Debugowanie jądra linuxa

Agata Chrobak Grzegorz Paszt Tomasz Witaszczyk

MIMUW

21 listopad 2007

O czym będzie mowa...

1. Po co debugować jądro?
   - Co to jest debugowanie?
   - Dlaczego chcemy debugować jądro?
2. Jak debugować jądro?
   - Oops
   - Printk()
   - Strace i ltrace
3. Jak inaczej debugować jądro?
   - KDB
   - GDB
   - KGDB
   - UML

Co to jest debugownie?

- **Debugowanie** (z ang. debugging) – proces systematycznego redukowania liczby błędów w oprogramowaniu bądź systemie mikroprocesorowym, który zazwyczaj polega na kontrolowanym wykonaniu programu pod nadzorem debugera.

Co to jest debugownie?

- **Debugowanie** (z ang. debugging) – proces systematycznego redukowania liczby błędów w oprogramowaniu bądź systemie mikroprocesorowym, który zazwyczaj polega na kontrolowanym wykonaniu programu pod nadzorem debugera.
- **Debuger** (ang. debugger - odpluskwiacz) – program komputerowy służący do analizy dynamicznej programów w celu znalezienia w nich błędów (zwanych również z ang. bugami).

Dlaczego chcemy debugować jądro?

- Dlaczego chcemy debugować jądro, skoro jest częścią systemu operacyjnego, który przecież nigdy nas nie zawodzi... Co, gdy jednak nas zawiedzie?

Dlaczego chcemy debugować jądro?

- Dlaczego chcemy debugować jądro, skoro jest częścią systemu operacyjnego, który przecież nigdy nas nie zawodzi... Co, gdy jednak nas zawiedzie?
- Trzeba naprawić.

Dlaczego chcemy debugować jądro?

Debugujemy jądro ponieważ...

- Chcemy dowiedzieć się, dlaczego nie działa sterownik.

Dlaczego chcemy debugować jądro?

Debugujemy jądro ponieważ...

- Chcemy dowiedzieć się, dlaczego nie działa sterownik.
- Chcemy dowiedzieć się, czemu jądro się załamuje i wypisuje optymistyczny komunikat "kernel panic".

Dlaczego chcemy debugować jądro?

Debugujemy jądro ponieważ...

- Chcemy dowiedzieć się, dlaczego nie działa sterownik.
- Chcemy dowiedzieć się, czemu jądro się załamuje i wypisuje optymistyczny komunikat "kernel panic".
- Chcemy się dowiedzieć, czemu komputer się zawiesza.

Dlaczego chcemy debugować jądro?

Debugujemy jądro ponieważ...

- Chcemy dowiedzieć się, dlaczego nie działa sterownik.
- Chcemy dowiedzieć się, czemu jądro się załamuje i wypisuje optymistyczny komunikat "kernel panic".
- Chcemy się dowiedzieć, czemu komputer się zawiesza.
- Mamy prezentację albo projekt z so i musimy nauczyć się debuggować jądro.

Dlaczego chcemy debugować jądro?

Debugujemy jądro ponieważ...

- Chcemy dowiedzieć się, dlaczego nie działa sterownik.
- Chcemy dowiedzieć się, czemu jądro się załamuje i wypisuje optymistyczny komunikat "kernel panic".
- Chcemy się dowiedzieć, czemu komputer się zawiesza.
- Mamy prezentację albo projekt z so i musimy nauczyć się debuggować jądro.
- Chcemy poszpanować przed koleżankami.

Odpluskwianie jądra

Dlaczego odpluskwianie jądra nie jest takie proste?

- Przede wszystkim, dlatego że jądro nie działa w żadnym zewnętrznym środowisku.

Odpluskwianie jądra

Dlaczego odpluskwianie jądra nie jest takie proste?

- Przede wszystkim, dlatego że jądro nie działa w żadnym zewnętrznym środowisku.
- Przebieg działania jest zależny od wielu czynników i ciężko jest nam powtórzyć błąd, który zauważyliśmy.

Odpluskwianie jądra

Dlaczego odpluskwianie jądra nie jest takie proste?

- Przede wszystkim, dlatego że jądro nie działa w żadnym zewnętrznym środowisku.
- Przebieg działania jest zależny od wielu czynników i ciężko jest nam powtórzyć błąd, który zauważyliśmy.
- Nie zawsze po zaistniałym błędzie zostają ślady (np. w postaci plików).

Jakie mamy narzędzia do dyspozycji?

Jest wiele, ale niekoniecznie wygodnych...

- oops

Jakie mamy narzędzia do dyspozycji?

Jest wiele, ale niekoniecznie wygodnych...

- oops
- printk()

Jakie mamy narzędzia do dyspozycji?

Jest wiele, ale niekoniecznie wygodnych...

- oops
- printk()
- strace i ltrace

Jakie mamy narzędzia do dyspozycji?

Jest wiele, ale niekoniecznie wygodnych...

- oops
- printk()
- strace i ltrace
- KDB - built-in Kernel DeBugger for Linux

Jakie mamy narzędzia do dyspozycji?

Jest wiele, ale niekoniecznie wygodnych...

- oops
- printk()
- strace i ltrace
- KDB - built-in Kernel DeBugger for Linux
- GDB - GNU DeBugger

Jakie mamy narzędzia do dyspozycji?

Jest wiele, ale niekoniecznie wygodnych...

- oops
- printk()
- strace i ltrace
- KDB - built-in Kernel DeBugger for Linux
- GDB - GNU DeBugger
- KGDB - Kernel GNU DeBugger

Jakie mamy narzędzia do dyspozycji?

Jest wiele, ale niekoniecznie wygodnych...

- oops
- printk()
- strace i ltrace
- KDB - built-in Kernel DeBugger for Linux
- GDB - GNU DeBugger
- KGDB - Kernel GNU DeBugger
- UML - User Mode Linux

Czym jest oops?

Komunikat oops pojawia się w momencie wystąpienia błędu lub nieprzewidzianej sytuacji. Informuje o błędzie i inicjuje zakończenie pracy systemu, zapobiegając dalszemu niepoprawnemu działaniu.

W krytycznym momencie wypisuje na ekran:

- numer oopsa

Czym jest oops?

Komunikat oops pojawia się w momencie wystąpienia błędu lub nieprzewidzianej sytuacji. Informuje o błędzie i inicjuje zakończenie pracy systemu, zapobiegając dalszemu niepoprawnemu działaniu.

W krytycznym momencie wypisuje na ekran:

- numer oopsa
- status programu

Czym jest oops?

Komunikat oops pojawia się w momencie wystąpienia błędu lub nieprzewidzianej sytuacji. Informuje o błędzie i inicjuje zakończenie pracy systemu, zapobiegając dalszemu niepoprawnemu działaniu.

W krytycznym momencie wypisuje na ekran:

- numer oopsa
- status programu
- zawartość rejestrów procesora

Czym jest oops?

