo dobra praktyka również na systemach uważanych za stabilne).

Część I

Podstawy

Rozdział 1

Jądro, łatki, drzewa i kompilacja

1.1 Jądro

Aktualną wersję jądra systemu możemy pobrać z „The Linux Kernel Archives" `http://www.kernel.org/` w postaci dużego archiwum `tar` skompresowanego za pomocą programu `gzip` lub `bzip2` (warto pamiętać o tym, że pliki `bz2` są mniejsze niż `gz`). Archiwum należy rozpakować do jakiegoś katalogu (standardowo `/usr/src/`), można to zrobić na kilka sposobów (urok systemów uniksowych), moim ulubionym jest:
`$ tar xjvf linux-2.6.x.y.tar.bz2` - pliki skompresowane programem bzip2
`$ tar xzvf linux-2.6.x.y.tar.gz` - pliki skompresowane programem gzip
Równie dobrze możemy postąpić zgodnie z instrukcją zawartą w pliku `README`

```
$ bzip2 -dc linux-2.6.x.y.tar.bz2 | tar xvf -
$ gzip -cd linux-2.6.x.y.tar.gz | tar xvf -
```

Po rozpakowaniu kodu źródłowego jądra otrzymujemy katalog o nazwie odpowiadającej jego wersji (np. `linux-2.6.18`), którego zawartość często jest nazywana „drzewem jądra" (ang. kernel tree). Z tego powodu słowo „drzewo" bywa wykorzystywane do nazywania różnych wersji kodu źródłowego jądra. Na przykład zamiast mówić „wersja źródeł jądra przygotowywana przez Andrew Mortona" wygodniej jest powiedzieć „drzewo -mm".

1.2 Łatki

Aby zaktualizować źródła nie powinniśmy pobierać całego archiwum, szybciej jest pobrać łatkę (*ang. patch*). Aplikujemy ją będąc wewnątrz katalogu z źródłami systemu:

```
$ bzip2 -cd /ścieżka/do/patch-2.6.x.bz2 | patch -p1
$ gzip -cd /ścieżka/do/patch-2.6.x.gz | patch -p1
```

lub

```
$ patch -p1 < /ścieżka/do/patch-x.y.z
$ cat /ścieżka/do/patch-x.y.z | patch -p1
```

w przypadku łatek rozpakowanych. (To oczywiście są tylko najprostsze sposoby, bez problemu można wymyślić ciekawsze.)

Bardzo użyteczną funkcją jest odwracanie łatek gdy ich już nie potrzebujemy (lub źle działają), możemy to wykonać dodając do flag `patch` parametr `-R`.

```
$ bzip2 -cd /ścieżka/do/patch-2.6.x.bz2 | patch -p1 -R
```
Możemy też sprawdzić czy łatka aplikuje się czysto (nie ma żadnych odrzuconych fragmentów kodu) przy użyciu flagi `dry-run`
```
$ bzip2 -cd /ścieżka/do/patch-2.6.x.bz2 | patch -p1 --dry-run
```
Jeżeli nie zauważyliśmy żadnych komunikatów
„`1 out of 1 hunk FAILED -- saving rejects to file *`”,
możemy spokojnie nałożyć łatkę.

Rzecz jasna po nałożeniu łatek otrzymujemy drzewo jądra różne od tego, które mieliśmy na początku. W związku z tym drzewo, od którego zaczynamy nakładanie łatek, często nazywa się drzewem surowym (*ang. raw*), lub zwyczajnym (*ang. vanilla*). Większość różnych wersji kodu źródłowego jądra, czyli drzew jądra, jest dostępna w postaci zbiorów łatek, lub nawet w postaci pojedynczych łatek, które należy nałożyć na odpowiednią „surową” wersję jądra dostępną z `ftp://ftp.kernel.org`. Stabilne wersje jądra mają oznaczenie `2.6.x` lub `2.6.x.y`, wersje rozwojowe są oznaczane jako `2.6.x-git*`, `2.6.x-rc*` oraz `2.6.x-rc*-git*`. Ten sposób oznaczania pozwala na łatwe zorientowanie się, jakie łatki należy nałożyć na aktualnie posiadaną wersję, aby uzyskać tą która nas interesuje.

Poniższy przykład rozjaśni całą sytuację. Mamy do dyspozycji stabilną wersję `2.6.16.25` i chcemy przejść do wersji rozwojowej `2.6.20-rc1-git1`. Rozpakowujemy naszą kopię źródeł:
```
$ tar xjvf linux-2.6.16.25.tar.bz2
```
Pobieramy łatkę patch-2.6.16.25.bz2 i odwracamy ją.
```
$ bzip2 -cd /ścieżka/do/patch-2.6.16.25.bz2 | patch -p1 -R
```
Teraz w katalogu roboczym mamy wersję `2.6.16` - o to nam chodziło, łatki stabilne i rozwojowe aplikują się czysto tylko na wersję `2.6.x`, ponieważ już zawierają poprawki wchodzące w skład `2.6.x.y`. Teraz pobieramy łatki w wersjach `2.6.17`, `2.6.18` oraz `2.6.19` i aplikujemy je po kolei.
```
$ bzip2 -cd /ścieżka/do/patch-2.6.17.bz2 | patch -p1
$ bzip2 -cd /ścieżka/do/patch-2.6.18.bz2 | patch -p1
$ bzip2 -cd /ścieżka/do/patch-2.6.19.bz2 | patch -p1
```
Otrzymaliśmy wersję systemu `2.6.19`.
Następnie trzeba pobrać dwie łatki rozwojowe `2.6.20-rc1` i `2.6.20-rc1-git1`, aplikujemy je w podanej kolejności. Prawda że proste? Ale żmudne - dlatego ludzie, którzy nie lubią marnować czasu, stosują narzędzie nazywające się `ketchup` `http://www.selenic.com/ketchup/`.
Więcej o aplikowaniu i ściąganiu łatek można przeczytać w plikach:
`Documentation/applying-patches.txt` oraz `README` znajdujących się w źródłach jądra.

Jeśli podczas nakładania łatki zobaczymy komunikat „`patch: **** malformed patch at line`” np.

```
$ cat ../sched-2.patch | patch -p1 -R --dry-run
patching file kernel/sched.c
patch: **** malformed patch at line 33:         if (next == rq->idle)
```

oznacza to, że łatka najprawdopodobniej została prze formatowana przez naszego klienta poczty - lub też kopiowaliśmy łatkę bezpośrednio ze strony archiwum LKML - wtedy najczęściej wszystkie wcięcia używane w kodzie źródłowym znikają. Na stronie `http:`

//marc.theaimsgroup.com/?l=linux-kernel wszystkie wiadomości są dostępne w postaci tekstowej (bez formatowania HTML) - wystarczy kliknąć na link „Download message RAW".

1.3 Ketchup

Ketchup jest narzędziem, dzięki któremu możemy w prosty i szybki sposób pobrać najnowszą wersję interesującego nas jądra systemu. Robi to w całkiem inteligentny sposób, sprawdza aktualną wersję i pobiera tylko wymagane łatki. Jeżeli używamy Debiana testing/unstable, to po prostu instalujemy pakiet ketchup przy pomocy naszego ulubionego narzędzia np.
`# apt-get install ketchup`
dostępny, to najłatwiej będzie wykonać:
`$ mkdir ~/bin` (jeżeli nie mamy bin w katalogu domowym)
`$ cd ~/bin`
`$ wget http://www.selenic.com/ketchup/ketchup-0.9.8.tar.bz2`
`$ tar xjvf ketchup-0.9.8.tar.bz2`
Musimy także mieć zainstalowany pakiet `python`.
Należy pamiętać o tym, że ketchup potrafi sprawdzać sygnatury łatek, a więc odwiedzamy stronę `http://www.kernel.org/signature.html` zapisujemy aktualny klucz do jakiegoś pliku tekstowego
(`$ gpg --keyserver wwwkeys.pgp.net --recv-keys 0x517D0F0E` jakoś nigdy u mnie nie działało) i robimy:
`$ gpg --import plik_z_kluczem_kernel.org`
Teraz możemy sprawdzić jak to wszystko działa:
`$ mkdir ~/tree`
`$ cd ~/tree`
`$ ketchup -m`
Powinien pojawić się taki błąd:

```
Traceback (most recent call last):
  File "/usr/bin/ketchup", line 695, in ?
    lprint(get_ver("Makefile"))
   File "/usr/bin/ketchup", line 160, in get_ver
     m = open(makefile)
 IOError: [Errno 2] No such file or directory: 'Makefile'
```

Oznacza to tylko i wyłącznie tyle, że nie mamy w tym katalogu żadnego drzewa Linuksa. Poleceniem „`ketchup 2.6-tip`" pobieramy najnowszą wersję drzewa „stable":
`$ ketchup 2.6-tip`

```
None -> 2.6.16.1
Unpacking linux-2.6.15.tar.bz2
Applying patch-2.6.16.bz2
Downloading patch-2.6.16.1.bz2
--21:11:47--  http://www.kernel.org/pub/linux/kernel/v2.6/patch-2.6.16.1.bz2
           => `/home/michal/.ketchup/patch-2.6.16.1.bz2.partial'
```

```
Translacja www.kernel.org... 204.152.191.5, 204.152.191.37
Connecting to www.kernel.org|204.152.191.5|:80... połączono.
Żądanie HTTP wysłano, oczekiwanie na odpowiedź... 200 OK
Długość: 5,284 (5.2K) [application/x-bzip2]

100%[====================================>] 5,284          6.19K/s

21:11:49 (6.18 KB/s) - '/home/michal/.ketchup/patch-2.6.16.1.bz2.partial'
saved [5284/5284]

Downloading patch-2.6.16.1.bz2.sign
--21:11:49--  http://www.kernel.org/pub/linux/kernel/v2.6/
patch-2.6.16.1.bz2.sign
           => '/home/michal/.ketchup/patch-2.6.16.1.bz2.sign.partial'
Translacja www.kernel.org... 204.152.191.37, 204.152.191.5
Connecting to www.kernel.org|204.152.191.37|:80... połączono.
Żądanie HTTP wysłano, oczekiwanie na odpowiedź... 200 OK
Długość: 250 [application/pgp-signature]

100%[====================================>] 250           --.--K/s

21:11:49 (18.34 MB/s) - '/home/michal/.ketchup/
patch-2.6.16.1.bz2.sign.partial' saved [250/250]

Verifying signature...
gpg: Signature made wto 28 mar 2006 09:37:31 CEST using DSA key ID 517D0F0E
gpg: Good signature from "Linux Kernel Archives Verification Key
 <ftpadmin@kernel.org>"
gpg: OSTRZEŻENIE: Ten klucz nie jest poświadczony zaufanym podpisem!
gpg:           Nie ma pewności co do tożsamości osoby która złożyła podpis.
Odcisk klucza głównego: C75D C40A 11D7 AF88 9981  ED5B C86B A06A 517D 0F0E
Applying patch-2.6.16.1.bz2
```

Teraz wydajemy polecenie „`ketchup -m`" aby sprawdzić, czy rzeczywiście mamy najnowsze stable:

```
$ ketchup -m
2.6.16.1
```

Jak widać, ketchup w razie potrzeby pobiera potrzebne łatki, inne bierze z folderu `~/.ketchup`, znajdującego się w naszym katalogu domowym - więc dobrym pomysłem będzie wrzucenie tam łatek, które aktualnie już posiadamy. Przejście do konkretnej wersji drzewa nie stanowi dla `ketchup`'a żadnego problemu. Po prostu wpisujemy „`ketchup wersja_drzewa`" np.

```
$ ketchup 2.6.16-rc4-mm1

2.6.16.1 -> 2.6.16-rc4-mm1
Applying patch-2.6.16.1.bz2 -R
Applying patch-2.6.16.bz2 -R
```

```
Applying patch-2.6.16-rc4.bz2
Applying 2.6.16-rc4-mm1.bz2
```

lub, gdy chcemy najnowsze drzewo -rt
`$ ketchup 2.6-rt`

```
ketchup 2.6-rt
2.6.16-rc4-mm1 -> 2.6.16-rt12
Applying 2.6.16-rc4-mm1.bz2 -R
Applying patch-2.6.16-rc4.bz2 -R
Applying patch-2.6.16.bz2
Applying patch-2.6.16-rt12
```

Narzędzie automatycznie nakłada/wyrzuca łatki.
Aby pobrać archiwum z łatkami wchodzącymi w skład ostatniego -mm, powinniśmy wykonać
`$ wget -c ‘ketchup -u 2.6-mm | sed "s/.bz2/-broken-out.tar.bz2/"‘`
Wybrane drzewa:

- 2.6-tip - aktualne stabilne wydanie Linuksa
- 2.6-rc - najnowsza wersja rozwojowa -rc
- 2.6-git - najnowsza wersja rozwojowa -git
- 2.6-rt - drzewo Ingo Molnara
- 2.6-mm - drzewo Andrew Mortona
- 2.6-ck - drzewo Cona Kolivasa (desktop)
- 2.6-cks - drzewo Cona Kolivasa (serwer)

Po co w takim razie wypisywałem te wszystkie polecenia służące do nakładania/odwracania łatek? Powód jest bardzo prosty - jak znajdziemy i zgłosimy błąd, to dostaniemy do przetestowania poprawkę, którą powinniśmy umieć poprawnie nałożyć.

