Odpluskwianie jądra Linuksa

Autorzy prezentacji

- Łukasz Kajda
  - funkcja printk
  - demony klogd i syslogd
  - polecenia strace i ltrace
- Arkadiusz Firus
  - kdb
  - kgdb
- Marcin Jałmużna
  - uml
- Krzysztof Dębski
  - pokaz praktyczny: gdb i uml

en.wikipedia.org/
wiki/Debugging

Co to jest debugowanie?

- Stwierdzenie istnienia błędu
- Wyłuskanie błędu w kodzie
- Znalezienie przyczyny błędu
- Naprawienie błędu
- Testowanie ze względu na znaleziony błąd oraz efekty uboczne

Kiedy należy używać debugera?

Lista kolejnych czynności przy szukaniu błędu:

- czytanie kodu ze zrozumieniem :)
- sprawdzanie kodu pod kątem popularnych potknięć składniowych
- wypisywanie na konsolę pomocnych komunikatów lub wartości zmiennych
- sięgnięcie po debuger

en.wikipedia.org/wiki/Debugger

Podstawowe możliwości debugera

- **single-stepping**, czyli wykonywanie kodu krok po kroku
- **break-points**, czyli zatrzymywanie programu do testowania jego stanu (ustawianie tzw. pułapek)
- **watch-points**, czyli zatrzymywanie programu w razie zmiany jego stanu
- modyfikowanie stanu programu w czasie jego działania

Debugowanie jądra Linuksa...

... czyli co dalej:

- funkcja `printk()`
- demony klogd i syslogd
- oops
- `strace` i `ltrace`
- gdb oraz kgdb
- UML (User-Mode Linux)

Funkcja `printk()`

- Składnia funkcji
- Znaczenie funkcji
- Przykład użycia:

```
if (sscanf(operation, "%ld %c %ld", &args[0], &op, &args[1]) != 3)
    return -EINVAL;                    /* Niepoprawne wyrazenie */
printk(KERN_DEBUG "calc: Skladnia wyrazenia poprawna...\n");
```

Funkcja `printk()`

- Piorytety komunikatów:

<0>	KERN_EMERG	sytuacje awaryjne
<1>	KERN_ALERT	błędy alarmowe
<2>	KERN_CRIT	błędy krytyczne
<3>	KERN_ERR	błędy systemu
<4>	KERN_WARNING	ostrzeżenia
<5>	KERN_NOTICE	istotne notatki
<6>	KERN_INFO	informacje
<7>	KERN_DEBUG	rozwijanie jądra

Funkcja `printk()`

- Jak są przekazywane informacje z użycia funkcji?
- Dokąd można wypisywać te infromacje?

Funkcja `printk()`

Zalety:

- uniwersalność
- łatwość użycia

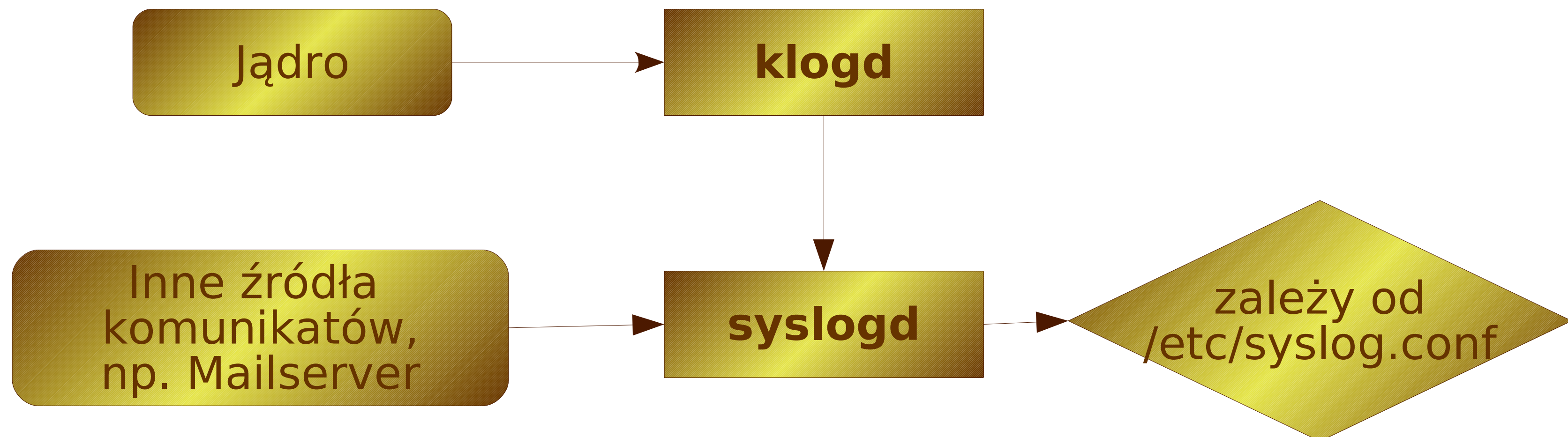
Wady:

- możliwość przeciążenia systemu zbyt dużą ilością komunikatów
- brak sensu wywołania przed inicjacją konsoli (`early_printk()`)

www.linux-magazine.pl/
issue/02/Sysadmin_syslog.pdf

Demony klogd oraz syslogd

- Logi jądra do klogd (kernel log daemon)
- Obsługa zdarzeń systemowych przez syslogd



`/etc/syslog.conf`

Demony klogd oraz syslogd

Zawartość i modyfikacja syslog.conf:

Konwencja zapisu: źródło.rodzaj wyjście

Źródła komunikatów:

- **auth** – komunikaty związane z autoryzacją
- **authpriv** – inne komunikaty związane z autoryzacją
- **ftp** – komunikaty z serwera ftp
- **mail** – komunikaty związane z pocztą
- **news** – komunikaty podsystemu newsów
- **syslog** – komunikaty demona syslog
- **cron** – komunikaty crona
- **kern** – komunikaty jądra systemu
- **lpr** – komunikaty drukarki
- **user** – procesy użytkowników
- **daemon** – demony systemowe
- **mark** – w regularnych odstępach czasu wysyła datę

www.linux-tutorial.info/modules.php?name=Tutorial&pageid=57

Demony klogd oraz syslogd

Zawartość i modyfikacja syslog.conf:

Rodzaje komunikatów:

- **none** – po prostu nic
- **info** – komunikaty informacyjne
- **alert** – komunikaty alarmujące, wymagające natychmiastowego działania
- **crit** – komunikaty krytyczne
- **debug** – komunikaty przydatne do wyszukiwania błędów
- **emerg** – system jest zablokowany
- **err** – komunikaty błędów
- **notice** – typowe zdarzenia
- **warning** – ostrzeżenia

Dodatkowo:

- **!** – za wyjątkiem
- ***** – wszystko

en.wikipedia.org/wiki/Linux_kernel_oops

Oops

- Oops pojawia się w razie błędów i nieprzewidzianych sytuacji
- Dwie funkcje oops: informacja i zapobieganie

`man ksymoops / man objdump`

Oops

- Ksymoops – czyni oops czytelnym
  - Instalacja i użycie ksymoops
- Szukanie w kodzie błędu (użycie `objdump`)