Komunikat oops pojawia się w momencie wystąpienia błędu lub nieprzewidzianej sytuacji. Informuje o błędzie i inicjuje zakończenie pracy systemu, zapobiegając dalszemu niepoprawnemu działaniu.

W krytycznym momencie wypisuje na ekran:

- numer oopsa
- status programu
- zawartość rejestrów procesora
- zawartość stosu

Czym jest oops?

Komunikat oops pojawia się w momencie wystąpienia błędu lub nieprzewidzianej sytuacji. Informuje o błędzie i inicjuje zakończenie pracy systemu, zapobiegając dalszemu niepoprawnemu działaniu.

W krytycznym momencie wypisuje na ekran:

- numer oopsa
- status programu
- zawartość rejestrów procesora
- zawartość stosu
- adresy ostatnich wywołań

Oops

```
Unable to handle kernel NULL pointer dereference at virtual address  00000014
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0 c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000 00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038 00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>] [<c0108f77>]
[<c010002b>]

Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

Jak działa oops?

Oops generuje się za pomocą umieszczenia w kodzie jądra następujących funkcji:

- BUG()
- BUG_ON()
- panic()

Ksymoops na pomoc...

Komunikaty wypisywane przez oops ciężko się interpretuje, dlatego stworzono inny proces o nazwie ksymoops. Wykorzystuje on wyniki oopsa i przerabia je na bardziej zrozumiałe. Ksymoops zbiera dane z Oops.file, logów, konsoli albo kopiuje z ekranu. Na podstawie ich oraz informacji z System.map (plik z symbolimi wykorzystywanymi przez jądro), proc/ksyms i lib/modules tłumaczy adresy z oopsa na symbole.

Printk() - składnia funkcji

- Funkcja printk() umożliwia debugowanie poprzez wypisywanie komunikatów będących jej parametrami.

Printk() - składnia funkcji

- Funkcja printk() umożliwia debugowanie poprzez wypisywanie komunikatów będących jej parametrami.
- Składnia:
printk(KERN_NOTICE "I am a kernel hacker!");.

Printk() - składnia funkcji

- Funkcja printk() umożliwia debugowanie poprzez wypisywanie komunikatów będących jej parametrami.
- Składnia:
  printk(KERN_NOTICE "I am a kernel hacker!");.
- Wypisuje komunikaty w zależności od ustawionego poziomu.

Printk() - składnia funkcji

- Funkcja printk() umożliwia debugowanie poprzez wypisywanie komunikatów będących jej parametrami.
- Składnia:
  printk(KERN_NOTICE "I am a kernel hacker!");.
- Wypisuje komunikaty w zależności od ustawionego poziomu.
- KERN_NOTICE jest zdefiniowane w pliku kernel.h i przyjmuje wartości 0-7 (im mniejsza liczba, tym większy priorytet).

Poziomy wypisywania komunikatów przez printk()

- KERN_EMERG <0> awaria systemu
- KERN_ALERT <1> należy podjąć natychmiastową interwencję
- KERN_CRIT <2> warunki krytyczne
- KERN_ERR <3> błąd
- KERN_WARNING <4> ostrzeżenie
- KERN_NOTICE <5> normalna, ale istotna sytuacja
- KERN_INFO <6> informacja
- KERN_DEBUG <7> komunikat diagnostyczny

Printk() - dlaczego to w ogóle działa?

- Komunikaty związane z działaniem jądra są zapisywane do cyklicznego bufora, z którego czytają je demony.

Printk() - dlaczego to w ogóle działa?

- Komunikaty związane z działaniem jądra są zapisywane do cyklicznego bufora, z którego czytają je demony.
- Printk() pomagają dwa demony: **syslogd** i **klogd**.

Printk() - dlaczego to w ogóle działa?

- Komunikaty związane z działaniem jądra są zapisywane do cyklicznego bufora, z którego czytają je demony.
- Printk() pomagają dwa demony: **syslogd** i **klogd**.
- Klogd do czytania komunikatów wykorzystuje proc/kmsg i przekazuje je do demona syslogd.

Printk() - dlaczego to w ogóle działa?

- Komunikaty związane z działaniem jądra są zapisywane do cyklicznego bufora, z którego czytają je demony.
- Printk() pomagają dwa demony: **syslogd** i **klogd**.
- Klogd do czytania komunikatów wykorzystuje proc/kmsg i przekazuje je do demona syslogd.
- Demon ksyslogd odbiera komunikaty i zapisuje je domyślnie w pliku /var/log/messages w zależności od priorytetów w innych miejscach.

Printk() - dlaczego to w ogóle działa?

- Komunikaty związane z działaniem jądra są zapisywane do cyklicznego bufora, z którego czytają je demony.
- Printk() pomagają dwa demony: **syslogd** i **klogd**.
- Klogd do czytania komunikatów wykorzystuje proc/kmsg i przekazuje je do demona syslogd.
- Demon ksyslogd odbiera komunikaty i zapisuje je domyślnie w pliku /var/log/messages w zależności od priorytetów w innych miejscach.
- Miejsce zapisywania komunikatów można zmienić w pliku /etc/syslog.conf.

Printk() - dlaczego to w ogóle działa?

- Komunikaty związane z działaniem jądra są zapisywane do cyklicznego bufora, z którego czytają je demony.
- Printk() pomagają dwa demony: **syslogd** i **klogd**.
- Klogd do czytania komunikatów wykorzystuje proc/kmsg i przekazuje je do demona syslogd.
- Demon ksyslogd odbiera komunikaty i zapisuje je domyślnie w pliku /var/log/messages w zależności od priorytetów w innych miejscach.
- Miejsce zapisywania komunikatów można zmienić w pliku /etc/syslog.conf.
- Jeżeli klogd nie działa i bufor cykliczny się przepełni, stracimy najstarsze komunikaty (zostaną one nadpisane).

Wpis z dziennika (plik var/log/messages)

