

Podręcznik testera jądra Linux

(wersja 0.1)

Copyright (C) 2006-2007 Michał Piotrowski <michal.k.k.piotrowski@gmail.com>

LTG – Linux Testers Group (<http://www.stardust.webpages.pl/ltg/>)

Współautorzy:

Copyright (C) 2006-2007 Maciej Rutecki <maciej.rutecki@gmail.com>

Unixy.pl (<http://www.unixy.pl/>)

Copyright (C) 2007 Rafał J. Wysocki <rjw@sisk.pl>

Podziękowania dla Mariusza Kozłowskiego za wyłapanie błędów i udzielenie cennych rad.

Uznanie autorstwa 2.5 Polska

Wolno:

- kopiować, rozpowszechniać, odtwarzać i wykonywać utwór
- tworzyć utwory zależne
- użytkować utwór w sposób komercyjny

Na następujących warunkach:

- Uznanie autorstwa. Utwór należy oznaczyć w sposób określony przez Twórcę lub Licencjodawcę
- W celu ponownego użycia utworu lub rozpowszechniania utworu należy wyjaśnić innym warunki licencji, na której udostępnia się utwór.
- Każdy z tych warunków może zostać uchylony, jeśli uzyska się zezwolenie właściciela praw autorskich.

Powyższe postanowienia w żaden sposób nie naruszają uprawnień wynikających z dozwolonego użytku ani żadnych innych praw.

(<http://creativecommons.org/licenses/by/2.5/pl/legalcode>)

Spis treści

Wprowadzenie.....	3
Rozdział 1 – Jądro, łątki, drzewa i kompilacja.....	5
Łatki.....	5
Ketchup.....	7
Drzewa.....	9
Drzewo -mm.....	9
Kompilacja, przygotowanie systemu testowego.....	10
Rozdział 2 - Testowanie.....	16
„Faza 1”.....	16
„Faza 2” (Autotest).....	16
„Faza 3”.....	19
Witaj świecie, czyli czego właściwie szukamy?.....	19
Sterowniki binarne - dlaczego nie możemy ich stosować podczas testowania.....	23
Rozdział 3 – Przechwytywanie błędów.....	24
Konsola szeregową.....	24
Rozdział 4 – Git, Quilt i przeszukiwanie binarne.....	26
GIT.....	26
Quilt.....	27
Przeszukiwanie binarne - idea.....	29
Przeszukiwanie binarne z pomocą quilt'a.....	29
Przeszukiwanie binarne z pomocą git-bisect.....	31
Rozdział 5 – Zgłaszanie błędów.....	33
Rozdział 6 – Testowanie sprzętu.....	34
Dodatek A.....	36
Wysyłanie latek.....	36
System testowy.....	37
KLive.....	37
Jak zostać deweloperem Linuksa?.....	38

Wprowadzenie

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

Wprowadzenie

„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/>). 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 w 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).

Rozdział 1 – Jądro, łatki, drzewa i kompilacja

Rozdział 1 – Jądro, łatki, drzewa i kompilacja

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. 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 – dla plików kompresowanych programem bzip2
```

```
tar xzvf linux-2.6.x.y.tar.gz – dla plików kompresowanych 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”.

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)

Rozdział 1 – Jądro, łatki, drzewa i kompilacja

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.

Więcej o aplikowaniu i ściąganiu łatek można przeczytać w pliku README oraz Documentation/applying-patches.txt znajdujących się w źródłach jądra.

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/>).

Rozdział 1 – Jądro, łatki, drzewa i kompilacja

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
```

Jeżeli używamy Debiana stable lub innej dystrybucji, w której nie ma ketchup'a, 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 zainstalowanego pythona.

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'
```

Rozdział 1 – Jądro, łatki, drzewa i kompilacja

```
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:

```
michal@ltg01-sid:~/tree$ ketchup -m
2.6.16.1
```

Jak widać ketchup w razie potrzeby pobiera potrzebne łatki, inne bierze z katalogu ~/.ketchup, więc dobrym pomysłem będzie wrzucenie tam łątek które aktualnie już posiadamy.

Przejsięcie 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
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 należy wykonać

```
wget -c `ketchup -u 2.6-mm | sed "s/.bz2/-broken-out.tar.bz2/"`
```


Rozdział 1 – Jądro, łatki, drzewa i kompilacja

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.

Drzewa

Często spotkamy się z pojęciem “drzewo jądra” (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 wchodzi do drzewa “surowego”, prowadzonego przez Linusa Torvaldsa.

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.

Rozdział 1 – Jądro, łatki, drzewa i kompilacja

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”.

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 nazywającą się “kernel”, następnie tworzymy katalog

Rozdział 1 – Jądro, łatki, drzewa i kompilacja

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_jadra>

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

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
- `modprobe -r isofs` – usunie moduł isofs

Rozdział 1 – Jądro, łatki, drzewa i kompilacja

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
```

Pozwala nam na "zapanowanie" nad systemem nawet w przypadku braku odpowiedzi ze strony komputera (pod warunkiem, że jądro systemu nadal funkcjonuje). Cóż nam to daje? 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	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

Rozdział 1 – Jądro, łatki, drzewa i kompilacja

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?