Oops

www.linuxhq.com/lnxlists/linux-kernel/lk_9906_02/msg00338.html

Przykład oops z użyciem ksymoops:

```
Unable to handle kernel paging request at virtual address 4024af04
current->tss.cr3 = 00101000, %cr3 = 00101000
*pde = 00000000
Oops: 0002
CPU:    0
EIP:    0010:[remove_inode_page+27/116]
EFLAGS: 00010282
eax: c028adc8   ebx: c02ed2fc   ecx: 4024af04   edx: c02ed2fc
esi: 000000fa   edi: 00000030   ebp: c000c000   esp: c000dfb8
ds: 0018   es: 0018   ss: 0018
Process kswapd (pid: 4, process nr: 4, stackpage=c000d000)
Stack: 00000005 00000006 c01227ba 00000006 00000030 00000000 c01c0f6e  c000c1cd
       c012285e 00000030 00000f00 c0003fa8 c0106000 000000a0 c01089af  00000000
       00000f00 c0205fd8
Call Trace: [do_try_to_free_pages+38/120] [tvecs+10478/19552] [kswapd+82/200]
[get_options+0/112] [kernel_thread+35/48]
Code: 89 01 c7 42 2c 00 00 00 00 ff 0d 60 25 1f c0 8b 4a 08 c7 42

Code:  00000000 Before first symbol           00000000 <_IP>: <===
Code:  00000000 Before first symbol              0:    89 01
movl   %eax,(%ecx) <===
Code:  00000002 Before first symbol              2:    c7 42 2c 00 00 00  00
movl   $0x0,0x2c(%edx)
Code:  00000009 Before first symbol              9:    ff 0d 60 25 1f c0
decl   0xc01f2560
Code:  0000000f Before first symbol              f:    8b 4a 08
movl   0x8(%edx),%ecx
Code:  00000012 Before first symbol             12:    c7 42 00 00 00 00  00
movl   $0x0,0x0(%edx)
```

Linux/include/asm-generic/bug.h

Oops

Generowanie oops w kodzie:
- `BUG()`
- `BUG_ON()`
- `panic()`

```
#define BUG() do { \
   printk("kernel BUG at %s:%d!\n", __FILE__, __LINE__); \
   panic("BUG!"); \
}

#define BUG_ON(condition) do { if
(unlikely((condition)!=0)) BUG(); }
```

`man ltrace / man strace`

Polecenia strace, oraz ltrace

Znaczenie poleceń:
- s – system
- l – library

Parametry wywołania:
- `-r` – wypisuje odległości czasowe między kolejnymi udanymi wywołaniami systemowymi
- `-t` – poprzedza każdą linię czasem wywołania
- `-o` – przekierowanie komunikatów z stderr do danego pliku
- `-p` – śledzenie procesu o podanym pidzie

krzysztof.gibas.org/
text/strace.html

Polecenia `strace` oraz `ltrace`

Przykład użycia `strace` z parametrem `-t`:

```
19:51:36 clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|
CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x4014bae8) = 6381
19:51:36 close(7)                        = 0
19:51:36 close(3)                        = 0
19:51:36 close(4)                        = 0
19:51:36 close(5)                        = 0
19:51:36 read(0, "qwerty\n", 4096)       = 7
19:51:40 write(1, "qwerty 0\0", 9)       = 9
19:51:40 close(3)                        = -1 EBADF (Bad file
descriptor)
19:51:40 write(2, "*3* ", 4)             = 4
19:51:40 --- SIGCHLD (Child exited) @ 0 (0)
```

Built-in Kernel Debuger (kdb)

Możliwości:

- Wykonywanie instrukcji “krok po kroku”.
- Zatrzymanie na konkretnej instrukcji.
- Zatrzymanie przy modyfikacji adresu pamięci.
- Zatrzymanie przy dostępnie do rejestru (we/wy).
- Śledzenie zmian na stosie dowolnego procesu.
- Deasemblowanie instrukcji.

Instalacja:

- Wgranie patcha do jądra.
- Włączenie kdb w konfiguracji jądra.

Konfiguracja:

Kdb może wykonać pewien zestaw instrukcji w trakcie uruchamiania. Owe instrukcje muszą być zapisane w pliku kdb_cmds, znajdującym sie w katalogu KDB, który jest podkatalogiem źródeł jądra.