```
Plik: messages          Kol 0            73817 bajtów
Nov 20 18:15:34 agata-laptop syslogd 1.4.1#21ubuntu3: restart.
Nov 20 18:15:34 agata-laptop kernel: Inspecting /boot/System.map-2.6.22-14-generic
Nov 20 18:15:34 agata-laptop kernel: Loaded 25445 symbols from /boot/System.map-2.6.22-14-generic.
Nov 20 18:15:34 agata-laptop kernel: Symbols match kernel version 2.6.22.
Nov 20 18:15:34 agata-laptop kernel: No module symbols loaded - kernel modules not enabled.
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] Linux version 2.6.22-14-generic (buildd@palmer) (gcc version 4.1.3 20070929 (prerelease)
ubuntu2)) #1 SMP Sun Oct 14 23:05:12 GMT 2007 (Ubuntu 2.6.22-14.46-generic)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] BIOS-provided physical RAM map:
Nov 20 18:15:34 agata-laptop kernel: [    0.000000]  BIOS-e820: 0000000000000000 - 000000000009fc00 (usable)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000]  BIOS-e820: 000000000009fc00 - 00000000000a0000 (reserved)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000]  BIOS-e820: 00000000000e0000 - 0000000000100000 (reserved)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000]  BIOS-e820: 0000000000100000 - 000000003ffc0000 (usable)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000]  BIOS-e820: 000000003ffc0000 - 000000003ffcf000 (ACPI data)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000]  BIOS-e820: 000000003ffcf000 - 0000000040000000 (ACPI NVS)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000]  BIOS-e820: 00000000fed14000 - 00000000fed1a000 (reserved)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000]  BIOS-e820: 00000000fed1c000 - 00000000fed20000 (reserved)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000]  BIOS-e820: 00000000ffb80000 - 0000000100000000 (reserved)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] 127MB HIGHMEM available.
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] 896MB LOWMEM available.
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] found SMP MP-table at 000ff780
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] Zone PFN ranges:
Nov 20 18:15:34 agata-laptop kernel: [    0.000000]   DMA             0 ->     4096
Nov 20 18:15:34 agata-laptop kernel: [    0.000000]   Normal       4096 ->   229376
Nov 20 18:15:34 agata-laptop kernel: [    0.000000]   HighMem    229376 ->   262080
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] early_node_map[1] active PFN ranges
Nov 20 18:15:34 agata-laptop kernel: [    0.000000]     0:        0 ->   262080
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] DMI 2.3 present.
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] ACPI: RSDP signature @ 0xC00F6980 checksum 0
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] ACPI: RSDP 000F6980, 0014 (r0 ACPIAM)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] ACPI: RSDT 3FFC0000, 0038 (r1 A M I  OEMRSDT   6000615 MSFT       97)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] ACPI: FACP 3FFC0200, 0084 (r2 A M I  OEMFACP   6000615 MSFT       97)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] ACPI: DSDT 3FFC0430, 89E6 (r1  0AAAA 0AAAA000        0 INTL  2002026)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] ACPI: FACS 3FFCF000, 0040
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] ACPI: APIC 3FFC0390, 0054 (r1 A M I  OEMAPIC   6000615 MSFT       97)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] ACPI: MCFG 3FFC03F0, 003C (r1 A M I  OEMMCFG   6000615 MSFT       97)
Nov 20 18:15:34 agata-laptop kernel: [    0.000000] ACPI: OEMB 3FFCF040, 0040 (r1 A M I  AMI_OEM   6000615 MSFT       97)
```

Printk()

Plusy printk()

- Łatwy w użyciu, podobny do powszechnie znanej funkcji printf() sposób debugowania.

Printk()

Plusy printk()

- Łatwy w użyciu, podobny do powszechnie znanej funkcji printf() sposób debugowania.
- Funkcji można używać w dowolnym miejscu kodu jądra.

Printk()

Minusy printk()

- Nie udaje się wykryć błędów, które pojawiają się przed zainicjowaniem konsoli. (Można wtedy użyć early_printk(), która działa bardzo podobnie).

Printk()

Minusy printk()

- Nie udaje się wykryć błędów, które pojawiają się przed zainicjowaniem konsoli. (Można wtedy użyć early_printk(), która działa bardzo podobnie).
- Debugowanie tym sposobem trzeba wesprzeć myśleniem i odrobiną rozsądku, ponieważ przy wypisywaniu dużej ilości komunikatów łatwo doprowadzić do przeciążenia (Albo użyć printk_ratelimit()).

Strace i ltrace

- **Strace** - wypisuje wywołania systemowe, sygnały jakie otrzymuje śledzony proces

Strace i ltrace

- **Strace** - wypisuje wywołania systemowe, sygnały jakie otrzymuje śledzony proces
- **Ltrace** - wypisuje wywołania bibliotek ładowanych dynamicznie, sygnały, również wywołania systemowe

Strace i ltrace

- **Strace** - wypisuje wywołania systemowe, sygnały jakie otrzymuje śledzony proces
- **Ltrace** - wypisuje wywołania bibliotek ładowanych dynamicznie, sygnały, również wywołania systemowe
- Nie trzeba posiadać źródeł procesu ani ich wcześniej odpowiednio kompilować.

Strace i ltrace

- **Strace** - wypisuje wywołania systemowe, sygnały jakie otrzymuje śledzony proces
- **Ltrace** - wypisuje wywołania bibliotek ładowanych dynamicznie, sygnały, również wywołania systemowe
- Nie trzeba posiadać źródeł procesu ani ich wcześniej odpowiednio kompilować.
- Wypisują nazwy, parametry i wyniki wywołanych funkcji lub nazwę i opis błędu.

Strace i ltrace

- **Strace** - wypisuje wywołania systemowe, sygnały jakie otrzymuje śledzony proces
- **Ltrace** - wypisuje wywołania bibliotek ładowanych dynamicznie, sygnały, również wywołania systemowe
- Nie trzeba posiadać źródeł procesu ani ich wcześniej odpowiednio kompilować.
- Wypisują nazwy, parametry i wyniki wywołanych funkcji lub nazwę i opis błędu.
- Same nie są zbyt przydatne, ale w połączenie z UML-em są dobrym narzędziem do debugowania.

Przykład strace

```
agata@agata-laptop:~$ strace sleep 0.1
execve("/bin/sleep", ["sleep", "0.1"], [/* 35 vars */]) = 0
brk(0)                                  = 0x804c000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fc4000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=40191, ...}) = 0
mmap2(NULL, 40191, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7fba000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\260a\1"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=1339816, ...}) = 0
mmap2(NULL, 1349136, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7e70000
mmap2(0xb7fb4000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x143) = 0xb7fb4000
mmap2(0xb7fb7000, 9744, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7fb7000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7e6f000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7e6f6b0, limit:1048575, seg_32bit:1, contents:0, read_exec_only:0,
seable:1}) = 0
mprotect(0xb7fb4000, 4096, PROT_READ)   = 0
munmap(0xb7fba000, 40191)               = 0
brk(0)                                  = 0x804c000
brk(0x806d000)                          = 0x806d000
open("/usr/lib/locale/locale-archive", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or directory)
open("/usr/share/locale/locale.alias", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=2586, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fc3000
read(3, "# Locale name alias data base.\n#"..., 4096) = 2586
read(3, "", 4096)                       = 0
close(3)                                = 0
munmap(0xb7fc3000, 4096)                = 0
open("/usr/lib/locale/pl_PL.UTF-8/LC_IDENTIFICATION", O_RDONLY) = -1 ENOENT (No such file or directory)
```

Ciąg dalszy krótkiego przykładu...