1.4 Drzewa

Często spotkamy się z pojęciem „drzewo jądra” (*ang. kernel tree*). Co to takiego? Drzewo, jest to zbiór łatek posiadający swoją własną nazwę np. drzewo -mm, -rt dający się nałożyć na konkretną wersję systemu. Do popularniejszych drzew można zaliczyć:

- Drzewo -mm (prowadzone przez Andrew Mortona) jest zbiorem innych mniejszych drzew i eksperymentalnych łatek.
- Drzewo -rt (prowadzone przez Ingo Molnara) - znajdują się w nim łatki zamieniające Linuksa w system ograniczony czasowo (Real Time).
- Drzewo -ck (prowadzone przez Cona Kolivasa) - skupia się na poprawianiu wydajności systemu.

Praktycznie każdy deweloper, oraz podsystem jądra posiada swoje drzewo nad którym prowadzone są prace. Po pewnym czasie różne części drzew wchodzą do drzewa „surowego", prowadzonego przez Linusa Torvaldsa.

1.5 Drzewo -mm

Jak już wcześniej wspomniałem, drzewo -mm „jest zbiorem innych mniejszych drzew i eksperymentalnych łatek", to właśnie w nim przebiega główna linia frontu walki z błędami w Linuksie. Zrozumienie znaczenia tego drzewa ułatwi schemat przepływu łatek, a jest on następujący (przynajmniej tak to wygląda w teorii) drzewo -mm - drzewa podsystemów - drzewo Linusa.

Nowe łatki wysyłane przez deweloperów, albo są włączane od razu do drzewa danego podsystemu, albo najpierw trafiają do -mm (jeżeli trafią od razu do drzewa podsystemu, to i tak zostaną przetestowane w -mm, ponieważ właśnie z tych drzew się ono składa). Andrew Morton łączy wszystkie drzewa w jedną spójną całość, którą następnie wypuszcza do przetestowania.

Drzewo -mm jest zawsze dostępne w postaci pojedynczej łatki i zbioru wszystkich, z których zostało złożone (razem z poprawkami, które wprowadził Andrew). Ułatwia to odnajdywanie łatek wprowadzających błędy (więcej informacji o `quilt` i przeszukiwaniu binarnym można znaleźć w rozdziale czwartym). W katalogu, w którym umieszczone są łatki (na `kernel.org`) znajdziemy podkatalog hot-fixes - przed przystąpieniem do kompilacji warto zawsze do niego zajrzeć i pobrać poprawki, które się w nim znajdują.

Łatki włączane i wyrzucane z drzewa -mm są wysyłane na listę mm-commits (`http://vger.kernel.org/vger-lists.html#mm-commits`) - jej śledzenie pozwoli nam na zorientowanie się w aktualnej jego zawartości, oraz ułatwi odnajdywanie łatek, które wprowadziły nowe błędy.

Od czasu do czasu Andrew wysyła na mm-commits list zatytułowany „mm snapshot broken-out-data-godzina.tar.gz uploaded" jest to najnowsza migawka drzewa -mm przeznaczona głównie dla ludzi, którzy chcą pomóc w jego „poskładaniu".

1.6 Kompilacja, przygotowanie systemu testowego

Powinniśmy zacząć od przygotowania środowiska do pracy, potrzebne nam będą: `gcc`, `make`, `binutils`, `util-linux`, pakiet z plikami nagłówkowymi `ncurses` (`libncurses5-dev` w Debianie lub `ncurses-devel` w Fedorze). Plik `Documentation/Changes` w katalogu ze źródłami systemu zawiera spis wszystkich niekoniecznie wymaganych programów.

Poniższe polecenia należy wykonać z uprawnieniami administratora.

```
# groupadd kernel
# mkdir /usr/src/kernel
# chgrp kernel /usr/src/kernel
# chmod 775 /usr/src/kernel/
# usermod -G kernel <login>
```

Pierwsze polecenie tworzy nową grupę nazywająca się „kernel", następnie tworzymy katalog roboczy, zmieniamy uprawnienia dostępu do katalogu tak, aby użytkownicy należący do grupy „kernel" mieli w nim prawo do zapisu. Ostatnie polecenie dodaje nasze konto do świeżo utworzonej grupy. Jeżeli używamy systemu opartego na Debianie, to polecenie „`usermod -G src <login>`" doda nasze konto do grupy `src`, która ma już prawo do zapisu w `/usr/src`.
Warto zwrócić uwagę na to, że po pewnym czasie nasz katalog `/usr/src/kernel` może zajmować kilka gigabajtów, więc może powinniśmy go umieścić w jakimś innym miejscu.
Aby skompilować jądro musimy wykonać sekwencję poleceń:

- `make oldconfig` - przetwarza nasz stary plik konfiguracyjny (nazywający się po prostu `.config` (na początku nazwy jest kropka - plik jest plikiem ukrytym)) i wyświetla zapytania czy chcemy użyć nowych opcji, sterowników etc.

- `make menuconfig` - pozwala zmienić konfigurację - instrukcja krok po kroku jak przeprowadzić konfigurację jądra znajduje się pod adresem: `http://kompilacja_jadra_linuxa_26.xt.pl/`

- `make` - kompilujemy jądro systemu

- `make modules_install` - instalujemy wybrane moduły w `/lib/modules/wersja`

- `cp arch/i386/boot/bzImage /boot/vmlinuz-2.6.x.y` - kopiujemy jądro do katalogu `/boot`

- `cp System.map /boot/System.map-2.6.x.y` - kopiujemy mapę symboli do katalogu `/boot`

Na początku, gdy nie wiemy jeszcze co włączyć a co wyłączyć w pliku konfiguracyjnym naszego jądra, można skorzystać z konfiguracji jądra dystrybucyjnego (przeważnie zawiera mnóstwo nie potrzebnych nam rzeczy, dlatego też warto się nauczyć konfigurować jądro ;)).

```
$ zcat /proc/config.gz > .config
$ make oldconfig
```

Kilka przydatnych opcji:

- `make O=/katalog/` - zapisuje wynik kompilacji w innym katalogu. Opcja bardzo przydatna, szczególnie, gdy nie chcemy sobie zaśmiecać katalogu z jądrem plikami `*.o`. (Trzeba wtedy też używać zmodyfikowanych wersji „`make O=/katalog menuconfig`", „`make O=/katalog oldconfig`", „`make O=/katalog modules_install`" itd.).

- `make CC=` - możemy podać nazwę naszego kompilatora gcc, np. w debianie mamy do dyspozycji kilka wersji `gcc`, gdy chcemy kompilować jądro wersją 3.4 wystarczy wpisać „`make CC=gcc-3.4`".

- `make C=1` - przed kompilacją źródła są sprawdzane przez `sparse`.

- `make -j<liczba>` - możemy podać, ile jednocześnie procesów `gcc` chcemy uruchomić. Na maszynach SMP przyśpiesza to proces kompilacji.

- `make xconfig` - konfiguracja w trybie graficznym (qt).
- `make gconfig` - konfiguracja w trybie graficznym (gtk).

Po wybraniu opcji:

```
Loadable module support  --->
[*] Enable loadable module support
[*]   Automatic kernel module loading
```

będziemy mogli korzystać z tak zwanych modułów. Są to fragmenty kodu, które nie są wbudowywane w jądro na stałe. Dzięki modularyzacji jądro może być znacznie mniejsze, przez co może działać szybciej. Aby skompilować wybrany fragment kodu jako moduł musimy oznaczyć go literą „M" podczas konfiguracji np.
`<M>   Check for non-fatal errors on AMD Athlon/Duron / Intel Pentium 4`

Jeśli wcześniej wybraliśmy opcję „`Automatic kernel module loading`", to moduły powinny być ładowane przez jądro automatycznie wtedy, gdy będą potrzebne - czyli, jeżeli chcemy zamontować płytę cd-rom, to zostanie załadowany moduł isofs.ko. Do ręcznego ładowania i wyładowywania modułów można użyć polecenia `modprobe` (lub `insmod`, `rmmod`)

- `modprobe isofs` - załaduje moduł `isofs.ko`
- `modprobe -r isofs` - usunie moduł `isofs.ko`

Wszystkie moduły jakie są aktualnie dostępne dla naszego systemu możemy znaleźć w podkatalogach `/lib/modules/<wersja_jądra>/kernel`.
Podczas konfiguracji jądra warto zwrócić uwagę na opcje znajdujące się w menu
`Kernel hacking`
Powinniśmy włączyć przynajmniej:

```
[*] Kernel debugging
[*]   Compile the kernel with debug info
```

Poniższe opcje służą do debugowania konkretnej funkcjonalności jądra, w miarę możliwości powinniśmy włączyć je wszystkie (niektóre mogą znacznie obniżyć szybkość działania systemu)

```
[*]   Debug shared IRQ handlers
[*]   Detect Soft Lockups
[*]   Debug slab memory allocations
[*]     Slab memory leak debugging
[*]   RT Mutex debugging, deadlock detection
[*]   Built-in scriptable tester for rt-mutexes
[*]   Lock debugging: prove locking correctness
[*]   Lock dependency engine debugging
[*]   Locking API boot-time self-tests
[*]   Highmem debugging
[*]   Debug VM
```

Bardzo użyteczną funkcją jest

```
[*] Magic SysRq key
```

Często pozwala ograniczyć skutki błędu jądra, oraz może służyć do uzyskania dodatkowych informacji o stanie systemu. Oferuje nam skróty klawiaturowe, które pozwalają wykonać ściśle określone instrukcje:

- Alt + SysRq + h - wyświetla pomoc
- Alt + SysRq + b - natychmiastowy restart systemu, bez odmontowania dysków i zapisania buforów. NIEBEZPIECZNE!
- Alt + SysRq + c - crashdump
- Alt + SysRq + e - wysyła sygnał TERM do wszystkich procesów z wyjątkiem procesu INIT
- Alt + SysRq + i - wysyła sygnał KILL do wszystkich procesów z wyjątkiem procesu INIT
- Alt + SysRq + k - sekwencja SAK (Secure Access Key), zabicie wszystkich procesów na danej konsoli, np. awaryjne wyłączenie XWindow, gdy Ctrl+Alt+Backspace zawiedzie
- Alt + SysRq + l - wysyła sygnał KILL do wszystkich procesów, z procesem INIT włącznie, skutkuje zatrzymaniem systemu
- Alt + SysRq + m - informacja o pamięci
- Alt + SysRq + o - wyłączenie systemu
- Alt + SysRq + p - wyświetla zawartość rejestrów i flag procesora
- Alt + SysRq + r - przełącza klawiaturę w tryb RAW - gdy np. Ctr+Alt+Del nie działa
- Alt + SysRq + s - synchronizacja dysków i zapisania zawartości buforów dysków
- Alt + SysRq + t - lista wszystkich zadań
- Alt + SysRq + u - przemontowanie wszystkich systemów plików w trybie „tylko do odczytu”

SysRq, który to klawisz?

- x86 - Print Screen
- SPARC - STOP
- szeregowa konsola - Break
- PowerPC - PrintScreen (lub F13)

- Wszystkie - `echo t > /proc/sysrq-trigger`

Jak najbezpieczniej restartować komputer?
Najpierw powinniśmy spróbować Ctrl + Alt + Backspace (XWindow) lub Ctrl + Alt + Del (konsola). Gdy to zawiedzie, to:

- aby zabić wszystkie procesy na danej konsoli: Alt+SysRq+k
- aby bezpiecznie zrestartować komputer: Alt+SysRq+s Alt+SysRq+u Alt+SysRq+b

Więcej informacji na temat „`Magic SysRq key`" można znaleźć w pliku `Documentation/sysrq.txt`.

W niektórych dystrybucjach (lub jeżeli skompilowaliśmy sterowniki naszego kontrolera SCSI / IDE / SATA albo system plików używany na / jako moduł) musimy wygenerować tzw. `initrd`. Powinno to za nas załatwić wydanie polecenia
`$ make install`
Jeżeli używamy `grub`, to plikiem konfiguracyjnym jest `/boot/grub/menu.lst`. Wpisujemy do niego:

```
title Linux 2.6.wersja
       root (hd0,0)
       kernel /boot/vmlinuz-2.6.wersja ro root=/dev/<nasza partycja />
#      Jeżeli używamy initrd
#      initrd /boot/initrd-2.6.wersja.img
```

Użytkownicy `lilo` powinni w pliku `/etc/lilo.conf` wpisać

```
image=/boot/vmlinuz-2.6.wersja
label=linux
initrd=/boot/initrd-2.6.wersja.img
read-only
root=/dev/<nasza partycja />
```

Następnie trzeba powiadomić `lilo` o zmianie konfiguracji
`$ sudo /sbin/lilo`
Powyższe kroki mogą już być wykonane przez skrypt instalacyjny jądra dostępny w naszej dystrybucji. Więcej informacji na temat konfiguracji bootloaderów można znaleźć w „`man lilo.conf`" i „`info grub`".