Uruchomienie:

Zależnie od opcji wybranej w konfiguracji:

- Kdb startuje wraz z systemem.
- Uruchamiamy ręcznie:
  - podając jako parametr jądra kdb=on
  - wywołując polecenie:

```
echo "1" >/proc/sys/kernel/kdb
```

Podstawowe komendy operujące na pamięci:

- `md 0xc000000 15` //wypisuje zawartość pamięci począwszy od adresu 0xc000000 do 0xc00000e
- `mm 0xc000000 0x10` // zapisuje 16 pod adresem 0xc000000

Podstawowe komendy operujące na rejestrach:

- `rd` //wypisuje zawartość wszystkich rejestrów
- `rm %ebx 0x2a` //zapisuje do rejestru ebx wartość 42

Wstrzymywanie wykonania kodu:

- `bp sys_write` //po natrafieniu na instrukcję sys_write jądro zatrzymuje się i przekazuje sterowanie kdb.
- `bl` //wypisuje listę wszystkich wszystkich instrukcji po których nastąpi wstrzymanie.
- `bc 1` //usuwa z powyższej listy wpis pod indeksem pierwszym.

Wypisywanie zawartości stosu:

- `bt` //wypisuje stos aktualnego procesu
- `btp 42` //wypisuje stos procesu o identyfikatorze równym 42
- `bta` //wypisuje stosy wszystkich procesów

Pozostałe komendy:

- `ss` //wykonuje jedną instrukcję i zwraca sterowanie do kdb
- `go` //wykonuje instrukcje aż do najbliższego punktu wstrzymania.
- `reboot` //robi to samo co ”twardy” reset

Strona domowa:

http://oss.sgi.com/projects/kdb/

User Mode Linux (UML)

Instalacja

Ściągamy gotowe jądro(5MB)
i system plików(50MB, po rozpakowaniu 1,5GB)

http://uml.nagafix.co.uk/

Uruchamiamy:

```
$ ./linux ubda=system_plików
```

Alternatywa dla '`make xconfig ARCH=um`' to:
'`make config ARCH=um`' i '`make menuconfig ARCH=um`'

Instalacja z rpm

Ściągamy pakiet rpm z http://user-mode-linux.sourceforge.net i jakiś system plików

```
$ rpm -i pakiet.rpm
```

Dzieki temu mamy jądro, narzędzia i dokumentacje

Instalacja modułów

Moduł powinien być skompilowany w tym samym katalogu, co jądro

```
$ make moduł ARCH=um
```

Jest kilka możliwości instalacji modułów:

1. Przenosimy moduł na wirtualną maszyne i umieszczamy w `/lib/modules/`uname -r``

2.
 - Montujemy system plików u hosta

```
$ mount root_fs mnt -o loop
```

- uruchamiamy

```
$ make modules_install
INSTALL_MOD_PATH=`pwd`/mnt ARCH=um
```

- odmontowujemy

```
$ umount mnt
```

Pakiet uml_utilities

Zawiera narzędzia:

- port-helper – używany przez konsole do połączenia z xterm lub portami
- tunctl – konfiguracja, tworzenie i usuwanie urządzeń tap
- uml_net – automatyczna konfiguracja urządzeń tap (sieć)
- uml_switch – do przenoszenia demonów
- uml_mconsole – dodawanie sprzętu...
- uml_moo – obsługa plików cow
- uml_mkcow – do tworzenia plików cow

Kompilacja:

```
$ make && make install
```

Kanały i konsole

Możemy przyporządkować UML-a do wejścia/wyjścia hosta.
Możemy ustawiać konsole (con) i porty szeregowe (ssl)
con0=fd:0, fd:1

kanały:
- pseudo-terminale `pts, pty`
- terminal hosta `tty:terminal`
- xterm `xterm`
- deskryptor pliku `fd:nr`
- port `port:nr_portu`

Tworzenie systemu plików

Tworzymy pusty plik:

```
$ dd if=/dev/zero of=new_filesystem
seek=100 count=1 bs=1M
```

Uruchamiamy uml-a z dodatkowym parametrem (niekoniecznie ubd4)

```
ubd4=new_filesystem
```

Tworzymy system plików:

```
$ mkreiserfs /dev/ubd/4
```

Montujemy:

```
$ mount /dev/ubd/4 /mnt
```

Pilik COW (copy-on-write)