```
fstat64(3, {st_mode=S_IFREG|0644, st_size=34, ...}) = 0
mmap2(NULL, 34, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7e6c000
close(3)                                = 0
open("/usr/lib/locale/pl_PL.UTF-8/LC_MESSAGES", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/lib/locale/pl_PL.utf8/LC_MESSAGES", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
close(3)                                = 0
open("/usr/lib/locale/pl_PL.utf8/LC_MESSAGES/SYS_LC_MESSAGES", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=54, ...}) = 0
mmap2(NULL, 54, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7e6b000
close(3)                                = 0
open("/usr/lib/locale/pl_PL.UTF-8/LC_MONETARY", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/lib/locale/pl_PL.utf8/LC_MONETARY", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=294, ...}) = 0
mmap2(NULL, 294, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7e6a000
close(3)                                = 0
open("/usr/lib/locale/pl_PL.UTF-8/LC_COLLATE", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/lib/locale/pl_PL.utf8/LC_COLLATE", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=915398, ...}) = 0
mmap2(NULL, 915398, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7d8a000
close(3)                                = 0
open("/usr/lib/locale/pl_PL.UTF-8/LC_TIME", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/lib/locale/pl_PL.utf8/LC_TIME", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=2404, ...}) = 0
mmap2(NULL, 2404, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7d89000
close(3)                                = 0
open("/usr/lib/locale/pl_PL.UTF-8/LC_NUMERIC", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/lib/locale/pl_PL.utf8/LC_NUMERIC", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=54, ...}) = 0
mmap2(NULL, 54, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7d88000
close(3)                                = 0
open("/usr/lib/locale/pl_PL.UTF-8/LC_CTYPE", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/lib/locale/pl_PL.utf8/LC_CTYPE", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=254020, ...}) = 0
mmap2(NULL, 254020, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7d49000
```

KDB - instalacja

Aby uruchomić KDB należy:

- Ściągnąć odpowiednie łatki.

KDB - instalacja

Aby uruchomić KDB należy:

- Ściągnąć odpowiednie łatki.
- W kernel hacking ustawić odpowiednie flagi (CONFIG_KDB).

KDB - instalacja

Aby uruchomić KDB należy:

- Ściągnąć odpowiednie łatki.
- W kernel hacking ustawić odpowiednie flagi (CONFIG_KDB).
- Przekompilować jądro.

KDB - instalacja

Aby uruchomić KDB należy:

- Ściągnąć odpowiednie łatki.
- W kernel hacking ustawić odpowiednie flagi (CONFIG_KDB).
- Przekompilować jądro.
- Włączyć kdb (echo "1»/proc/sys/kernel/kdb) i uruchomić ponownie komputer.

KDB

Co daje nam KDB?

- Możliwość wykonywania kodu jądra krok po kroku.

KDB

Co daje nam KDB?

- Możliwość wykonywania kodu jądra krok po kroku.
- Możliwość zatrzymania działania na wybranej instrukcji.

KDB

Co daje nam KDB?

- Możliwość wykonywania kodu jądra krok po kroku.
- Możliwość zatrzymania działania na wybranej instrukcji.
- Możliwość zatrzymania działania podczas modyfikacji wskazanego adresu pamięci.

KDB

Co daje nam KDB?

- Możliwość wykonywania kodu jądra krok po kroku.
- Możliwość zatrzymania działania na wybranej instrukcji.
- Możliwość zatrzymania działania podczas modyfikacji wskazanego adresu pamięci.
- Możliwość zatrzymania działanie przy dostępie do wskazanego rejestru.

KDB

Co daje nam KDB?

- Możliwość wykonywania kodu jądra krok po kroku.
- Możliwość zatrzymania działania na wybranej instrukcji.
- Możliwość zatrzymania działania podczas modyfikacji wskazanego adresu pamięci.
- Możliwość zatrzymania działanie przy dostępie do wskazanego rejestru.
- Możliwość śledzenia stosu wybranego procesu.

KDB

Wady KDB:

- Brak możliwości modyfikacji kodu jądra w czasie jego działania - niewygoda.

KDB

Wady KDB:

- Brak możliwości modyfikacji kodu jądra w czasie jego działania - niewygoda.
- Nie jest możliwe debugowanie z poziomu kodu - tylko instrukcje assemblera.

KDB

KDB zostaje przywołane w następujących przypadkach:

- automatycznie - jeśli napotkamy na kernel panic,
- automatycznie - jeśli napotka na wcześniej zdefiniowany breakpoint,
- ręcznie - przez naciśnięcie PAUSE (Ctrl-Pause Break) przez użytkownika,
- Ctrl-A - wywołuje KDB z serial console.

KDB - kilka podstawowych komend

- **md** <adres> - wypisuje zawartość pamięci od tego adresu

KDB - kilka podstawowych komend

- **md** <adres> - wypisuje zawartość pamięci od tego adresu
- **rd** - wypisuje zawartość wszystkich rejestrów

KDB - kilka podstawowych komend

- **md** <adres> - wypisuje zawartość pamięci od tego adresu
- **rd** - wypisuje zawartość wszystkich rejestrów
- **rm** <rejestr> <wartosc> - zapisuje wartosc w podanym rejestrze

KDB - kilka podstawowych komend

- **md** <adres> - wypisuje zawartość pamięci od tego adresu
- **rd** - wypisuje zawartość wszystkich rejestrów
- **rm** <rejestr> <wartosc> - zapisuje wartosc w podanym rejestrze
- **bp** <instrukcja> - zatrzymuje się na podanej instrukcji

KDB - kilka podstawowych komend

- **md** <adres> - wypisuje zawartość pamięci od tego adresu
- **rd** - wypisuje zawartość wszystkich rejestrów
- **rm** <rejestr> <wartosc> - zapisuje wartosc w podanym rejestrze
- **bp** <instrukcja> - zatrzymuje się na podanej instrukcji