Poniżej znajduje się skrypt, dzięki któremu można pobrać, skompilować i zainstalować najnowsze jądro -stable. Skrypt jest dość uniwersalny, łatwo też jest dostosować go do własnych potrzeb.

```
#! /bin/sh

# Ścieżka do katalogu ze źródłami jądra
SRC_PATH="/usr/src/kernel/linux-stable"
OBJ_PATH="$SRC_PATH-obj/"
```

```
cd $SRC_PATH
# Pobieramy najnowsze -stable
ketchup 2.6
# Zapamiętujemy wersję
VER=‘ketchup -m‘
# Odświeżamy konfigurację
make O=$OBJ_PATH oldconfig
# Budujemy jądro
make O=$OBJ_PATH
# Instalujemy moduły
sudo make O=$OBJ_PATH modules_install
# Kopiujemy skompresowany obraz jądra do /boot
sudo cp $OBJ_PATH/arch/i386/boot/bzImage /boot/vmlinuz-$VER
# Kopiujemy System.map do /boot
sudo cp $OBJ_PATH/System.map /boot/System.map-$VER
# Jeżeli używamy Fedory, to trzeba wygenerować fedorowe initrd
# sudo /sbin/new-kernel-pkg --make-default --mkinitrd --depmod \\
 --install $VER
```

Poniżej zamieszczam odsyłacze do instrukcji budowania jądra metodą konkretnych dystrybucji

CentOS `http://www.howtoforge.com/kernel_compilation_centos` (ang.)
Debian `http://www.debian.org/doc/manuals/reference/ch-kernel.pl.html`
`http://www.howtoforge.com/howto_linux_kernel_2.6_compile_debian` (ang.)
Fedora `http://www.howtoforge.com/kernel_compilation_fedora` (ang.)
Gentoo `http://www.gentoo.org/doc/pl/kernel-upgrade.xml`
Mandriva `http://www.howtoforge.com/kernel_compilation_mandriva` (ang.)
OpenSUSE `http://www.howtoforge.com/kernel_compilation_suse` (ang.)
Ubuntu `http://www.howtoforge.com/kernel_compilation_ubuntu` (ang.)

Więcej informacji o arkanach konfiguracji i kompilacji jądra można znaleźć w książce Grega Kroah-Hartmana „*Linux Kernel in a Nutshell*” `http://www.kroah.com/lkn/`

Linux posiada bardzo specyficzny model rozwoju - niespotykany w innych projektach otwartych czy zamkniętych, dlatego warto zapoznać się z zasadami nim rządzącymi `http://www.stardust.webpages.pl/ltg/wiki/index.php/Proces_rozwoju_j%C4%85dra_Linux`
Krótkie wskazówki:

- wpisz „make help” i przyjrzyj się dostępnym opcjom
- w menu „`Kernel hacking  --->`” można znaleźć wiele użytecznych opcji
- zapoznaj się z opcjami skryptu `ketchup`

Rozdział 2

Testowanie

Proces testowania możemy podzielić na trzy fazy:

1. Używanie wersji testowej do normalnej pracy.
2. Użycie programów testujących system np. LTP.
3. Wykonywanie działań niekonwencjonalnych.

2.1 „Faza 1”

Pierwsza faza jest prosta - próbujemy uruchomić system:

- na początku w minimalnej konfiguracji z parametrem `init=/bin/bash` i w trybie bez sieci (przekazujemy 1 lub 2 do parametrów jądra - zależy od dystrybucji (sprawdź w dokumentacji dystrybucji lub w pliku `/etc/inittab`, który poziom init uruchamia system bez obsługi sieci) (przydaje się `grub` jako bootloader))
- dopiero po sprawdzeniu działania konfiguracji minimalnych uruchamiamy normalnie system

Następnie ściągamy jakiś plik z sieci, odbieramy pocztę, nagrywamy płytę z danymi/audio (przydaje się CD,DVD-RW) itd.

2.2 „Faza 2 (AutoTest)”

W następnej fazie używamy wyspecjalizowanych programów służących do sprawdzania poprawności działania różnych podsystemów jądra oraz przeprowadzamy testy regresyjne i wydajnościowe. Te ostatnie są bardzo wartościowe dla deweloperów (i dla nas) - jeżeli przesiadka z wersji `2.6.x-rc1` na wersję `2.6.x-rc2` zaowocowała 10% spadkiem wydajności naszego systemu plików, to warto wyśledzić, która łatka to spowodowała.

Do przeprowadzania zautomatyzowanych testów polecam użycie platformy AutoTest `http://test.kernel.org/autotest/`. Składają się na nią różnego rodzaju testy i narzędzia służące do profilowania systemu obudowane w proste w użyciu API. Na początku powinniśmy przejść do zalecanego w dokumentacji katalogu `/usr/local` i jako użytkownik uprzywilejowany

wykonać polecenie „`svn checkout svn://test.kernel.org/autotest/trunk autotest`”. Podczas pracy z AutoTestem będziemy wykorzystywać konto użytkownika uprzywilejowanego, wbrew wszystkim zaleceniom, które mówią, żeby nie pracować na takim koncie. Powód jest bardzo prosty, duża część testów wymaga uprawnień administratora, a AutoTest z założenia ma być platformą nieinteraktywną. Dzięki temu możemy zostawić go na kilka godzin bez opieki i nie martwić się, czy wyskoczy prośba o podanie hasła administratora. (Jest to też kolejny dobry argument za używaniem odseparowanych systemów testowych - gdy coś pójdzie nie tak, to zniszczymy tylko system przeznaczony do testów, a nie do pracy :)

Najprostszym sposobem uruchomienia testu jest wykonanie polecenia „`bin/autotest tests/nazwa_testu/control`” po przejściu do katalogu client. AutoTest wczytuje plik control i wykonuje zawarte w nim instrukcje. Najprostszą postacią pliku `control` jest:
`job.run_test('ltp')`
Nie podaje on żadnych parametrów, które mają być przekazane do testu, zawiera tylko jego nazwę. Bardziej „skomplikowane” pliki control mają postać:
`job.run_test('pktgen', 'eth0', 50000, 0, tag='clone_skb_off')`
`job.run_test('pktgen', 'eth0', 50000, 1, tag='clone_skb_on')`
Przekazują one parametry do programów testowych. Możemy zmodyfikować te parametry po zapoznaniu się z dokumentacją testu, oraz plikiem `tests/nazwa_testu/nazwa_testu.py`.

Gdy AT dowie się już jaki test ma uruchomić i jakie przekazać do niego parametry, uruchamia plik `tests/nazwa_testu/nazwa_testu.py`. Najczęściej jest on trochę dłuższy, niż przedstawiony poniżej, ale ten dobrze obrazuje, co w takim pliku powinno się znaleźć.

```
import test, os_dep
from autotest_utils import *

class isic(test.test):
        version = 1

        # http://www.packetfactory.net/Projects/ISIC/isic-0.06.tgz
        # + http://www.stardust.webpages.pl/files/crap/isic-gcc41-fix.patch

        def setup(self, tarball = 'isic-0.06.tar.bz2'):
                tarball = unmap_url(self.bindir, tarball, self.tmpdir)
                extract_tarball_to_dir(tarball, self.srcdir)
                os.chdir(self.srcdir)

                os_dep.library('libnet.so')
                system('./configure')
                system('make')

        def execute(self, args = '-s rand -d 127.0.0.1 -p 10000000'):
                system(self.srcdir + '/isic ' + args)
```

Pierwszą funkcją zawartą w skrypcie jest funkcja `setup()`, która ma za zadanie przygotować test do działania, czyli wykonuje za nas konfigurację i kompilacje testu. Jak sama nazwa wskazuje, funkcja `execute()` uruchamia test z zadanymi parametrami. Rezultaty wykonania testu można obejrzeć w katalogu `results/default/nazwa_testu/`. Plik `status` zawiera informację o tym, czy test zakończył się pomyślnie, lub czy coś poszło nie tak.

Jeżeli chcemy uruchomić kilka testów jeden po drugim, to najlepiej jest przygotować plik z instrukcjami dla autotestu. Powinien on zawierać te same informacje, które można znaleźć w plikach `control`. Na przykład pierwsze pięć linijek pliku „`samples/all_tests`” wygląda tak:

```
job.run_test('aiostress')
job.run_test('bonnie')
```

```
job.run_test('dbench')
job.run_test('fio')
job.run_test('fsx')
```

Następnie uruchamiamy `autotest` i jako parametr podajemy nasz plik z informacjami kontrolnymi. Możemy też po prostu wykonać polecenie „`bin/autotest samples/all_tests`”, jednak musimy się przygotować na to, że wykonanie wszystkich zawartych w nim testów będzie trwało bardzo długo.

Jeżeli chcemy uruchomić równolegle kilka testów, to plik kontrolny powinien mieć składnie taką, jak przykładowy plik „`samples/parallel`”

```
def kernbench():
        job.run_test('kernbench', 2, 5)

def dbench():
        job.run_test('dbench')

job.parallel([kernbench], [dbench])
```

(W każdej chwili możemy przerwać wykonywanie testu przy użyciu kombinacji Ctrl+c)

Dla ludzi, którzy nie lubią linii poleceń i plików konfiguracyjnych powstał ATCC (AutoTest Control Center). Można go uruchomić poleceniem „`ui/menu`”. Dostajemy do dyspozycji system prostych menu w, których możemy wybrać testy, programy służące do profilowania, obejrzeć rezultaty ich wykonania, oraz przeprowadzić prostą konfigurację.

Jeśli zestaw narzędzi dostarczony z AutoTestem Ci się znudził, to zawsze możesz odwiedzić stronę `http://ltp.sourceforge.net/tooltable.php`. Znajduje się na niej pokaźny spis narzędzi, które można używać do testowania systemu.

2.3 „Faza 3”

Ok, system przeszedł bez problemów dwie pierwsze fazy testów? Możemy zacząć improwizować tzn. robić głupie rzeczy, których nikt nie robi, dlatego nie wie, że można nimi spowodować oops'a :). Ale co takiego należy robić? Gdyby istniał jakiś standardowy sposób, to na pewno trafiłby do jakiegoś zestawu testów.

W fazie trzeciej możemy zacząć od wyciągania i przekładania wtyczek urządzeń podpiętych do magistrali USB. O ile przekładanie wtyczek teoretycznie nie powinno nic zmienić, o tyle notoryczne wyciąganie i wkładanie wtyczki może wywołać oops'a z lasu (pod warunkiem, że ktoś inny tego wcześniej nie próbował przy podobnej konfiguracji PC). Następnie możemy napisać skrypt, który będzie czytał w pętli zawartość plików znajdujących się w katalogu `/proc`. W telegraficznym skrócie faza trzecia polega na robieniu rzeczy, których nie robią zwykli użytkownicy (lub robią bardzo rzadko - po co ktoś miałby w nieskończoność montować i odmontowywać jakiś system plików? :).

2.4 Pomiar wydajności

Jak już wcześniej wspomniałem, warto jest sprawdzać wpływ zmian wprowadzanych w systemie na jego wydajność (nawiasem mówiąc - może to być świetne zajęcie dla początkujących testerów, którzy jeszcze nie chcą testować bardziej rozwojowych wersji systemu, a chcą bardzo pomóc w jego tworzeniu). Ale od czego zacząć i co warto wiedzieć?

Na początku warto jest się zdecydować na testowanie konkretnego podsystemu - wybierzmy jeden, który będziemy testować regularnie - wtedy nasze dane będą o wiele bardziej wartościowe dla deweloperów. Czasami ktoś wysyła na LKML list w stylu: „Witam, zauważyłem spory spadek wydajności mojej karty sieciowej po zaktualizowaniu jądra z `2.6.8` na `2.6.20`. Ktoś ma jakiś pomysł, co z tym zrobić?". Oczywiście, że nikt nie może mieć żadnego pomysłu co się stało, ponieważ pomiędzy wydaniem jądra `2.6.8` a `2.6.20` upłynęło ponad dwa i pół roku (i gadzilion różnych losowych łatek). Jeśli z kolei napiszemy, że pomiędzy wersją `2.6.x-rc3` a `2.6.x-rc4` wydajność naszej karty graficznej spadła o 50%, to łatwo będzie wyśledzić, dlaczego tak się stało. Dlatego regularne wykonywanie pomiarów jest ważne.

Drugą sprawą, na którą trzeba zwrócić uwagę, są szczegóły dotyczące poszczególnych testów - musimy się dowiedzieć jak najwięcej o danym benchmarku. Chodzi przede wszystkim o to, żeby uzyskane wyniki były wiarygodne. Czasami, w różnych publikacjach prasowych/internetowych możemy się spotkać z testem wydajności pod tytułem „time make". Chodzi o sprawdzenie szybkości budowania jądra systemu z pomocą programu time. Jest to test, który należy zaliczyć do grupy niewiarygodnych. Dlaczego? Odpowiedź na to pytanie jest bardzo prosta, jeśli zwrócimy uwagę na to, że pliki przed kompilacją są wczytywane z dysku. Różne dyski mają różną prędkość odczytu, zależy ona również od tego w jakim miejscu znajdują się dane i jak są porozrzucane. O ile prędkość procesora i pamięci RAM można przyjąć za stałą, o tyle szybkość odczytu danych z dysku jest na tyle losowa, że pomiędzy poszczególnymi wynikami mogą występować znaczące różnice. Aby uzyskać wiarygodne wyniki, trzeba więc jakoś ominąć niedoskonałości dysków twardych - najprostszym sposobem będzie buforowanie danych w pamięci operacyjnej (Linux bardzo intensywnie buforuje dane odczytywane z dysków, czym przyspiesza ich późniejszy ponowny odczyt). Jeżeli chcemy przeprowadzać testy pt. „time make" i otrzymywać wiarygodne wyniki, to najlepiej jest użyć skryptu kernbench `http://ck.kolivas.org/kernbench/` lub jego nowszej wersji wchodzącej w skład platformy `AutoTest`.