Platforma	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

Rozdział 1 – Jądro, łatki, drzewa i kompilacja

```
make install
```

Jeżeli używamy grub'a, 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 zamieszczam odsyłacze do instrukcji budowania jądra metodą konkretnych dystrybucji

CentOS http://www.howtoforge.com/kernel_compilation_centos

Debian <http://www.debian.org/doc/manuals/reference/ch-kernel.pl.html>

http://www.howtoforge.com/howto_linux_kernel_2.6_compile_debian

Fedora http://www.howtoforge.com/kernel_compilation_fedora

Gentoo <http://www.gentoo.org/doc/pl/kernel-upgrade.xml>

Mandriva http://www.howtoforge.com/kernel_compilation_mandriva

OpenSUSE http://www.howtoforge.com/kernel_compilation_suse

Ubuntu http://www.howtoforge.com/kernel_compilation_ubuntu

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

Rozdział 1 – Jądro, łatki, drzewa i kompilacja

zamkniętych – warto zapoznać się z zasadami nim rządzącymi

(http://www.stardust.webpages.pl/ltg/wiki/index.php/Proces_rozwoju_j%C4%85dra_Linux).

Krótkie wskazówki:

- 1) wpisz “make help” i przyjrzyj się dostępnym opcjom
- 2) w menu “Kernel hacking --->” można znaleźć wiele użytecznych opcji
- 3) zapoznaj się z opcjami skryptu ketchup

Rozdział 2 - Testowanie

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.

„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.

„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 wysledzić, która łątka 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

Rozdział 2 - Testowanie

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')
```

Rozdział 2 - Testowanie

```
system('./configure')
system('make')

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

Pierwszą funkcją zawartą w skrypcie jest funkcja „setup”, która ma za zadanie przygotować test do działania, czyli wykonuje za nas konfigurację i kompilację 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 równoległe kilka testów, to plik kontrolny powinien mieć składnię 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

Rozdział 2 - Testowanie

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.

„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 Oopsa :). 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 Oopsa 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? :).

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
```

Rozdział 2 - Testowanie

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
[<c02fab3>] syscall_call+0x7/0xb
```

Rozdział 2 - Testowanie

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)

```
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_contrack_netbios_ns ipt_REJECT xt_state
ip_contrack 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 :).

Rozdział 2 - Testowanie

```
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...

Rozdział 2 - Testowanie

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.

Sterowniki binarne - dlaczego nie możemy ich stosować podczas testowania

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 liniiki “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.

Rozdział 3 – Przechwytywanie błędów

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).

Konsola szeregową

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 konsolę szeregową w jądrze

```
Device Drivers --->
Character devices --->
Serial drivers --->
<*> 8250/16550 and compatible serial support
[*] Console on 8250/16550 and compatible serial port
```


Rozdział 3 – Przechwytywanie błędów

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 szeregową zawsze może się przydać. Jeśli jej nie potrzebujemy wystarczy, że zrobimy komentarz przed parametrami ją włączającymi.

```
kernel /boot/vmlinuz-2.6.20 ro root=LABEL=/ [...] #console=tty0 console=ttyS0,115200n8
```

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`

Krótkie wskazówki:

- 1) zapoznaj się z poleceniem `dmesg`
- 2) 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.

GIT

Git jest systemem kontroli wersji napisanym przez Linusa Torvaldsa na potrzeby jądra Linux (aktualnym opiekunem git'a 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 łątek? 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
```

Rozdział 4 – Git, Quilt i przeszukiwanie binarne

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'a 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

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://www.kernel.org/git/> oraz <http://git.infradead.org/> .

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

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

Rozdział 4 – Git, Quilt i przeszukiwanie binarne

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'a 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.

Rozdział 4 – Git, Quilt i przeszukiwanie binarne

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 $O(\log_2 N)$, więc w łatwy sposób możemy obliczyć (najlepiej za pomocą kalkulatora :) ile razy będziemy musieli przebudować system, aby znaleźć łatkę wprowadzającą błąd.

Przeszukiwanie binarne z pomocą quilt'a

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
```

Rozdział 4 – Git, Quilt i przeszukiwanie binarne

```
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 więc 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

Rozdział 4 – Git, Quilt i przeszukiwanie binarne

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'a

(http://www.youtube.com/watch?v=LS_hTnBDIYk)

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) oznaczmy 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 z wizualizować 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
```

Rozdział 4 – Git, Quilt i przeszukiwanie binarne

```
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 tą wersję jako dobrą:

```
git-bisect good
Bisecting: 55 revisions left to test after this
[a4657141091c2f975fa35aclad28fffd756091e] 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
```

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)

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 łątką 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 anti-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 łątką naprawia problem, to pozostaje nam tylko podziękować deweloperowi.

Rozdział 6 – Testowanie sprzętu

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 podkreślamy procesor, pamięć etc. to na czas wykonywania testów musimy z tego zrezygnować. Podkreślanie sprzętu może wprowadzić przekłamanie 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

Rozdział 6 – Testowanie sprzętu

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 Oopsie (EIP: 0060:[<046c7c3>] 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

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.

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 sendmaila i mutta
- 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"
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.

Dodatek A

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
```

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 SELinuxa i wszystkich nowości, które oferują najnowsze wersje jądra.

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

Dodatek A

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).

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 Kroah-Hartman napisał krótki dokument na ten temat. Jeżeli chcesz poznać odpowiedź na to pytanie, to po prostu przeczytaj Documentation/HOWTO :).