Pozwalają “współdzielić” system plików

Tworzenie pliku cow

```
$ ./linux ubd0=nowy_cow,system_plików
```

Bez uruchamiania uml-a

```
$ uml_mkcow [-f] plik_cow system_plików
```

Uruchamianie uml-a

```
$ ./linux ubd0=nowy_cow
```

Scalanie pliku cow z systemem plików

```
$ uml_moo plik_cow nowy_system_plików
```

Scalanie bez tworzenia nowego pliku

```
$ uml_moo -d plik_cow
```

Konsola zarządzania – MCONSOLE

Nadanie identyfikatora:

```
$ ./linux ubda=system_plikow umid=nasze_id
```

Uruchamiamy konsolę:

```
$  uml_mconsole nasze_id
```

Możliwe komendy:

* version
* halt
* reboot
* config
* remove
* sysrq
* help
* cad
* stop
* go
* log
* proc

MCONSOLE cd.

version – zwraca informacje o systemie
OK Linux usermode 2.4.5-9um #1 Wed Jun 20 22:47:08 EDT 2001 i686

halt i reboot – zamknięcie systemu

config – dodaje nowe urządzenie

```
config ubd3=root_fs_debian22
```

remove – usuwa urządzenie

help – pomoc:)

cad – efekt jak przy wcisnięciu Alt+Ctrl+Delete

MCONSOLE cd.

stop, go – wykonuje tylko polecenia z mconsole, aż dostanie go

log – treść argumentu zapisuje do message log-a (wykonuje printk)

proc – wypisuje zawartosc podanego jako argument pliku z katalogu proc

sysrq – informacje w Documentation/sysrq.txt

Dostęp do sieci (TUN/TAP)

W konfiguracji dostępu do sieci pomaga uml_net

Uruchamiamy UML-a z parametrem

```
eth0 =tuntap,,,adres_hosta
```

Konfigurujemy UML-a

```
$ ifconfig eth0 adres_umla up
```

Linki

http://user-mode-linux.sourceforge.net
http://en.wikipedia.org/wiki/User-mode_Linux
http://www.kernel.org/pub/linux/kernel/
http://uml.nagafix.co.uk/

Jak debugować przez sieć, czyli krótko o kgdb

Po co debugować przez sieć?

Zalety:

- Wykrywamy błędy działania kodu na “żywym” komputerze.
- Nie tracimy stabilności i bezpieczeństwa systemu macierzystego.

Wady:

- Konieczność posiadania dwóch komputerów połączonych ze sobą.

Zrób to sam

- Wgraj patcha do jądra.
- Skompiluj jądro włączając opcję kgdb w “Kernel hacking”.
- Wgraj jądro na komputer przeznaczony do testów.
- Dodaj “kgdboe=@LOCAL-IP/,@REMOTE-IP/” jako parametr uruchomienia jądra.
- Uruchom gdb na głównym komputerze (gdb ./vmlinux).
- Wywołaj polecenie “target remote udp:HOSTNAME:6443”

Problem:

Najnowsza wersja jądra wspierana przez kgdb to 2.6.15.5 :(

Podsumowanie:

Oficjalna strona kgdb:

http://kgdb.linsyssoft.com/

Pokaz praktyczny:
UML i GDB

Co potrzebujemy

- Jądro linuxa (kernel.org)
- Obraz systemu plików, np. Damn Small Linux (uml.nagafix.co.uk)
- gdb (gnu.org/software/gdb)

Kompilacja jądra

- `tar -xjvf linux-2.6.17.13.tar.bz2`
- `make defconfig ARCH=um`
- `make xconfig ARCH=um`
- `make linux ARCH=um`

Przykład ustawień w xconfigu

- [x] Tracing thread support
  - (1) Kernel address space size (in .5G units)
  - [x] Separate Kernel Address Space support
- [ ] 2G/2G host address space split
- [ ] Three-level pagetables (EXPERIMENTAL)
- Memory model
  - (•) Flat Memory
- [x] Networking support
- [x] Kernel support for ELF binaries
- [•] Kernel support for MISC binaries
- [x] Host filesystem
- [x] HoneyPot ProcFS (EXPERIMENTAL)
- [x] Management console
  - [ ] Magic SysRq key
- [ ] Symmetric multi-processing support (EXPERIMENTAL)
- (0) Nesting level
- [ ] Highmem support (EXPERIMENTAL)
- (2) Kernel stack size order
- [x] Real-time Clock

UWAGA!