Kolejną sprawą, o której warto pamiętać, jest stabilność (niezmienność) środowiska, w którym wykonujemy pomiary - czyli nie powinniśmy aktualizować programów, które mogą mieć wpływ na uzyskiwane dane. Jeśli nasze pomiary polegają na kompilacji jądra systemu, to niech to cały czas będzie ta sama wersja źródeł i ten sam plik konfiguracyjny (kompilator i wszystkie potrzebne przy budowaniu jądra narzędzia też nie powinny zmieniać swoich wersji).

Dobrym przykładem na złe testowanie, mogą być porównania wydajności systemów plików, w których nie wzięto pod uwagę czynnika budowy dysku twardego. Jak wcześniej wspomniałem, prędkość odczytu/zapisu w różnych miejscach dysku jest różna - dlatego nie powinniśmy tworzyć pięciu partycji do wykonywania testów pięciu różnych systemach plików. Jedynym sposobem na otrzymanie wiarygodnych wyników jest utworzenie jednej partycji (i nie zmienianie jej położenia), na której będziemy tworzyć różne systemy plików. Pomiędzy kolejnymi

pomiarami najlepiej jest też zrestartować system, aby pominąć czynnik buforowania danych.

Na stronie `http://ltp.sourceforge.net/tooltable.php` można znaleźć kilka dobrych benchmarków (część z nich znajduje się również w `AutoTest`), jednak powinniśmy pamiętać o tym, że nie zależy nam na mierzeniu wydajności sprzętu tylko systemu. Dlatego polecam zapoznanie się z dokumentacją wybranego benchmarku - zazwyczaj znajduje się w niej kilka użytecznych informacji.

Podsumowując, w mierzeniu wydajności najważniejsze są:

- regularne pomiary
- znajomość „szczegółów" pozwalających na uzyskiwanie wiarygodnych wyników
- stabilność środowiska testowego

Jeśli wszystkie trzy czynniki zostaną spełnione, to dane przez nas dostarczane będą nieocenionym źródłem informacji na temat wydajności konkretnego podsystemu.

2.5 Witaj świecie, czyli czego właściwie szukamy?

Czego właściwie szukamy? Poniżej zamieszczam kilka przykładów błędów na które można natrafić podczas testowania systemu.

```
=============================================
[ INFO: possible recursive locking detected ]
---------------------------------------------
idle/1 is trying to acquire lock:
 (lock_ptr){....}, at: [<c021cbd2>] acpi_os_acquire_lock+0x8/0xa

but task is already holding lock:
 (lock_ptr){....}, at: [<c021cbd2>] acpi_os_acquire_lock+0x8/0xa

other info that might help us debug this:
1 lock held by idle/1:
 #0:  (lock_ptr){....}, at: [<c021cbd2>] acpi_os_acquire_lock+0x8/0xa

stack backtrace:
 [<c0103e89>] show_trace+0xd/0x10
 [<c0104483>] dump_stack+0x19/0x1b
 [<c01395fa>] __lock_acquire+0x7d9/0xa50
 [<c0139a98>] lock_acquire+0x71/0x91
 [<c02f0beb>] _spin_lock_irqsave+0x2c/0x3c
 [<c021cbd2>] acpi_os_acquire_lock+0x8/0xa
 [<c0222d95>] acpi_ev_gpe_detect+0x4d/0x10e
 [<c02215c3>] acpi_ev_sci_xrupt_handler+0x15/0x1d
 [<c021c8b1>] acpi_irq+0xe/0x18
 [<c014d36e>] request_irq+0xbe/0x10c
 [<c021cf33>] acpi_os_install_interrupt_handler+0x59/0x87
 [<c02215e7>] acpi_ev_install_sci_handler+0x1c/0x21
 [<c0220d41>] acpi_ev_install_xrupt_handlers+0x9/0x50
 [<c0231772>] acpi_enable_subsystem+0x7d/0x9a
 [<c0416656>] acpi_init+0x3f/0x170
 [<c01003ae>] _stext+0x116/0x26c
 [<c0101005>] kernel_thread_helper+0x5/0xb
```

Powyżej znajduje się błąd znaleziony przez mechanizm sprawdzający poprawność użycia blokad w systemie (`lockdep`).

```
BUG: sleeping function called from invalid context at /usr/src/linux-mm/sound/core/info.c:117
in_atomic():1, irqs_disabled():0
 <c1003ef9> show_trace+0xd/0xf
 <c100440c> dump_stack+0x17/0x19
 <c10178ce> __might_sleep+0x93/0x9d
 <f988eeb5> snd_iprintf+0x1b/0x84 [snd]
 <f988d808> snd_card_module_info_read+0x34/0x4e [snd]
 <f988f197> snd_info_entry_open+0x20f/0x2cc [snd]
 <c1067a17> __dentry_open+0x133/0x260
 <c1067bb7> nameidata_to_filp+0x1c/0x2e
 <c1067bf7> do_filp_open+0x2e/0x35
 <c1068bf2> do_sys_open+0x54/0xd7
 <c1068ca1> sys_open+0x16/0x18
 <c11dab67> sysenter_past_esp+0x54/0x75
BUG: using smp_processor_id() in preemptible [00000001] code: init/1
caller is __handle_mm_fault+0x2b/0x20d
 [<c0103ba8>] show_trace+0xd/0xf
 [<c0103c7a>] dump_stack+0x17/0x19
 [<c0203bcc>] debug_smp_processor_id+0x8c/0xa0
 [<c0160e60>] __handle_mm_fault+0x2b/0x20d
 [<c0116f7b>] do_page_fault+0x226/0x61f
 [<c0103959>] error_code+0x39/0x40
 [<c019d4c1>] padzero+0x19/0x28
 [<c019e716>] load_elf_binary+0x836/0xc02
 [<c017db53>] search_binary_handler+0x123/0x35a
 [<c019d3b9>] load_script+0x221/0x230
 [<c017db53>] search_binary_handler+0x123/0x35a
 [<c017deee>] do_execve+0x164/0x215
 [<c0101e7a>] sys_execve+0x3b/0x7e
 [<c02fabc3>] syscall_call+0x7/0xb
```

Poniżej znajduje się tzw. `oops` - po wystąpieniu którego, bardzo często nasz system kończy działanie tzw. `kernel panic`. Czyli stało się coś naprawdę złego i nasz system przestaje działać, żeby nie ryzykować utraty danych lub innych przykrych doświadczeń. (W artykule OSWeekly.com Puru Govind wyjaśnił skąd się wziął kernel panic i co może oznaczać `http://www.osweekly.com/index.php?option=com_content&task=view&id=2241&Itemid=449` (ang.))

```
BUG: unable to handle kernel paging request at virtual address 6b6b6c07
 printing eip:
c0138722
*pde = 00000000
Oops: 0002 [#1]
4K_STACKS PREEMPT SMP
last sysfs file: /devices/pci0000:00/0000:00:1d.7/uevent
Modules linked in: snd_timer snd soundcore snd_page_alloc intel_agp agpgart
ide_cd cdrom ipv6 w83627hf hwmon_vid hwmon i2c_isa i2c_i801 skge af_packet
ip_conntrack_netbios_ns ipt_REJECT xt_state ip_conntrack nfnetlink xt_tcpudp
iptable_filter ip_tables x_tables cpufreq_userspace p4_clockmod speedstep_lib
binfmt_misc thermal processor fan container rtc unix
CPU:    0
EIP:    0060:[<c0138722>]    Not tainted VLI
EFLAGS: 00010046   (2.6.18-rc2-mm1 #78)
EIP is at __lock_acquire+0x362/0xaea
eax: 00000000   ebx: 6b6b6b6b   ecx: c0360358   edx: 00000000
esi: 00000000   edi: 00000000   ebp: f544ddf4   esp: f544ddc0
ds: 007b   es: 007b   ss: 0068
Process udevd (pid: 1353, ti=f544d000 task=f6fce8f0 task.ti=f544d000)
Stack: 00000000 00000000 00000000 c7749ea4 f6fce8f0 c0138e74 000001e8 00000000
        00000000 f6653fa4 00000246 00000000 00000000 f544de1c c0139214 00000000
        00000002 00000000 c014fe3a c7749ea4 c7749e90 f6fce8f0 f5b19b04 f544de34
Call Trace:
[<c0139214>] lock_acquire+0x71/0x91
[<c02f2bfb>] _spin_lock+0x23/0x32
[<c014fe3a>] __delayacct_blkio_ticks+0x16/0x67
[<c01a4f76>] do_task_stat+0x3df/0x6c1
[<c01a5265>] proc_tgid_stat+0xd/0xf
[<c01a29dd>] proc_info_read+0x50/0xb3
```

```
[<c0171cbb>] vfs_read+0xcb/0x177
[<c017217c>] sys_read+0x3b/0x71
[<c0103119>] sysenter_past_esp+0x56/0x8d
DWARF2 unwinder stuck at sysenter_past_esp+0x56/0x8d
Leftover inexact backtrace:
[<c0104318>] show_stack_log_lvl+0x8c/0x97
[<c010447f>] show_registers+0x15c/0x1ed
[<c01046c2>] die+0x1b2/0x2b7
[<c0116f5f>] do_page_fault+0x410/0x4f0
[<c0103d1d>] error_code+0x39/0x40
[<c0139214>] lock_acquire+0x71/0x91
[<c02f2bfb>] _spin_lock+0x23/0x32
[<c014fe3a>] __delayacct_blkio_ticks+0x16/0x67
[<c01a4f76>] do_task_stat+0x3df/0x6c1
[<c01a5265>] proc_tgid_stat+0xd/0xf
[<c01a29dd>] proc_info_read+0x50/0xb3
[<c0171cbb>] vfs_read+0xcb/0x177
[<c017217c>] sys_read+0x3b/0x71
[<c0103119>] sysenter_past_esp+0x56/0x8d
Code: 68 4b 75 2f c0 68 d5 04 00 00 68 b9 75 31 c0 68 e3 06 31 c0 e8 ce 7e fe ff
e8 87 c2 fc ff 83 c4 10 eb 08 85 db 0f 84 6b 07 00 00 <f0> ff 83 9c 00 00 00 8b
55 dc 8b 92 5c 05 00 00 89 55 e4 83 fa
EIP: [<c0138722>] __lock_acquire+0x362/0xaea SS:ESP 0068:f544ddc0
```

Poniższy typ błędu wynika najczęściej z sytuacji o której deweloperzy mówią, że nie powinna mieć miejsca :).

```
KERNEL: assertion ((int)tp->lost_out >= 0) failed at net/ipv4/tcp_input.c (2148)
KERNEL: assertion ((int)tp->lost_out >= 0) failed at net/ipv4/tcp_input.c (2148)
KERNEL: assertion ((int)tp->sacked_out >= 0) failed at net/ipv4/tcp_input.c (2147)
KERNEL: assertion ((int)tp->sacked_out >= 0) failed at net/ipv4/tcp_input.c (2147)
```

```
BUG: warning at /usr/src/linux-mm/kernel/cpu.c:56/unlock_cpu_hotplug()
 [<c0103e41>] dump_trace+0x70/0x176
 [<c0103fc1>] show_trace_log_lvl+0x12/0x22
 [<c0103fde>] show_trace+0xd/0xf
 [<c01040b0>] dump_stack+0x17/0x19
 [<c0140e19>] unlock_cpu_hotplug+0x46/0x7c
 [<fd9560b0>] cpufreq_set+0x81/0x8b [cpufreq_userspace]
 [<fd956109>] store_speed+0x35/0x40 [cpufreq_userspace]
 [<c02ac9f2>] store+0x38/0x49
 [<c01aec16>] flush_write_buffer+0x23/0x2b
 [<c01aec69>] sysfs_write_file+0x4b/0x6c
 [<c01770af>] vfs_write+0xcb/0x173
 [<c0177203>] sys_write+0x3b/0x71
 [<c010312d>] sysenter_past_esp+0x56/0x8d
 [<b7fbe410>] 0xb7fbe410
 [<c0103fc1>] show_trace_log_lvl+0x12/0x22
 [<c0103fde>] show_trace+0xd/0xf
 [<c01040b0>] dump_stack+0x17/0x19
 [<c0140e19>] unlock_cpu_hotplug+0x46/0x7c
 [<fd9560b0>] cpufreq_set+0x81/0x8b [cpufreq_userspace]
 [<fd956109>] store_speed+0x35/0x40 [cpufreq_userspace]
 [<c02ac9f2>] store+0x38/0x49
 [<c01aec16>] flush_write_buffer+0x23/0x2b
 [<c01aec69>] sysfs_write_file+0x4b/0x6c
 [<c01770af>] vfs_write+0xcb/0x173
 [<c0177203>] sys_write+0x3b/0x71
 [<c010312d>] sysenter_past_esp+0x56/0x8d
```

Tutaj mamy wynik skanowania pliku `/sys/kernel/debug/memleak` (kernel memory leak detector jeszcze nie jest częścią standardowego jądra)

```
orphan pointer 0xf5a6fd60 (size 39):
c0173822: <__kmalloc>
```

```
c01df500: <context_struct_to_string>
c01df679: <security_sid_to_context>
c01d7eee: <selinux_socket_getpeersec_dgram>
f884f019: <unix_get_peersec_dgram>
f8850698: <unix_dgram_sendmsg>
c02a88c2: <sock_sendmsg>
c02a9c7a: <sys_sendto>
```

Dzięki tym informacjom, oraz plikowi konfiguracyjnemu naszego jądra deweloperzy mogą poprawić błąd, który udało się nam znaleźć.