- **bt** - wypisuje stos aktualnego procesu

KDB - kilka podstawowych komend

- **md** <adres> - wypisuje zawartość pamięci od tego adresu
- **rd** - wypisuje zawartość wszystkich rejestrów
- **rm** <rejestr> <wartosc> - zapisuje wartosc w podanym rejestrze
- **bp** <instrukcja> - zatrzymuje się na podanej instrukcji
- **bt** - wypisuje stos aktualnego procesu
- **bta** - wypisuje stosy wszystkich procesów

KDB - kilka podstawowych komend

- **md** <adres> - wypisuje zawartość pamięci od tego adresu
- **rd** - wypisuje zawartość wszystkich rejestrów
- **rm** <rejestr> <wartosc> - zapisuje wartosc w podanym rejestrze
- **bp** <instrukcja> - zatrzymuje się na podanej instrukcji
- **bt** - wypisuje stos aktualnego procesu
- **bta** - wypisuje stosy wszystkich procesów
- **ss** - wykonuje jedną instrukcję

KDB - kilka podstawowych komend

- **md** <adres> - wypisuje zawartość pamięci od tego adresu
- **rd** - wypisuje zawartość wszystkich rejestrów
- **rm** <rejestr> <wartosc> - zapisuje wartosc w podanym rejestrze
- **bp** <instrukcja> - zatrzymuje się na podanej instrukcji
- **bt** - wypisuje stos aktualnego procesu
- **bta** - wypisuje stosy wszystkich procesów
- **ss** - wykonuje jedną instrukcję
- **go** - wykonuje instrukcje do najbliższego breakpointu

KDB - kilka podstawowych komend

- **md** <adres> - wypisuje zawartość pamięci od tego adresu
- **rd** - wypisuje zawartość wszystkich rejestrów
- **rm** <rejestr> <wartosc> - zapisuje wartosc w podanym rejestrze
- **bp** <instrukcja> - zatrzymuje się na podanej instrukcji
- **bt** - wypisuje stos aktualnego procesu
- **bta** - wypisuje stosy wszystkich procesów
- **ss** - wykonuje jedną instrukcję
- **go** - wykonuje instrukcje do najbliższego breakpointu
- **reboot** - restartuje

GDB

GDB - co to takiego?

- Gnu Project Debugger

GDB

GDB - co to takiego?

- Gnu Project Debugger
- Standardowy Debugger Linuxowy

GDB

GDB - co to takiego?

- Gnu Project Debugger
- Standardowy Debugger Linuxowy

Co może?

- Uruchamia programy użytkownika.

GDB

GDB - co to takiego?

- Gnu Project Debugger
- Standardowy Debugger Linuxowy

Co może?

- Uruchamia programy użytkownika.
- Zatrzymuje przy zajściu podanych warunków.

GDB

GDB - co to takiego?

- Gnu Project Debugger
- Standardowy Debugger Linuxowy

Co może?

- Uruchamia programy użytkownika.
- Zatrzymuje przy zajściu podanych warunków.
- Przegląda pamięć.

GDB

GDB - co to takiego?

- Gnu Project Debugger
- Standardowy Debugger Linuxowy

Co może?

- Uruchamia programy użytkownika.
- Zatrzymuje przy zajściu podanych warunków.
- Przegląda pamięć.
- Modyfikuje zawartość pamięci „w locie”.

GDB

Co jest potrzebne?

- Program, który opluskwiamy musi mieć zapisane pewne informacje symboliczne.

GDB

Co jest potrzebne?

- Program, który opluskwiamy musi mieć zapisane pewne informacje symboliczne.
- Kompilując przy pomocy gcc musimy podać opcję –g.

GDB

Co jest potrzebne?

- Program, który opluskwiamy musi mieć zapisane pewne informacje symboliczne.
- Kompilując przy pomocy gcc musimy podać opcję –g.
- W przypadku jądra linuxa musi ono pracować w trybie UML i być odpowiednio skompilowane – szczegóły w dalszej części prezentacji.

GDB

Początek z GDB

- Uruchamiamy gdb <program>.

GDB

Początek z GDB

- Uruchamiamy gdb <program>.
- Nie podajemy argumentów wywołania.

GDB

Początek z GDB

- Uruchamiamy gdb <program>.
- Nie podajemy argumentów wywołania.
- Aby uruchomić program w gdb wydajemy polecenie r (run) <argumenty>.

GDB

Początek z GDB

- Uruchamiamy gdb <program>.
- Nie podajemy argumentów wywołania.
- Aby uruchomić program w gdb wydajemy polecenie r (run) <argumenty>.
- Gdy gdb natrafi w programie na breakpoint, program zostanie zatrzymany i przejdziemy do konsoli gdb.

GDB

Początek z GDB

- Uruchamiamy gdb <program>.
- Nie podajemy argumentów wywołania.
- Aby uruchomić program w gdb wydajemy polecenie r (run) <argumenty>.
- Gdy gdb natrafi w programie na breakpoint, program zostanie zatrzymany i przejdziemy do konsoli gdb.
- Analogicznys efekt uzyskamy, gdy wyślemy do debugowanego programu SIGINT.

GDB - breakpointy

- Aby ustawić breakpoint należy skorzystać z komendy **b** (break) <arg>.

GDB - breakpointy

- Aby ustawić breakpoint należy skorzystać z komendy **b** (break) <arg>.
- **b <nazwa-funkcji>** spowoduje zatrzymanie programu, gdy wejdziemy do funkcji podanej jako parametr.

GDB - breakpointy

- Aby ustawić breakpoint należy skorzystać z komendy **b** (break) <arg>.
- **b <nazwa-funkcji>** spowoduje zatrzymanie programu, gdy wejdziemy do funkcji podanej jako parametr.
- **b <numer-linii>** spowoduje zatrzymanie programu, gdy wykonywanie dojdzie do danej linii w kodzie źródłowym.

GDB - breakpointy

- Aby ustawić breakpoint należy skorzystać z komendy **b** (break) <arg>.
- **b <nazwa-funkcji>** spowoduje zatrzymanie programu, gdy wejdziemy do funkcji podanej jako parametr.
- **b <numer-linii>** spowoduje zatrzymanie programu, gdy wykonywanie dojdzie do danej linii w kodzie źródłowym.
- Aby skasować breakpoint podajemy komende **d** (delete) <numer-breakpointa>.

GDB - breakpointy

- Aby ustawić breakpoint należy skorzystać z komendy **b** (break) <arg>.
- **b <nazwa-funkcji>** spowoduje zatrzymanie programu, gdy wejdziemy do funkcji podanej jako parametr.
- **b <numer-linii>** spowoduje zatrzymanie programu, gdy wykonywanie dojdzie do danej linii w kodzie źródłowym.
- Aby skasować breakpoint podajemy komende **d** (delete) <numer-breakpointa>.