Wykorzystane przeze mnie jądro zostało przed kompilacją nieznacznie zmienione.

W pliku kernel/signal.c dodałem następujące polecenie do funkcji “kill_something_info”:

“if (sig == SIGUNUSED) (*(int*)NULL)=0;”

Gdy system spróbuje przesłać do jakiegoś procesu sygnał SIGUNUSED(numer 31) wykona błędną operację i nastąpi oops.

Uruchomienie UML

Jeśli obraz plików ma nazwę “root_fs”, to wystarczy wpisać “`./linux`”, wpp. trzeba wpisać

“`./linux ubda=system_plików`”

Gdy system się włączy wpisujemy polecenie

“`kill -31 0`”

(wysłanie sygnału 31 do procesu nr 0, normalnie wszystkie sygnały wysłane do inita są ignorowane)

Zgodnie z oczekiwaniami system wyświetla błąd “Kernel Panic”

GDB

W celu debugowania jądra wpisujemy:

```
gdb linux
```

Jądro UML komunikuje się z jądrem hosta poprzez dwa sygnały(USR1, SEGV). Ponieważ jądro UML zostało uruchomione w gdb, więc linux nie może wysyłać komunikatów bezpośrednio do jądra UML.

Można jednak nakazać GDB pośredniczenie w wymianie sygnałów. W tym celu wpisujemy w GDB:

```
handle SIGSEGV pass nostop noprint

handle SIGUSR1 pass nostop noprint
```

Następnie uruchamiamy UML: `r [parametry UML]`

Breakpointy w GDB

W celu ustawienia breakpointa w GDB wpisujemy:

`b nazwa funkcji (np. b start_kernel)`

Jeżeli UML jest już uruchomiony, to terminal będzie kontrolowany przez UML i dlatego nie będziemy mogli wykonać żadnego polecenia w GDB.

Aby zatrzymać na chwilę UML i przekazać terminal GDB należy:

- Uruchomić drugi terminal
- Wpisać w nim "`ps ux|grep linux`"
- Wpisać "`kill -INT pid_linux`", gdzie pid_linux = pid pierwszego wypisanego procesu bezpośrednio po "gdb linux"
- W terminalu z GDB wpisać `b nazwa funkcji`
- Nakazać jądru UML kontynuowanie: `c`

Przykład Breakpointu

(jakiś inny terminal)

```
(BASH)$ps ux|grep "linux"
krzysiek 15773  0.9 11.4  32036 29432 pts/0    S    20:58   0:06 gdb linux
krzysiek 15791  0.4  9.6  32868 24840 pts/0    S+   20:58   0:02 /home/krzysiek/Desktop/uml/linux
krzysiek 15802  0.0  9.6  32868 24840 pts/0    S+   20:58   0:00 /home/krzysiek/Desktop/uml/linux
krzysiek 15803  0.0  9.6  32868 24840 pts/0    S+   20:58   0:00 /home/krzysiek/Desktop/uml/linux
krzysiek 15809  0.0  9.6  32868 24840 pts/0    S+   20:58   0:00 /home/krzysiek/Desktop/uml/linux
krzysiek 15810  0.0  0.0     88    72 pts/0    T+   20:58   0:00 [linux]
krzysiek 16109  0.0  0.5   1416  1416 pts/0    T+   20:59   0:00 [linux]
krzysiek 16602  0.0  0.3   2888   812 pts/2    S+   21:08   0:00 grep linux
(BASH)$kill -INT 15791
```

(terminal GDB/UML)

```
Program received signal SIGINT, Interrupt.
(GDB)$ b sys_kill
Breakpoint 1 at 0xa00364e8: file kernel/signal.c, line 2150.
(GDB)$ c
```