Poza wymienionymi wyżej błędami możemy spotkać się z sytuacją, że jądro nie będzie się chciało zbudować z powodu błędu na etapie kompilacji. Jest to bardzo rzadko spotykany problem i świadczy o wyjątkowej niedbałości autora kodu. Dobrym przykładem może być moja poprawka do sterownika bufora ramki dla kart firmy SIS, która nie kompilowała się jako moduł...

Czasami powodem wystąpienia błędu jest błędne działanie np. naszego kompilatora, który w niewłaściwy sposób buduje jądro. Również programy, które wykorzystują niektóre części ABI (Application Binary Interface) mogą przestać działać poprawnie po jego zmianie w nowej wersji - w takich wypadkach ważne jest ustalenie czy zmiana ABI była zamierzona czy też przypadkowa.

Część problemów nie zawsze jest widoczna od razu, niektóre występują tylko w określonych sytuacjach i mogą się objawiać np. zawieszaniem losowych procesów, wyrzucaniem lub wrzucaniem losowych danych do zapisywanych plików etc. Jednym słowem trzeba mieć oczy i uszy szeroko otwarte.
Błędy możemy klasyfikować na kilka różnych sposobów, jednak najbardziej użyteczna dla nas klasyfikacja dzieli je na dwa typy:

- łatwe do odtworzenia - takie, o których dokładnie wiemy kiedy i gdzie występują
- trudne do odtworzenia - występują (pozornie) losowo i nie wiemy dokładnie jak je odtworzyć

Pierwszy typ jest najłatwiejszy do naprawienia, bardzo łatwo można znaleźć łatkę która dany błąd wprowadziła. Drugi typ jest tym typem, którego nikt nie lubi. Jeżeli nie mamy odpowiedniego doświadczenia, to będziemy musieli zdać się na doświadczenie deweloperów.

2.6 Sterowniki binarne i jądra dystrybucyjne

Bardzo często słyszymy, że sterowniki binarne są złe i nie należy ich stosować - ale dlaczego? Powód jest prosty - jeżeli znajdziemy błąd i wyślemy raport na Linux Kernel Mailing List, to deweloperzy nie będą nam w stanie pomóc, ponieważ nie mają dostępu do zamkniętego kodu sterownika. Informację o użytym sterowniku można otrzymać z linijki „`EIP: 0060:[<c046c7c3>] Tainted: P VLI`”, litera „`P`” informuje nas, że został użyty sterownik z zamkniętym kodem. Taki błąd należy spróbować odtworzyć bez załadowanego binarnego bloba. Powinniśmy zgłosić błąd do twórców sterownika i niech oni się martwią o jego poprawienie. W pliku `Documentation/oops-tracing.txt` możemy znaleźć listę powodów, dla których dane jądro zostało uznane za „skażone”. Podsumowując: sterowniki binarne

są bardzo przydatne - dostrzeżone błędy należy wysyłać do ich producentów, nie na LKML.

Czasami ludzie wysyłają na LKML raporty błędów znalezionych w jądrach dystrybucyjnych, nie należy tego robić z kilku powodów:

- jądra dystrybucyjne zawierają modyfikacje, które nie są dostępne w standardowej wersji
- często te modyfikacje zawierają błędy (czasami są bardzo eksperymentalne)
- błąd najprawdopodobniej nie występuje w wersji oryginalnej
- to niełądnie obarczać ludzi odpowiedzialnością za błędy, których najprawdopodobniej nie popełnili :)

Powinniśmy spróbować odtworzyć taki błąd na jądrze z `kernel.org`, jeżeli się nie uda, to jest to błąd producenta dystrybucji i do niego powinien zostać zgłoszony.

Rozdział 3

Przechwytywanie błędów

Istnieje kilka metod przechwytywania błędów - jedne bardziej, drugie mniej skuteczne - każda z nich ma swoje zalety i wady. Najpopularniejszą metodą jest użycie `sysklogd` - demona, który zapisuje informacje wypisywane przez jądro systemu do pliku logowania (plikiem tym może być `/var/log/messages` lub `/var/log/kern.log` - zależy od konfiguracji `sysklogd` w naszej dystrybucji). Zaletą tej metody jest to, że w praktycznie każdej dystrybucji `sysklogd` jest dostępny domyślnie i nie trzeba go konfigurować. Do wad należy zaliczyć to, że jest on uruchamiany bardzo późno podczas procesu startu systemu (i bardzo wcześnie wyłączany podczas jego zamykania), co bardzo często uniemożliwia przechwycenie błędu występującego we wczesnej fazie uruchamiania systemu, lub późnej fazie jego zamykania.

W metodzie, którą możemy nazwać „tradycyjną", wykorzystujemy kartkę papieru i coś do pisania... Jest ona bardzo czasochłonna i nikt jej nie lubi, ze względu na to, że można popełnić wiele błędów podczas przepisywania wszystkich literek i cyferek. Czasami treść błędu ucieka nam z ekranu i niewiele możemy na to poradzić, poza uruchomieniem systemu z parametrem „`vga=1`", dzięki temu będziemy mieli konsolę w rozdzielczości 80 kolumn na 50 linii. Niestety przy „ciekawszych" błędach 50 linijek może się okazać niewystarczającą ilością. Zamiast przepisywać treść błędu, możemy go po prostu sfotografować i umieścić taki „zrzut ekranu" na naszym www (warto zmniejszyć rozdzielczość do np. 1024x768 i zmienić skalę kolorów na skalę szarości, zdjęcie będzie trochę mniejsze). Nie powinniśmy wysyłać zdjęć na LKML w załącznikach, ponieważ najczęściej są za duże (LKML ma ograniczoną wielkość listu do 100KB).

3.1 Konsola szeregowa

Metoda przechwytywania błędów za pomocą konsoli szeregowej ma jedną zasadniczą wadę - wymaga użycia dwóch komputerów. Łączymy je za pomocą kabla szeregowego podpiętego pod jeden z portów COM (najczęściej COM1). Na początku musimy skonfigurować konsole szeregową w jądrze

```
Device Drivers  --->
Character devices  --->
Serial drivers  --->
<*> 8250/16550 and compatible serial support
```

```
[*]   Console on 8250/16550 and compatible serial port
```

Następnie dopisujemy do `/boot/grub/menu.lst`
`serial --unit=0 --speed=115200 --word=8 --parity=no --stop=1 terminal --timeout=5 serial console`
oraz
`console=ttyS0,115200n8 console=tty0`
do parametrów z jakimi uruchamiamy jądro.

Wskazówki zamieszczone poniżej są specyficzne dla dystrybucji Fedora, w innych dystrybucjach wykonanie poniższych kroków może nie być potrzebne. Edytujemy plik `/etc/sysconfig/init` i zmiennej `BOOTUP` nadajemy wartość „`serial`". Następnie w pliku `/etc/sysconfig/kudzu` włączamy tryb „`SAFE=yes`" a w pliku `/etc/securetty` dopisujemy „`ttyS0`".
Teraz w pliku `/etc/inittab` dopisujemy linijkę
`S1:23:respawn:/sbin/mgetty -L ttyS0 115200 vt100`
(musimy mieć zainstalowany pakiet `mgetty`).

Opisane powyżej czynności warto przeprowadzić na obu komputerach - konsola szeregowa zawsze może się przydać. Jeśli jej nie potrzebujemy, wystarczy że zrobimy komentarz przed parametrami ją włączającymi.

Na komputerze, który będzie służył do przechwytywania danych uruchamiamy program
`# minicom -o -C log.txt`
Prawdopodobnie otrzymamy błąd „`Device /dev/modem/ acces failed: (..)`", aby temu zaradzić wystarczy utworzyć dowiązanie symboliczne do `ttyS0`
`# ln -s /dev/ttyS0 /dev/modem`
Pamiętaj o ustawieniu odpowiednich parametrów transmisji!

Więcej informacji na temat konfiguracji konsoli szeregowej możemy znaleźć w pliku
`Documentation/serial-console.txt`

3.2 Konsola sieciowa

Największą wadą tej metody przechwytywania błędów (tak jak w przypadku konsoli szeregowej) jest to, że wymaga użycia dwóch komputerów. Z kolei niewątpliwą zaletą jest to, że komputery mogą być w znacznej odległości od siebie - nie ogranicza nas długość kabla szeregowego. Trochę mniejszą wadą konsoli sieciowej jest to, że zaczyna działać dopiero po uruchomieniu sieci, więc nie będzie się nadawała do przechwytywania wszystkich błędów. (Niestety konsola sieciowa działa tylko na kartach Ethernet.)

Konfigurację zaczynamy od wybrania jednej opcji podczas konfiguracji jądra

```
Device Drivers  --->
Network device support  --->
<*>   Network console logging support (EXPERIMENTAL)
```

Następnie do parametrów jądra dodajemy linijkę podobną do poniższej
`netconsole=4444@192.168.100.1/eth0,6666@192.168.100.2/00:14:38:C3:3F:C4`

Pierwsze trzy parametry dotyczą komputera, z którego będziemy przechwytywać dane:

- 4444 - oznacza port na którym będą one wysyłane
- 192.168.100.1 - adres IP interfejsu
- eth0 - interfejs przez który będziemy wysyłać dane

Kolejne trzy parametry odnoszą się do komputera na którym będziemy przechwytywać błąd:

- 6666 - port na którym będziemy nasłuchiwać
- 192.168.100.2 - adres IP interfejsu, który będzie odbierał dane
- 00:14:38:C3:3F:C4 - MAC adres karty, która będzie odbierać dane

Następnie na komputerze, który posłuży nam do przechwytywania błędów uruchamiamy np. program `netcat` `http://netcat.sourceforge.net/`

```
# netcat -u -l -p 6666
```

Użyteczne informacje na temat konsoli sieciowej można znaleźć w `Documentation/networking/netconsole.txt`

Krótkie wskazówki:

- zapoznaj się z poleceniem `dmesg`
- znajdź plik w którym twoja ulubiona dystrybucja przechowuje logi jądra

Rozdział 4

Git, quilt i przeszukiwanie binarne

Do zarządzania drzewami deweloperzy używają dwóch podstawowych narzędzi `git` i `quilt`. Można też używać kilku nakładek (Cogito, Stacked GIT, Patchy Git (pg), (h)gct), jednak skupię się na opisaniu dwóch pierwszych narzędzi, ponieważ zasadniczo różnią się sposobem działania i podejściem do zarządzania drzewem.

4.1 Git

Git jest systemem kontroli wersji napisanym przez Linusa Torvaldsa na potrzeby jądra Linux (aktualnym opiekunem `git` jest Junio C. Hamano). Instalacja sprowadza się do zainstalowania pakietu `git-core` z naszej ulubionej dystrybucji lub pobrania źródeł `http://www.kernel.org/pub/software/scm/git/` i samodzielnej kompilacji.

(Przed przystąpieniem do klonowania jakiegoś drzewa dobrze się nad tym zastanów - czy będzie Ci ono potrzebne? Jeżeli nie planujesz prowadzenia intensywnych testów, to może lepiej korzystać ze zwykłych łatek? Aktualnie drzewo `linux-2.6` zajmuje trochę ponad 600 MB, ze względu na duże obciążenie `kernel.org` klonowanie będzie trwać bardzo długo...)