- **d** bez argumentów kasuje wszystkie breakpointy.

GDB - komendy

- l (list) – wypisanie kodu przy miejscu zatrzymania programu

GDB - komendy

- **l** (list) – wypisanie kodu przy miejscu zatrzymania programu
- **n** (next) - przejście do następnej funkcji

GDB - komendy

- l (list) – wypisanie kodu przy miejscu zatrzymania programu
- **n** (next) - przejście do następnej funkcji
- **s** (step) – wykonanie następnej linijki kodu

GDB - komendy

- l (list) – wypisanie kodu przy miejscu zatrzymania programu
- **n** (next) - przejście do następnej funkcji
- **s** (step) – wykonanie następnej linijki kodu
- **c** (continue) – wykonywanie programu do momentu jego zakończenia lub zatrzymania się na breakpoincie

GDB - komendy

- l (list) – wypisanie kodu przy miejscu zatrzymania programu
- n (next) - przejście do następnej funkcji
- s (step) – wykonanie następnej linijki kodu
- c (continue) – wykonywanie programu do momentu jego zakończenia lub zatrzymania się na breakpoincie
- f (finish) – wykonywanie całej aktualnej funkcji

GDB - komendy cz. 2

- **print** <wyr> - powoduje wypisanie wartości wyrażenia na ekran, korzystając z wartości zmiennych w danym momencie

GDB - komendy cz. 2

- **print** <wyr> - powoduje wypisanie wartości wyrażenia na ekran, korzystając z wartości zmiennych w danym momencie
- **display** <wyr> - print, który po każdym wykonaniu kroku (step, next) wypisuje wartość na ekran, korzystając z aktualnego wartościowania

GDB - komendy cz. 2

- **print** <wyr> - powoduje wypisanie wartości wyrażenia na ekran, korzystając z wartości zmiennych w danym momencie
- **display** <wyr> - print, który po każdym wykonaniu kroku (step, next) wypisuje wartość na ekran, korzystając z aktualnego wartościowania
- **set** var <przyp> - wykonuje wskazane przypisanie (modyfikuje pamięć „w locie")

GDB - komendy cz. 2

- **print** <wyr> - powoduje wypisanie wartości wyrażenia na ekran, korzystając z wartości zmiennych w danym momencie
- **display** <wyr> - print, który po każdym wykonaniu kroku (step, next) wypisuje wartość na ekran, korzystając z aktualnego wartościowania
- **set** var <przyp> - wykonuje wskazane przypisanie (modyfikuje pamięć „w locie”)
- **bt** (backtrace) – pokazuje stos programu

GDB - komendy cz. 3

- **handle** <sig> <arg> - ustawia zachowanie gdb przy napotkaniu na sygnał
  - (no)stop – czy ma się zatrzymywac

GDB - komendy cz. 3

- **handle** <sig> <arg> - ustawia zachowanie gdb przy napotkaniu na sygnał
  - (no)stop – czy ma się zatrzymywac
  - (no)print – czy ma wypisywać na ekran

GDB - komendy cz. 3

- **handle** <sig> <arg> - ustawia zachowanie gdb przy napotkaniu na sygnał
  - (no)stop – czy ma się zatrzymywac
  - (no)print – czy ma wypisywać na ekran
  - (no)pass – czy ma być widoczny

GDB - komendy cz. 3

- **handle** <sig> <arg> - ustawia zachowanie gdb przy napotkaniu na sygnał
  - (no)stop – czy ma się zatrzymywac
  - (no)print – czy ma wypisywać na ekran
  - (no)pass – czy ma być widoczny
- **info** <arg> - pokazuje pomoc do polecenia

GDB - komendy cz. 3

- **handle** <sig> <arg> - ustawia zachowanie gdb przy napotkaniu na sygnał
  - (no)stop – czy ma się zatrzymywac
  - (no)print – czy ma wypisywać na ekran
  - (no)pass – czy ma być widoczny
- **info** <arg> - pokazuje pomoc do polecenia
- **quit** – wyjście z gdb

KGDB

KGDB –jak debugować z daleka?

- KGDB – Patch, który wgrywamy do kodu źródłowego jądra.

KGDB

KGDB –jak debugować z daleka?

- KGDB – Patch, który wgrywamy do kodu źródłowego jądra.
- Debugowanie odbywa się na niemalże tej samej zasadzie co debugowanie normalnych programów.

KGDB

KGDB –jak debugować z daleka?

- KGDB – Patch, który wgrywamy do kodu źródłowego jądra.
- Debugowanie odbywa się na niemalże tej samej zasadzie co debugowanie normalnych programów.
- Jest to debugowanie zdalne – potrzebne są do tego dwie maszyny (mogą być wirtualne).

KGDB

KGDB –jak debugować z daleka?

- KGDB – Patch, który wgrywamy do kodu źródłowego jądra.
- Debugowanie odbywa się na niemalże tej samej zasadzie co debugowanie normalnych programów.
- Jest to debugowanie zdalne – potrzebne są do tego dwie maszyny (mogą być wirtualne).
- Jądro uruchamiamy na maszynie docelowej.

KGDB

KGDB –jak debugować z daleka?

- KGDB – Patch, który wgrywamy do kodu źródłowego jądra.
- Debugowanie odbywa się na niemalże tej samej zasadzie co debugowanie normalnych programów.
- Jest to debugowanie zdalne – potrzebne są do tego dwie maszyny (mogą być wirtualne).
- Jądro uruchamiamy na maszynie docelowej.
- Debugujemy na maszynie deweloper.

KGDB

Co możemy zrobić?

- Przede wszystkim: pełen zasób możliwości zwykłego gdb (breakpointy, zmienne, przeglądanie kodu, itp.)

KGDB

Co możemy zrobić?

- Przede wszystkim: pełen zasób możliwości zwykłego gdb (breakpointy, zmienne, przeglądanie kodu, itp.)
- Przykład: b /net/ipv4/tcp_ipv4.c:744.

KGDB

Co możemy zrobić?

- Przede wszystkim: pełen zasób możliwości zwykłego gdb (breakpointy, zmienne, przeglądanie kodu, itp.)
- Przykład: b /net/ipv4/tcp_ipv4.c:744.
- Debugowanie modułów (wymaga gdbmod – patrz strona kgdb).

KGDB

Co możemy zrobić?

- Przede wszystkim: pełen zasób możliwości zwykłego gdb (breakpointy, zmienne, przeglądanie kodu, itp.)
- Przykład: b /net/ipv4/tcp_ipv4.c:744.
- Debugowanie modułów (wymaga gdbmod – patrz strona kgdb).
- Obsługa wieloprocesorowości (wszystkie procesory są naraz w kgdb).

KGDB

Co możemy zrobić?

- Przede wszystkim: pełen zasób możliwości zwykłego gdb (breakpointy, zmienne, przeglądanie kodu, itp.)