Debugowanie jądra

Ustawiliśmy breakpoint na funkcji “sys_kill”, która jest odpowiedzialna za przekazywanie sygnałów. Teraz znów spróbujemy doprowadzić do Oops. Wpisujemy:

```
(UML)$kill -31 0
```

```
Breakpoint 1, sys_kill (pid=0, sig=0) at kernel/signal.c:2150
```

Możemy obserwować kolejne instrukcje systemu za pomocą polecenia `s` (step)

```
(GDB)$s
```

```
2154            info.si_errno = 0;
```

Po kilku krokach zostaje wywołana funkcja.

```
kill_something_info (sig=31, info=0xa0ab3bd4, pid=0) at kernel/signal.c:1151
```

```
(GDB)$s
```

```
if (sig == SIGUNUSED) (*(int*)NULL)=0;
```

```
(GDB)$s
```

```
Kernel panic - not syncing: Kernel mode fault at addr 0x0, ip 0x400816c1
```

Debugowanie cd.

Możemy oglądać kolejne kroki za pomocą `s` lub wpisać `c`, aby kontynuować.

```
(GDB)$c

Program exited with code 01.
```

Teraz już wiemy, w którym miejscu dokładnie wystąpił Oops, dzięki temu możemy wrócić do kodu i naprawić błąd.

Instalacja modułów jadra:

- `make modules ARCH=um`
- `sudo mount root_fs /mnt -o loop`
- `sudo make modules_install INSTALL_MOD_PATH=/mnt ARCH=um`
- `sudo umount /mnt`
- `make modules_install INSTALL_MOD_PATH=mods ARCH=um`
- `depmod -a (w UML)`

root_fs-obraz systemu plików

mods-dowolna ścieżka na hoscie(jednak lepiej nie używać /lib/modules), zostanie tam zainstalowana kopia modułów, gdyż jest to wymagane do debugowania

Debugowanie modułów

Będę debugował moduł “loop”. Dodaję go do jądra

```
(UML)$modprobe loop
```

Zatrzymujemy działanie UML(tak jak opisane wcześniej).

Następnie:

```
(GDB)$p modules
$9 = {next = 0xa2829e04, prev = 0xa2829e04}
(GDB)$p *((struct module *)0xa2829e00)
```

Zamiast liczby 0x2829e00 wpisujemy, to co zwrócił “p modules”(w polu “next”) minus 4 (dla architektury 32bit) lub 8 (dla architektury 64bit). Interesuje nas module_core.

```
(...) module_core = 0xa2827000 (....)
(GDB)$add-symbol-file adres_modułu_na_hoscie module_core
(GDB)$~/linux-2.6.17.13/mods/lib/modules/2.6.17.13/kernel/drivers/block/loop.ko 0xa2827000
```

Debugowanie modułów cd.

```
(y or n)
(GDB)$y
Reading symbols from /home/krzysiek/mods/lib/modules/2.6.17.13/kernel/drivers/block/loop.ko...done.
(GDB)$b lo_open
```

(lo_open jest funkcją modułu loop)

```
(GDB)$c
(UML)$cat /dev/loop0
```

(Zostaje wywołana funkcja lo_open)

```
Breakpoint 2, lo_open (inode=0xa28287a4, file=0xa13cc3ec
```

Następnie debugujemy, tak jak jądro.

Kończenie pracy z UML

Zatrzymujemy system z UML:

```
(UML)$halt
```

```
Breakpoint 1, sys_kill (pid=-776, sig=0).
```

Podczas zamykania systemu wysyłanych jest dużo sygnałów. Jeżeli nie chcemy oglądać każdego breakpointa, używamy instrukcji “c n”, która powoduje, że następne n - 1 tego typu breakpointów zostanie zignorowane.

```
(GDB)$c 1000
```

```
Will ignore next 999 crossings of breakpoint 1.  Continuing.

(...)

Program exited normally.
```

Kończymy pracę z GDB:

```
(GDB)$q
```