Pierwszym naszym krokiem po zainstalowaniu git'a powinno być sklonowanie jakiegoś repozytorium np. tego którym zarządza Linus

```
$ git-clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/
linux-2.6.git linux-git
```

Klonowanie repozytorium może trwać bardzo długo - zależy od szybkości naszego połączenia internetowego. Po przejściu do katalogu `linux-git` wykonujemy polecenie

```
$ git-checkout -f
```

Teraz mamy już dostęp do aktualnego drzewa „surowego". Gdy chcemy uaktualnić nasze lokalne repozytorium wpisujemy

```
$ git-pull
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

a następnie

```
$ git-checkout
```

Uwaga! Gdy chcemy uaktualnić lokalne repozytorium, to **nie powinniśmy** wykonywać tego w następujący sposób

```
$ git-clone $drzewo $katalog
```

```
$ rm -rf $katalog
$ git-clone $drzewo $katalog
```
Do uaktualniania repozytorium należy używać programu `git-pull`! Klonując całe drzewo marnujemy cenne zasoby łącza `kernel.org` (piszę o tym tylko dlatego, ponieważ widziałem, że niektórzy ludzie tak robią).

Dodanie flagi „`-f`" do `git-checkout` cofnie wszystkie zmiany, jakie dokonaliśmy w plikach źródłowych. Najczęściej używane przeze mnie polecenia `git` to:

- `git-whatchanged plik` - pokazuje wszystkie zmiany, jakie zostały wprowadzone w danym pliku
- `git-bisect *` - przeszukiwanie binarne repozytorium
- `git-pull *` - pobiera najnowszą wersję drzewa
- `git-revert *` - odwraca commit (łatkę)
- `gitk` - graficzny program do wizualizacji drzew

Bardzo użytecznym programem jest `git-log` - pokazuje on listę zmian wprowadzonych w drzewie. Przeważnie będziemy go używać w następujący sposób:
```
$ git log
```
Możemy też wyświetlić wszystkie zmiany jakie zaszły od wersji `2.6.19`
```
$ git log v2.6.19..
```
lub przez ostatnie siedem dni
```
$ git log since=7 days ago
```

Za pomocą git-show wyświetlimy interesujący nas commit np.
```
$ git-show b5bf28cde894b3bb3bd25c13a7647020562f9ea0
```
Warto też zwrócić uwagę na to, że nie musimy wpisywać/wklejać całej nazwy obiektu - czasami wystarczy pierwszych kilka znaków. Ten sam efekt uzyskamy więc po wydaniu polecenia
```
$ git-show b5bf28
```
Bardzo przydatnym zastosowaniem git-show jest
```
$ git-show b5bf28 > latka.patch
```
W ten oto prosty sposób wyciągamy konkretną łatkę z drzewa.

Do transferu obiektów powinniśmy używać protokołu `git://` - jest on przeznaczony specjalnie do tego celu. Jednak w pewnych sytuacjach (np. gdy jesteśmy za firewallem, którego nie możemy skonfigurować) możemy użyć zwykłego protokołu `http://` - nie jest to jednak polecane rozwiązanie.

Sporą kolekcję różnych drzew można znaleźć pod adresami `http://git.kernel.org/` oraz `http://git.infradead.org/`

Więcej informacji o git można uzyskać pod adresem `http://git.or.cz/`

4.2 Quilt

Quilt jest narzędziem służącym do zarządzania seriami łatek, najczęściej taka seria aplikuje się na konkretną wersję drzewa Linusa np. `2.6.18-rc4`. Możemy go zainstalować z paczki dostarczonej z dystrybucją lub pobrać ze strony projektu `https://savannah.nongnu.org/projects/quilt/` . Najszybszym sposobem na nałożenie łatek jest skopiowanie ich do katalogu `patches` w katalogu ze źródłami jądra, następnie kopiujemy plik `series` do folderu głównego.

Przykładowy folder `patches` może zawierać cztery pliki:

```
łatka01.patch
łatka02.patch
łatka03.patch
łatka04.patch
```

W pliku series mamy łatki poukładane w tej samej kolejności. Aby nałożyć całą serię musimy wykonać polecenie „`quilt push -a`" (ew. możemy podać nazwę łatki lub jej numer):

```
$ quilt push -a
$ quilt push łatka04.patch
$ quilt push 4
```

Jeżeli wszystko będzie w porządku, to zobaczymy serię komunikatów:

```
Applying patch patches/łatka01.patch
patching file include/linux/kernel.h
patching file include/linux/memleak.h
patching file init/main.c
Applying patch patches/łatka02.patch
patching file arch/i386/kernel/vmlinux.lds.S
patching file include/asm-i386/processor.h
Applying patch patches/łatka03.patch
patching file kernel/fork.c
Applying patch patches/łatka04.patch
patching file kernel/delayacct.c
Now at patch patches/łatka04.patch
```

Gdy chcemy wyrzucić wszystkie łatki wystarczy wykonać „`quilt pop -a`". Możemy wyrzucić też dwie ostatnie łatki wpisując:
`$ quilt pop łatka02.patch`
Wyrzucamy wszystkie łatki zaaplikowane po pliku „`łatka02.patch`". Dzięki temu możemy w prosty sposób przeprowadzić przeszukiwanie binarne o którym za chwilę napiszę.

Zarządzanie łatkami za pomocą `quilt` odbywa się na zasadzie stosu FIFO, czyli pierwsza łatka wymieniona w pliku series jest aplikowana jako pierwsza. Gdy chcemy dodać do drzewa naszą łatkę, powinniśmy napisać ją w oparciu o całe drzewo (wszystkie łatki zaaplikowane), a następnie dopisać ją jako ostatnią do pliku `series`. Ma to duże znaczenie - szczególnie przy bardzo dużych drzewach, jak -mm - ponieważ jeden plik może być zmieniany przez większą ilość łatek. Powinny być one wykonywane addytywnie.

4.3 Przeszukiwanie binarne - idea

Wyobraźmy sobie następującą sytuację - znaleźliśmy błąd, ale nie mamy zielonego pojęcia która z ponad tysiąca łatek wchodzących w skład drzewa go wywołuje. Wyrzucanie łatek po kolei byłoby mało rozsądne, dlatego stosujemy przeszukiwanie binarne. Na czym to polega? Mamy serię dziesięciu łatek:

```
łatka01.patch
łatka02.patch
[..]
łatka10.patch
```

Wiemy, że po nałożeniu wszystkich otrzymujemy błędnie działający system. Dlatego najpierw nakładamy pierwsze pięć i przeprowadzamy test. Jeżeli błąd występuje nadal, to musi wywoływać go jedna z tych pięciu łatek. Odwracamy dwie ostatnie łatki (zostają trzy pierwsze) i przeprowadzamy test. Okazało się, że błąd już nie występuje, należy wyciągnąć z tego wniosek, że wywołała go czwarta lub piąta łatka. Aplikujemy czwartą łatkę. Jeżeli wystąpił błąd, to spowodowała go czwarta łatka, w przeciwnym wypadku naszym podejrzanym zostaje łatka numer pięć.

Powyższy przykład ilustruje idee przeszukiwania binarnego, jednak kiepsko obrazuje jego skuteczność. Jest ona bardzo wysoka, ponieważ za każdym „przebiegiem" odrzucamy połowę łatek jakie zostały nam do przetestowania - przy grubo ponad tysiącu łatkach (standardowy rozmiar drzewa -mm) pierwszy „przebieg" odrzuca ponad pięćset łatek! Przeszukiwanie binarne jest algorytmem klasy O(log2N), więc w łatwy sposób możemy obliczyć (najlepiej za pomocą kalkulatora :) ile razy będziemy musieli przebudować system, aby znaleźć łatkę wprowadzającą błąd.

4.4 Przeszukiwanie binarne z pomocą quilt

Plik `series` zawiera spis następujących łatek:

```
01-kmemleak-base.patch
02-kmemleak-doc.patch
03-kmemleak-hooks.patch
04-kmemleak-modules.patch
05-kmemleak-i386.patch
06-kmemleak-arm.patch
07-kmemleak-false-positives.patch
08-kmemleak-keep-init.patch
09-kmemleak-test.patch
10-kmemleak-maintainers.patch
#11-new-locking.patch
#12-new-locking-fix.patch
13-vt-memleak-fix.patch
14-new-locking-fix.patch
15-fix-for-possible-leak-in-delayacctc.patch
```

Niestety nie możemy zbudować jądra po nałożeniu wszystkich, więc przeprowadzamy przeszukiwanie binarne.

Gdy mamy tylko piętnaście łatek to łatwiej jest sprawdzić, która z nich modyfikuje niekompilujący się plik - „`quilt patches jakiś_plik.c`". Jednak w przypadku dużych drzew bardzo często nie będziemy mogli wyrzucić łatek znajdujących się „w środku".

A wiec zaczęliśmy od zaaplikowania wszystkich łatek

```
$ quilt push -a
Applying patch patches/01-kmemleak-base.patch
patching file include/linux/kernel.h
[..]
patching file kernel/delayacct.c
Now at patch patches/15-fix-for-possible-leak-in-delayacctc.patch
```

jednak coś poszło nie tak jak miało i nie możemy skompilować systemu. Kopiujemy plik series np. do katalogu domowego i otwieramy w ulubionym edytorze tekstu, który wyświetla numer linii w której znajduje się kursor. Wybieramy łatkę, która znajduje się po środku pomiędzy
`01-kmemleak-base.patch`
a
`15-fix-for-possible-leak-in-delayacctc.patch`
wypadło na
`07-kmemleak-false-positives.patch`
Wycofujemy łatki znajdujące się po `07-kmemleak-false-positives.patch`.

```
$ quilt pop 07-kmemleak-false-positives.patch
Removing patch patches/15-fix-for-possible-leak-in-delayacctc.patch
Restoring include/linux/delayacct.h
[..]
Restoring lib/Kconfig.debug
Now at patch patches/07-kmemleak-false-positives.patch
```

Błąd występuje nadal? Wpisujemy sobie do kopi pliku `series` „ZŁA" za siódmą poprawką i ponownie wybieramy łatkę. Nasz wybór padł na `04-kmemleak-modules.patch`
`$ quilt pop 04-kmemleak-modules.patch`
Błąd zniknął? Wpisujemy za czwartą poprawką „DOBRA", zostały już tylko trzy
`$ quilt push 06-kmemleak-arm.patch`
i tak dalej, aż dojedziemy do

```
06-kmemleak-arm.patch
DOBRA
07-kmemleak-false-positives.patch
ZŁA
```

W „*How to perform bisection searches on -mm trees*" `http://www.zip.com.au/~akpm/linux/patches/stuff/bisecting-mm-trees.txt` Andrew Morton radzi aby podczas przeszukiwania binarnego w drzewie -mm nie dzielić łatek

```
łatka-bla.patch
łatka-bla-bla.patch
łatka-bla-bla-fix1.patch
łatka-bla-bla-fix2.patch
łatka-bla-bla-fix3.patch
```

zaaplikujmy wszystkie, lub nie aplikujmy żadnej.

Poniższy filmik pokazuje praktyczny przykład zastosowania wyszukiwania binarnego za pomocą quilt `http://www.youtube.com/watch?v=LS_hTnBDIYk`

4.5 Przeszukiwanie binarne z pomocą git-bisect

Git-bisect jest moim ulubionym narzędziem do przeszukiwania binarnego - prawdziwy killer-app. W `2.6.18-rc5` napotkaliśmy mały problem i nie bardzo wiemy który commit go spowodował, wiemy jednak że wersja `2.6.18-rc4` działała dobrze. Zaczynamy nasze poszukiwania:
`$ git-bisect start`
Oznaczamy aktualną wersję jako złą
`$ git-bisect bad`
Wersję `2.6.18-rc4` (dzięki `gitk` dowiemy się, że jest to commit 9f737633e6ee54fc174282d49b2559bd2208391d) oznaczymy jako dobrą:
`$ git-bisect good 9f737633e6ee54fc174282d49b2559bd2208391d`
Teraz `git` nam wybierze wersję, która znajduje się pomiędzy nimi. Wybrał:

```
Bisecting: 202 revisions left to test after this
[c5ab964debe92d0ec7af330f350a3433c1b5b61e] spectrum_cs: Fix firmware
uploading errors
```

Możemy sobie zwizualizować drzewo poleceniem „`git-bisect visualize`". Następnie budujemy i testujemy system. Powiedzmy, że błąd nadal występuje, dlatego oznaczamy aktualną wersję jako złą `git-bisect` wybierze nam nowego kandydata.

```
$ git-bisect bad
Bisecting: 101 revisions left to test after this
[1d7ea7324ae7a59f8e17e4ba76a2707c1e6f24d2] fuse: fix error case in
fuse_readpages
```

Budujemy, testujemy itd. Okazało się, że błąd już nie występuje, więc oznaczamy tę wersję jako dobrą:

```
git-bisect good
Bisecting: 55 revisions left to test after this
[a4657141091c2f975fa35ac1ad28fffdd756091e]
Merge gregkh@master.kernel.org:/pub/scm/linux/kernel/git/davem/net-2.6
```

Cały proces kontynuujemy do chwili otrzymania komunikatu podobnego do poniższego

```
$ git-bisect good
1d7ea7324ae7a59f8e17e4ba76a2707c1e6f24d2 is first bad commit
commit 1d7ea7324ae7a59f8e17e4ba76a2707c1e6f24d2
Author: Jan Kowalski <a@b>
Date:   Sun Aug 13 23:24:27 2006 -0700

    [PATCH] fuse: fix error case in fuse_readpages

    Don't let fuse_readpages leave the @pages list not empty when exiting
    on error.
[..]
```

Jest w nim podany numer commitu wprowadzającego nasz błąd. Aby się upewnić, że wszystko działa poprawnie bez tej łatki musimy zresetować `git-bisect` (powrócimy do wersji `2.6.18-rc5`) i wyrzucić zły commit.