- Przykład: b /net/ipv4/tcp_ipv4.c:744.
- Debugowanie modułów (wymaga gdbmod – patrz strona kgdb).
- Obsługa wieloprocesorowości (wszystkie procesory są naraz w kgdb).
- Obsługa wątków (info threads).

KGDB

Na co warto zwrócić uwagę?

- „Console messages through gdb" - pozwala wypisywać komunikaty z konsoli bezpośrednio do GDB (znacznie ułatwia debugowanie).

KGDB

Na co warto zwrócić uwagę?

- „Console messages through gdb” - pozwala wypisywać komunikaty z konsoli bezpośrednio do GDB (znacznie ułatwia debugowanie).
- Ethernet – we wstępnej wersji pojawiła się już możliwość debugowania przez sieć (uwaga: ta wersja nie jest równie stabilna co przez port szeregowy).

KGDB

Na co warto zwrócić uwagę?

- „Console messages through gdb" - pozwala wypisywać komunikaty z konsoli bezpośrednio do GDB (znacznie ułatwia debugowanie).
- Ethernet – we wstępnej wersji pojawiła się już możliwość debugowania przez sieć (uwaga: ta wersja nie jest równie stabilna co przez port szeregowy).
- Odłączanie GDB – możemy odłączyć gdb bez wyłączania jądra – po czym wrócić do niego.).

KGDB

Jak zainstalować?

- 2 komputery?

KGDB

Jak zainstalować?

- 2 komputery? Jasne – 2 maszyny wirtualne!

KGDB

Jak zainstalować?

- 2 komputery? Jasne – 2 maszyny wirtualne!
- Połączone kablem szeregowym?! Czemu nie – nazwane łącze pipe.

KGDB

Jak zainstalować?

- 2 komputery? Jasne – 2 maszyny wirtualne!
- Połączone kablem szeregowym?! Czemu nie – nazwane łącze pipe.
- Ściągnij źródła Linuxa oraz patch (patrz http://kgdb.linsyssoft.com/downloads.htm).

KGDB

Jak zainstalować?

- 2 komputery? Jasne – 2 maszyny wirtualne!
- Połączone kablem szeregowym?! Czemu nie – nazwane łącze pipe.
- Ściągnij źródła Linuxa oraz patch (patrz http://kgdb.linsyssoft.com/downloads.htm).
- Wypakuj źródła i patch, po czym nanieś patch.

KGDB

Jak zainstalować?

- 2 komputery? Jasne – 2 maszyny wirtualne!
- Połączone kablem szeregowym?! Czemu nie – nazwane łącze pipe.
- Ściągnij źródła Linuxa oraz patch (patrz http://kgdb.linsyssoft.com/downloads.htm).
- Wypakuj źródła i patch, po czym nanieś patch.
- Najprościej – przenieś patche do folderu ./patches quilt push -a.

KGDB

Kompilujemy

- make menuconfig

KGDB

Kompilujemy

- make menuconfig
- Zaznaczamy opcje w zakładce Kernel Hacking (to brzmi dumnie).

KGDB

Kompilujemy

- make menuconfig
- Zaznaczamy opcje w zakładce Kernel Hacking (to brzmi dumnie).
- KGDB: kernel debugging with remote gdb.

KGDB

Kompilujemy

- make menuconfig
- Zaznaczamy opcje w zakładce Kernel Hacking (to brzmi dumnie).
- KGDB: kernel debugging with remote gdb.
- Serial port number for KGDB: 0.

KGDB

Kompilujemy

- make menuconfig
- Zaznaczamy opcje w zakładce Kernel Hacking (to brzmi dumnie).
- KGDB: kernel debugging with remote gdb.
- Serial port number for KGDB: 0.
- Pozostałe opcje według uznania.

KGDB

Kompilujemy

- make menuconfig
- Zaznaczamy opcje w zakładce Kernel Hacking (to brzmi dumnie).
- KGDB: kernel debugging with remote gdb.
- Serial port number for KGDB: 0.
- Pozostałe opcje według uznania.
- Kilka uwag do aktualnie używanej wersji.

KGDB

Kompilujemy

- make menuconfig
- Zaznaczamy opcje w zakładce Kernel Hacking (to brzmi dumnie).
- KGDB: kernel debugging with remote gdb.
- Serial port number for KGDB: 0.
- Pozostałe opcje według uznania.
- Kilka uwag do aktualnie używanej wersji.
- Kompilujemy! (I kompilujemy.. i kompilujemy)...

KGDB

A potem...

- Wracamy po godzinie, może się udało!

KGDB

A potem...

- Wracamy po godzinie, może się udało!
- Prawdopodobnie nie..

KGDB

A potem...

- Wracamy po godzinie, może się udało!
- Prawdopodobnie nie..
- Ale jeśli się udało mamy kilka możliwości:

KGDB

A potem...

- Wracamy po godzinie, może się udało!
- Prawdopodobnie nie..
- Ale jeśli się udało mamy kilka możliwości:
  - Skopiować plik bzImage do /boot na maszynie docelowej (./arch/i386/boot/bzImage)

KGDB

A potem...

- Wracamy po godzinie, może się udało!
- Prawdopodobnie nie..
- Ale jeśli się udało mamy kilka możliwości:
  - Skopiować plik bzImage do /boot na maszynie docelowej (./arch/i386/boot/bzImage)
  - Stworzyć pakiet .deb (make-kpkg) i zainstalować na maszynie docelowej

KGDB

A potem...

- Wracamy po godzinie, może się udało!
- Prawdopodobnie nie..
- Ale jeśli się udało mamy kilka możliwości:
  - Skopiować plik bzImage do /boot na maszynie docelowej (./arch/i386/boot/bzImage)
  - Stworzyć pakiet .deb (make-kpkg) i zainstalować na maszynie docelowej
  - Zrobić to wszystko jeszcze raz na maszynie docelowej bez gwarancji, że zadziała...

KGDB

I dalej...

- Jeśli zrobiliśmy to przy użyciu pakietu – mamy już dodany wpis do GRUB'a, jeśli nie – dodajemy (/boot/grub/menu.lst).

KGDB

I dalej...

- Jeśli zrobiliśmy to przy użyciu pakietu – mamy już dodany wpis do GRUB'a, jeśli nie – dodajemy (/boot/grub/menu.lst).
- W tym wpisie na samym końcu dodajemy parametr wywołania kgdbwait.

KGDB

I dalej...

- Jeśli zrobiliśmy to przy użyciu pakietu – mamy już dodany wpis do GRUB'a, jeśli nie – dodajemy (/boot/grub/menu.lst).
- W tym wpisie na samym końcu dodajemy parametr wywołania kgdbwait.
- Pozostaje połączyć nasze maszyny kablem na COM1...

KGDB

I dalej...

- Jeśli zrobiliśmy to przy użyciu pakietu – mamy już dodany wpis do GRUB'a, jeśli nie – dodajemy (/boot/grub/menu.lst).
- W tym wpisie na samym końcu dodajemy parametr wywołania kgdbwait.
- Pozostaje połączyć nasze maszyny kablem na COM1...
- W VMWare edytujemy ustawienia naszej maszyny docelowej – dodajemy port szeregowy (\\.\pipe\com_1 oraz Yield CPU on Poll).