```
git-bisect reset
git-revert 1d7ea7324ae7a59f8e17e4ba76a2707c1e6f24d2
```

Podczas przeszukiwania binarnego za pomocą `git-bisect` najczęściej będziemy korzystać z poniższych poleceń:

- `git-bisect start` - rozpoczyna proces przeszukiwania
- `git-bisect reset` - kończy proces
- `git-bisect good <commit>` - oznacza aktualną wersję jako dobrą, możemy podać jako parametr commit
- `git-bisect bad <commit>` - oznacza aktualną wersję jako złą, możemy podać jako parametr commit
- `git-bisect visualize` - włącza „`gitk`” i pokazuje poprawki pomiędzy ostatnią dobrą i złą wersją

Polecam też przeczytanie „`man git-bisect`” - są w nim opisane dodatkowe opcje.

Poniższy filmik pokazuje praktyczny przykład zastosowania wyszukiwania binarnego za pomocą `git-bisect` `http://www.youtube.com/watch?v=R7_LY-ceFbE`

4.6 Uwaga na „make oldconfig”

Podczas prowadzenia przeszukiwania binarnego trzeba pamiętać o pewnej bardzo ważnej rzeczy - nie należy używać „`make oldconfig`”! Wiąże się to z tym, że wraz z nowymi łatkami dochodzą nowe opcje (czasami też znikają) - gdy np. cofniemy się o pięćdziesiąt łatek i stwierdzimy, że dana wersja działa dobrze, to nakładamy kolejne dwadzieścia pięć łatek. Teraz możemy nie pamiętać o tym, że opcja XY była wcześniej włączona - jeśli nie włączymy jej ponownie, to błąd może się już nie pojawić... To bardzo często popełniany błąd, podczas

przeszukiwania binarnego, dlatego warto o tym pamiętać.

Najlepiej jest więc, skopiować nasz plik konfiguracyjny jądra do innego katalogu przed rozpoczęciem przeszukiwania binarnego i za każdym razem przed kompilacją kopiować go do katalogu ze źródłami jądra. Takie postępowanie powinno nas uchronić przed popełnieniem błędu.

Rozdział 5

Zgłaszanie błędów

Raporty dotyczące usterek w jądrze należy wysyłać na LKML, na listę danego podsystemu (jeżeli taka istnieje - spis wszystkich list znajduje się pod adresem `http://vger.kernel.org/vger-lists.html`), oraz do opiekuna kodu w którym wystąpiła usterka. Jeżeli wiemy, która łatka wywołała problem, to powinniśmy powiadomić o tym deweloperów - najlepiej wysłać informację do wszystkich wymienionych w polach informacyjnych „From:", „Signed-off-by:", „Acked-by:" (zwiększamy tym samym prawdopodobieństwo szybkiej odpowiedzi).

Czasami nasz raport może zostać zjedzony przez filtr anty-spamowy, ewentualnie ginie gdzieś w czeluściach skrzynek pocztowych deweloperów (niektórzy otrzymują bardzo dużo listów, więc nie należy się dziwić, że czasami coś przeoczą), dlatego jeżeli przez kilka dni nie doczekaliśmy się żadnej odpowiedzi, powinniśmy wysłać go ponownie. Gdyby mimo wszystko nikt nie odpowiedział na nasz raport, to najwłaściwszym postępowaniem będzie umieszczenie go w bugzilli `http://bugzilla.kernel.org/`.

Dobry raport powinien zawierać:

- zgłaszany błąd
- plik konfiguracyjny naszego jądra systemu
- informacje potrzebne do odtworzenia błędu
- informacje o konfiguracji naszego sprzętu (jeśli będą potrzebne, np. przy problemach z jakimiś sterownikami)

Więcej informacji na ten temat można znaleźć w pliku `REPORTING-BUGS`.

Dzięki włączeniu `CONFIG_DEBUG_INFO` będziemy mogli dostarczyć dodatkowe informacje pozwalające zlokalizować błąd. Podczas testowania powinniśmy w miarę możliwości (głównie chodzi o wydajność systemu, a raczej o jej brak po włączeniu niektórych opcji) włączyć kilka „ficzerów" w menu „`Kernel hacking --->`".

Po zgłoszeniu błędu najprawdopodobniej dostaniemy do przetestowania poprawkę. Jeżeli po jej zaaplikowaniu problem nie znika, to należy powiadomić o tym dewelopera, który nam ją wysłał. Jeżeli łatka naprawia problem, to pozostaje nam tylko podziękować deweloperowi.

`ORT` - jest to narzędzie, którego zadaniem jest ułatwienie przygotowania dobrego raportu błędu. Po uruchomieniu:

```
$ ./ort.sh oops.txt
```
Zostaniemy zapytani o typ raportu:

- short - tylko podstawowe informacje
- custom - możemy wybrać co chcemy umieścić w raporcie
- template - raport z szablonu

Następnie wpisujemy wymagane informacje, wybieramy potrzebne nam opcje. Raport jest generowany zgodnie z szablonem, który można znaleźć w pliku `REPORTING-BUGS`.
Najnowszą wersję `ORT` można znaleźć pod adresem: `http://www.stardust.webpages.pl/ltg/files/tools/ort/`

Wadą tego narzędzia jest to, że w prosty sposób możemy dostarczyć wiele niepotrzebnych informacji.

Rozdział 6

Testowanie sprzętu

Ważną sprawą, o której powinniśmy pamiętać przed przystąpieniem do wykonywania pierwszych testów jest sprawdzenie sprzętu - musimy mieć pewność, że działa poprawnie. Na dobrą sprawę powinno wystarczyć jednorazowe, porządne przetestowanie wszystkich komponentów naszego komputera. W późniejszym czasie, gdy pojawią się jakieś wątpliwości np. co do działania dysku twardego, możemy go sprawdzić ponownie (dyski twarde czasami się sypią).

Do wykonania testów pamięci może nam posłużyć program `Memtest86+`. Zalecam pobranie obrazu iso ze strony projektu `http://www.memtest.org/` i uruchomienie programu z płyty cd. Można się też posłużyć starszym programem `Memtest86` `http://www.memtest86.com/`, lub jakimkolwiek innym, dobrze spełniającym powyższe zadanie.

Skanowanie powierzchni dysku w poszukiwaniu obszarów uszkodzonych możemy przeprowadzić przy pomocy standardowego „`badblocks`". Wpisanie

```
# /sbin/badblocks -v /dev/dysk
```

powinno w miarę szybko dać nam odpowiedź, czy nasz dysk jest tak sprawny jakbyśmy tego chcieli. Dodatkowe informacje o stanie dysku można uzyskać wykonując test przy pomocy mechanizmu S.M.A.R.T.

```
# smartctl --test=long /dev/dysk
```

Rezultat testu możemy zobaczyć wydając polecenie

```
# smartctl -a /dev/dysk
```

Jeżeli podkręcamy procesor, pamięć etc. to na czas wykonywania testów musimy z tego zrezygnować. Podkręcanie sprzętu może wprowadzić przekłamania, oraz wywołać różne losowe błędy.

Warto sprawdzić czy napięcia zasilające podzespoły naszego komputera są prawidłowe, tutaj program `lm_sensors` przyjdzie nam z pomocą. Możemy też użyć programu dostarczonego przez producenta płyty głównej (niestety przeważnie wersja tylko dla Windows), często umożliwiającego wykonanie dodatkowej diagnostyki.

Dodatkowo powinniśmy pamiętać o występowaniu różnych losowych błędów związanych ze sprzętem, a niekoniecznie będących jego winą - np. wynikłych z promieniowania elektromagnetycznego, kosmicznego. Nie da się zabezpieczyć na 100% przed takimi błędami, jednak producenci sprzętu stosują różne technologie np. pamięć RAM z ECC, sprawdzanie sum kontrolnych przesyłanych bajtów na magistralach etc. mające za zadanie ograniczyć występowanie problemów związanych z przekłamaniami danych.

Kolejną kategorią błędów sprzętowych są wyjątki MCE (Machine Check Exception) generowane przez procesor, który powiadamia w ten sposób system np. o problemie z wewnętrzną pamięcią cache, niedopuszczalnym przegrzaniu etc. Po wykryciu takiego błędu system uznaje się za „skażony" i dodaje literkę „M" do linijki zawierającej informacje o EIP w oops'ie (`EIP: 0060:[<c046c7c3>] Tainted: PM VLI`).

Przy pomocy Linux-ready Firmware Developer Kit `http://www.linuxfirmwarekit.org/` możemy sprawdzić czy BIOS naszego komputera nie zawiera poważnych błędów. Wprawdzie nie będziemy mogli na to zbyt wiele poradzić - poza zaktualizowaniem do najnowszej wersji i ewentualnym powiadomieniem producenta sprzętu, jednak będziemy mieć świadomość występowania usterek. To może nam później pomóc w stwierdzeniu, czy znaleziony problem leży po stronie sprzętu, czy systemu operacyjnego.

Dodatek A

Dodatek A

A.1 Wysyłanie łatek

Chcesz wysłać łatkę poprawiającą znaleziony błąd?
Przeczytaj `Documentation/SubmittingPatches`, oraz zwróć szczególną uwagę na SECTION 3 - REFERENCES. Znajdują się tam odnośniki do dokumentów, z którymi należy się zapoznać przed wysłaniem łatki. Pamiętaj o wyłączeniu zawijania tekstu w kliencie poczty. Niektóre programy pocztowe np. Thunderbird lubią zmieniać białe znaki na inne białe znaki - zainstalowanie rozszerzenia do szyfrowania listów z reguły rozwiązuje takie problemy (jeśli chcesz używać Thunderbirda do wysyłania łatek, to polecam lekturę `http://mbligh.org/linuxdocs/Email/Clients/Thunderbird`).

Jeżeli chcesz wysłać całą serię łatek, to lepszym rozwiązaniem jest wykorzystanie skryptu `http://www.aepfle.de/scripts/42_send_patch_mail.sh` . Jeżeli nie chcesz się męczyć z rozgryzaniem wszystkiego, to zamieszczam krótką instrukcję:

- zainstaluj `sendmail` i `mutt`
- na początku skryptu wpisz „`sudo /etc/init.d/sendmail start`" a na końcu „`sudo /etc/init.d/sendmail stop`" - ograniczy to czas otwarcia potencjalnej dziury w systemie (sendmail ma bardzo długą historię luk).
- do pliku `.muttrc` wpisz „`set realname='Imię Nazwisko'`" i „`set from=jakiś_adres@email`"
- w 99 linijce skryptu zmień adres -bcc na swój
- zmień nazwę swojego hosta na nazwę bramy, wtedy nie będziesz otrzymywał listów zwrotnych

```
----- The following addresses had permanent fatal errors -----
<ktoś@gdzieś.com>
(reason: 553 5.1.8 <ktoś@gdzieś.com>... Domain of sender address
ktoś2@gdzieś2.com does not exist) [..]
```

Teraz trzeba utworzyć plik, który zostanie wysłany jako [PATCH 0/x], jego format jest następujący:

```
Subject: Jakiś temat
To: adres_jakiejś@listy.com
CC: adres_jakiegoś@odbiorcy1.com, adres_jakiegoś@odbiorcy2.com

Opis łatek, diffstat etc.
```

Również na początku każdej łatki dodajemy podobny nagłówek. Następnie tworzymy plik z serią łatek (taki sam jak do `quilt`) i już możemy wysłać wszystkie łatki:
`$ 42_send_patch_mail.sh -d plik_z_wiadomością.txt -s plik_z_serią_łatek`

Ciekawym skryptem umilającym wysyłanie łatek jest również `http://www.speakeasy.org/~pj99/sgi/sendpatchset` - jego niewątpliwą zaletą jest to, że nie wymaga instalacji programów sendmail i mutt. Wystarczy tylko stworzyć plik kontrolny:

```
SMTP: adres_serwera.smtp.z.którego@korzystamy.com
From: Imię i Nazwisko <nasz_adres@email>
To: Adresat1 <adres1@gdzieś.com>
Cc: Adresat2 <adres2@gdzieś.com>
Cc: Adresat3 <adres3@gdzieś.com>
Subject: [PATCH 1/n] Opis pierwszej łatki
File: ścieżka-do-łatki
Subject: [PATCH n/n] Opis ostatniej łatki
File: ścieżka-do-ostatniej-łatki
```

Następnie wysyłamy łatki:
`$ sendpatchset control`

A.2 System testowy

Nasz system testowy powinien być odseparowany od systemu, na którym normalnie pracujemy - nie powinniśmy w nim montować żadnych partycji z systemu stabilnego. Dzięki temu, w razie znalezienia jakiegoś poważnego błędu w systemie plików, lub innym podsystemie jądra, który wiąże się z utratą danych, nasz system stabilny nie powinien na tym ucierpieć. Jeżeli nowe jądro zniszczy nasz system testowy, to znaleźliśmy poważny błąd o którym powinniśmy powiadomić deweloperów systemu.

Kolejną rzeczą, o której warto jest pamiętać, to to, że możemy pracować na innym systemie z poziomu systemu stabilnego. Wystarczy zamontować partycję gdzieś w `/mnt`, a następnie za pomocą `chroot` przejść do drugiego systemu. Teraz możemy przygotować nowe jądro, skompilować i zainstalować go (najlepiej za pomocą jakiegoś skryptu, który za nas wszystko zrobi), nie odrywając się od normalnych zajęć. Gdy znajdziemy chwilę czasu, możemy zrestartować system i przejść do systemu testowego.

Przy wyborze dystrybucji służącej nam za system testowy, powinniśmy się kierować tylko i wyłącznie naszą wygodą - powinna nam ułatwić pracę, a nie przysparzać dodatkowej. Gdy

testujemy nowe wersje jądra, to chcielibyśmy też testować jego nową funkcjonalność, dlatego fajnie jest, gdy dystrybucja oferuje nowe wersje narzędzi i bibliotek. Dobrym wyborem jest testowa wersja Debiana - posiada aktualne wersje narzędzi, oraz starsze wersje `gcc` (niestety nie wszyscy deweloperzy sprawdzają czy ich kod działa po skompilowaniu inną wersją `gcc` niż ta dostarczana razem z ich ulubioną dystrybucją). Fedora Core posiada bardzo dobre wsparcie dla SELinuksa i wszystkich nowości, które oferują najnowsze wersje jądra.

A.3 KLive

KLive jest narzędziem, dzięki któremu deweloperzy Linuksa mogą się dowiedzieć jak długo dane drzewo było testowane. Dlaczego warto go używać? Linus Torvalds tak zaanonsował wydanie `2.6.15-rc5`

> „There's a rc5 out there now, largely because I'm going to be out of email contact for the next week, and while I wish people were religiously testing all the nightly snapshots, the fact is, you guys don't."

Deweloperzy nie są jasnowidzami i 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 - wtedy opiekun drzewa nie ma żadnej pewności, czy ktokolwiek testował dane wydanie. Jeżeli chcesz zacząć używać KLive, to na początku proponuje odwiedzić stronę projektu `http://klive.cpushare.com/` - można na niej znaleźć informacje o sposobie działania i instalacji programu (ta ostatnia jest bardzo prosta, wystarczy zainstalować wymagane pakiety, a następnie pobrać i uruchomić skrypt (`sh klive.sh --install`).

A.4 Jak zostać deweloperem Linuksa?

Czasami na LKML pojawia się to pytanie (ja na nie nie będę odpowiadał z prostego powodu nie jestem deweloperem), dlatego Greg KroahHartman napisał krótki dokument na ten temat. Jeżeli chcesz poznać odpowiedź na to pytanie, to po prostu przeczytaj `Documentation/HOWTO` :).

Dodatek B

Dodatek B

B.1 Jak pomóc w dalszym rozwoju podręcznika?

Wersja „źródłowa" podręcznika znajduje się pod adresem:
`http://www.stardust.webpages.pl/ltg/files/handbook-current.tar.bz2`
(przed przystąpieniem do modyfikacji warto sprawdzić, czy posiadamy najnowszą).

Po wprowadzeniu modyfikacji wystarczy zrobić

```
$ diff -uprN wersja-oryginalna/handbook.tex \\
 wersja-zmodyfikowana/handbook.tex > łatka.patch
```

i wysłać ją na adres `michal.k.k.piotrowski@gmail.com` (przy większych zmianach proszę też dodać CC na listę dyskusyjną LTG).

B.2 Gdzie uzyskać pomoc w testowaniu?

Jeśli potrzebujesz pomocy przy rozwiązaniu jakiegoś problemu związanego z jądrem Linux, masz wątpliwości czy znalazłeś(aś) błąd, to śmiało zadawaj pytania!

Strona pomocy w wiki `http://www.stardust.webpages.pl/ltg/wiki/index.php/Pomoc`

Strona listy dyskusyjnej `http://groups.google.com/group/linux-testers-group-pl`

B.3 Trochę o Linux Testers Group

Czyli odpowiedzi na wcześniej niezadane pytania.

P: Czym się zajmuje LTG?
O: Testowaniem jądra Linux.

P: Jak można dołączyć do LTG?
O: Wystarczy zacząć testować...

P: Gdzie jest jakaś strona domowa?
O: Pod adresem `http://www.stardust.webpages.pl/ltg/`

P: Jest jakaś lista dyskusyjna?
O: Jasne - pod adresem `http://groups.google.com/group/linux-testers-group-pl`

B.4 Przyczyny propagacji błędów do stabilnych wersji Linuksa

Poniższy artykuł został napisany dla Dragonia Magazine

Nie będzie to odkrywczym stwierdzeniem jak napiszę, że w Linuksie jest dużo błędów. Jednak skąd się one konkretnie biorą i jak można zapobiegać ich propagacji do stabilnych wersji systemu? Na to pytanie postaram się odpowiedzieć w poniższym artykule.

Na początku musimy sobie odpowiedzieć na proste pytanie - co to jest błąd w programie? Błąd w programie jest niezamierzoną (zdarzają się przypadki, że zamierzoną) pomyłką programisty, której skutkiem jest nieprawidłowe działanie programu. To nieprawidłowe działanie może się objawiać na kilka różnych sposobów:

- zawieszanie się programu
- podawanie niepoprawnych wyników obliczeń
- możliwość przejęcia kontroli nad programem przez włamywacza
- zafałszowywanie/uszkadzanie danych wejściowych i/lub wyjściowych
- niewydajne/nieefektywne algorytmy
- wycieki pamięci, sytuacje wyścigowe etc.

Błąd może wynikać z pomyłki podczas pisania kodu źródłowego programu, jak i na etapie planowania jego modelu. Zdarzają się również sytuacje, w których błąd powstaje za sprawą niepoprawnego działania narzędzi służących do kompilacji programu.

Błędy, które powstają na etapie pisania kodu źródłowego są często dużo łatwiejsze do wyeliminowania niż te, które powstają na etapie planowania modelu programu. Niewłaściwe decyzje podjęte w tej fazie mogą powodować występowanie bardzo trudnych do wyeliminowania błędów.

Jakie są główne przyczyny powstawania błędów w Linuksie? Zasadniczą przyczyną jest to, że jest to projekt duży i skomplikowany, dlatego większość ludzi pracujących nad systemem skupia się na swoich ulubionych podsystemach, które są im dobrze znane. Problem pojawia się wtedy, gdy ktoś próbuje zmodyfikować kod, którego dobrze nie zna i do końca nie rozumie. Dlatego też za sporą część błędów są odpowiedzialni ludzie, którzy Linuksem zajmują się dorywczo.

Kolejną przyczyną jest coś co się nazywa Nowy Model Rozwoju `http://www.stardust.webpages.pl/ltg/wiki/index.php/Proces_rozwoju_j%C4%85dra_Linux`. Dawniej Linux był rozwijany na zasadach podobnych do tych, na jakich jest rozwijane normalne oprogramowanie:

- wydanie wersji stabilnej i usuwanie błędów, które się do niej przedostały (np. wersje 2.2.x)

- cykl rozwojowy w którym jest dodawana nowa funkcjonalność, następnie następowała próba stabilizacji (np. wersje 2.3.x)
- kolejne wydania stabilne (np. wersje 2.4.x)

Jednak od wersji 2.6.8 model rozwoju uległ zmianie i teraz wygląda tak (na przykładzie wersji 2.6.20 i 2.6.21):

- wydanie wersji stabilnej 2.6.20, następnie są publikowane poprawki oznaczane jako 2.6.20.x
- praca nad nową wersją, dodawanie nowej funkcjonalności - 2.6.20-gitX (przeważnie około dwóch tygodni)
- zakończenie prac nad wersją rozwojową, rozpoczęcie cyklu stabilizacyjnego - 2.6.21-rcX
- wydanie stabilnej wersji 2.6.21

Taki przyspieszony model rozwoju ma swoje wady i zalety. Do wad można zaliczyć to, że postępuje on naprawdę bardzo szybko, więc dużo błędów może przejść z wersji rozwojowej do stabilnej. Do zalet powinniśmy zaliczyć to, że nie trzeba czekać na nową funkcjonalność przez długi okres czasu (dawne wersje niestabilne były rozwijane bardzo długo, a pierwsze wersje stabilne przeważnie trudno było za takie uznać).

Jaka jest przyczyna wprowadzania w Linuksie błędów związanych z bezpieczeństwem systemu? W doskonałej książce „*Secure Programming for Linux and Unix HOWTO*" `http://www.dwheeler.com/secure-programs/` David A. Wheeler wymienił kilka powodów `http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.html#WHY-WRITE-INSECURE` dla których w programach powstają błędy, które mogą zostać wykorzystane przez włamywaczy. Ze swojej strony dorzucę tylko jeden powód, który nie znalazł się na tej liście - mała ilość odpowiednich programów (lub ich mała skuteczność) ułatwiających wykrywanie potencjalnych luk.

Deweloperzy Linuksa przywiązują bardzo duże znaczenie do jakości kodu. Przed zgłoszeniem do przeglądu nowego kodu, deweloper powinien upewnić się, że spełnia on wszystkie wymogi wymienione w `Documentation/SubmitChecklist`. Następnie kod jest czytany przez innych deweloperów, którzy zgłaszają swoje uwagi - najczęściej odnośnie:

- stylu kodowania - styl kodowania w Linuksie powinien być jednolity - jest to bardzo ważna sprawa, ponieważ ułatwia czytanie i rozumienie kodu (można się z nim zapoznać czytając `Documentation/CodingStyle`)
- zastosowanych algorytmów - każdemu zależy na tym, aby były one jak najwydajniejsze
- poprawności i przejrzystości struktur danych - nie ma gorszej rzeczy, niż nieprzemyślane struktury danych

W zasadzie trudno jest określić, jaka konkretnie część błędów (lub potencjalnych błędów) jest eliminowana na etapie przeglądania kodu.

Kolejnym krokiem jest testowanie kodu w drzewie danego podsystemu lub -mm, gdzie jest

on sprawdzany pod względem współgrania z innymi częściami jądra systemu. Jest to bardzo ważny etap, ponieważ w nim powinny zostać wyłapane wszystkie poważniejsze błędy. Następnie kod jest włączany do rozwojowej wersji drzewa Linusa i tam następuje ostateczna próba jego stabilizacji.

Włączenie kodu do Linuksa nie jest takie proste - musi on spełniać narzucone standardy. Najczęściej następuje to za pośrednictwem tak zwanej „sieci zaufania" - kod jest włączany do drzewa danego podsystemu i dopiero z niego trafia do mainline. Przyczyną takiego stanu rzeczy jest to, że Linus Torvalds nie ma czasu czytać każdej wprowadzanej poprawki i wychodzi z założenia, że opiekunowie podsystemów wiedzą lepiej jak powinny one wyglądać oraz, że włączone do nich łatki zostały już odpowiednio przetestowane.

Proces dbania o jakość kodu w Linuksie jest bardzo rygorystyczny - dużo bardziej niż w innych projektach. Dlaczego więc do wersji stabilnych przedostaje się tak dużo błędów? Ostatnia poprawka „stable" `2.6.20.2` składa się z ponad stu łatek poprawiających konkretne problemy.

Odpowiedź jest bardzo prosta - niewystarczająca ilość osób poświęca czas na testowanie i poprawianie błędów w nowej funkcjonalności.

Niektórzy uważają, że remedium na te problemy powinien być powrót do starego modelu rozwoju, jednak ma on bardzo poważne wady:

- na nową funkcjonalność trzeba czekać bardzo długo (ostatni duży cykl rozwojowy - 2.5.x - trwał ponad dwa lata!)
- deweloperzy ponoszą dodatkowe koszty czasowe związane z przenoszeniem nowej funkcjonalności do stabilnych jąder dystrybucyjnych - ten czas mogliby poświęcić na jej tworzenie, poprawianie błędów etc.
- długi czas potrzebny na stabilizacje po długim cyklu rozwojowym

Jak widać nowy model rozwoju ma poważne zalety, jednak bez wystarczającej ilości ludzi testujących wydania rozwojowe nie będzie się sprawdzał. Sprawa jest bardzo skomplikowana, ponieważ deweloperzy nie mogą poprawiać błędów o których istnieniu nie wiedzą. Większość problemów wychodzi dopiero, gdy inni ludzie zaczynają używać danego kodu. Sytuacja może się pogorszyć do tego stopnia, że nowe stabilne wersje Linuksa będą działały dobrze tylko na maszynach o zbliżonej konfiguracji do tych, na jakich były testowane ich wersje rozwojowe.

Dobrym rozwiązaniem obok zwiększenia liczby osób testujących kod, może być zmniejszenie szybkości wprowadzania zmian. Jednak czy naprawdę tego chcemy? Z jednej strony zahamowanie rozwoju Linuksa może mieć bardzo pozytywny wpływ na jego stabilność, z drugiej strony trzeba się liczyć z tym, że wszystkie potrzebne nowe funkcje będą wchodzić do niego z coraz większym opóźnieniem.

Wszystkich zainteresowanych zmianą tego stanu rzeczy zapraszam do współpracy oraz dyskusji na naszej liście `http://groups.google.com/group/linux-testers-group-pl`.

B.5 Licencja