KGDB

Już prawie...

- Potrzebny nam będzie program „Named Pipe TCP Proxy”.

KGDB

Już prawie...

- Potrzebny nam będzie program „Named Pipe TCP Proxy”.
- Ustawiamy \\.\pipe\com_1, port 6969 (lub inny wybrany), pozwalamy na dostęp z zewnątrz.

KGDB

Już prawie...

- Potrzebny nam będzie program „Named Pipe TCP Proxy”.
- Ustawiamy \\.\pipe\com_1, port 6969 (lub inny wybrany), pozwalamy na dostęp z zewnątrz.
- Na koniec: Na maszynie deweloperskiej uruchamiamy gdb.

KGDB

Już prawie...

- Potrzebny nam będzie program „Named Pipe TCP Proxy”.
- Ustawiamy \\.\pipe\com_1, port 6969 (lub inny wybrany), pozwalamy na dostęp z zewnątrz.
- Na koniec: Na maszynie deweloperskiej uruchamiamy gdb.
- Wywołujemy target remote ipkomputeradomowego:6969.

KGDB

Już prawie...

- Potrzebny nam będzie program „Named Pipe TCP Proxy”.
- Ustawiamy \\.\pipe\com_1, port 6969 (lub inny wybrany), pozwalamy na dostęp z zewnątrz.
- Na koniec: Na maszynie deweloperskiej uruchamiamy gdb.
- Wywołujemy target remote ipkomputeradomowego:6969.
- I do boju! (np. wpiszmy continue!).

UML - User Mode Linux

- Nie mylić z Unified Modeling Language ;)

UML - User Mode Linux

- Nie mylić z Unified Modeling Language ;)
- Port of Linux to Linux.

UML - User Mode Linux

- Nie mylić z Unified Modeling Language ;)
- Port of Linux to Linux.
- Odpowiednio skompilowane jądro.

UML - User Mode Linux

- Nie mylić z Unified Modeling Language ;)
- Port of Linux to Linux.
- Odpowiednio skompilowane jądro.
- Pracuje na architekturze wirtualnej um.

UML - User Mode Linux

- Nie mylić z Unified Modeling Language ;)
- Port of Linux to Linux.
- Odpowiednio skompilowane jądro.
- Pracuje na architekturze wirtualnej um.
- Od wersji 2.6.9 zintegrowany z jądrem.

UML - User Mode Linux

- Nie mylić z Unified Modeling Language ;)
- Port of Linux to Linux.
- Odpowiednio skompilowane jądro.
- Pracuje na architekturze wirtualnej um.
- Od wersji 2.6.9 zintegrowany z jądrem.
- Głównie do testowania modułów oraz innych niebezpiecznych aplikacji.

UML

Zalety UML

- Dużo szybszy od innych maszyn wirtualnych.

UML

Zalety UML

- Dużo szybszy od innych maszyn wirtualnych.
- Może być uruchomiony przez użytkownika nie posiadającego uprawnień roota.

UML

Zalety UML

- Dużo szybszy od innych maszyn wirtualnych.
- Może być uruchomiony przez użytkownika nie posiadającego uprawnień roota.
- Uruchamianie jako proces o zwyczajnym priorytecie zapewnia wysokie bezpieczeństwo w stosunku do hosta.

UML

Zalety UML

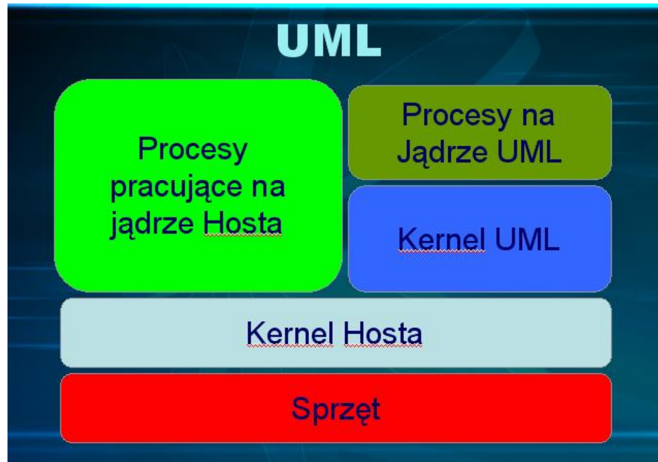
- Dużo szybszy od innych maszyn wirtualnych.
- Może być uruchomiony przez użytkownika nie posiadającego uprawnień roota.
- Uruchamianie jako proces o zwyczajnym priorytecie zapewnia wysokie bezpieczeństwo w stosunku do hosta.
- Możliwość zagnieżdżania instancji UML.

UML

Zalety UML

- Dużo szybszy od innych maszyn wirtualnych.
- Może być uruchomiony przez użytkownika nie posiadającego uprawnień roota.
- Uruchamianie jako proces o zwyczajnym priorytecie zapewnia wysokie bezpieczeństwo w stosunku do hosta.
- Możliwość zagnieżdżania instancji UML.
- Możliwość użycia dowolnego jądra.

UML



UML

Jak to działa?

- UML uruchamia swój scheduler, ma własną pamięc i sam zarządza procesami.

UML

Jak to działa?

- UML uruchamia swój scheduler, ma własną pamięc i sam zarządza procesami.
- Procesy uruchomione pod platformą UML proszą o dostęp do urządzeń jądro UML, a ono zgłasza się do jądra hosta.

UML

Jak to działa?

- UML uruchamia swój scheduler, ma własną pamięc i sam zarządza procesami.
- Procesy uruchomione pod platformą UML proszą o dostęp do urządzeń jądro UML, a ono zgłasza się do jądra hosta.
- I tylko po to? ;)

UML

Jak to działa?

- UML uruchamia swój scheduler, ma własną pamięc i sam zarządza procesami.
- Procesy uruchomione pod platformą UML proszą o dostęp do urządzeń jądro UML, a ono zgłasza się do jądra hosta.
- I tylko po to? ;)
- Dostarcza sterowniki dla wirtualnego systemu plików (VFS).

UML

Dwa tryby działania UML

- **TT** – Tracing Thread

UML

Dwa tryby działania UML

- **TT** – Tracing Thread
  - Każdy proces na UML jest procesem na hoście.

UML

Dwa tryby działania UML

- **TT** – Tracing Thread
  - Każdy proces na UML jest procesem na hoście.
  - Niewspierany i usuwany z nowych wersji jądra.

UML

Dwa tryby działania UML

- **TT** – Tracing Thread
  - Każdy proces na UML jest procesem na hoście.
  - Niewspierany i usuwany z nowych wersji jądra.
- **SKAS** - Single Kernel Address Space

UML

Dwa tryby działania UML

- **TT** – Tracing Thread
  - Każdy proces na UML jest procesem na hoście.
  - Niewspierany i usuwany z nowych wersji jądra.
- **SKAS** - Single Kernel Address Space
  - Procesy uruchamiane na UML mają swoją własną przestrzeń adresową.

UML

Dwa tryby działania UML

- **TT** – Tracing Thread
  - Każdy proces na UML jest procesem na hoście.
  - Niewspierany i usuwany z nowych wersji jądra.
- **SKAS** - Single Kernel Address Space
  - Procesy uruchamiane na UML mają swoją własną przestrzeń adresową.
  - Ogranicza liczbę procesów widocznych w maszynie hosta.

UML

Dwa tryby działania UML

- **TT** – Tracing Thread
  - Każdy proces na UML jest procesem na hoście.
  - Niewspierany i usuwany z nowych wersji jądra.
- **SKAS** - Single Kernel Address Space
  - Procesy uruchamiane na UML mają swoją własną przestrzeń adresową.
  - Ogranicza liczbę procesów widocznych w maszynie hosta.
  - Upraszcza widok UML z punktu widzenia hosta.

UML

Dwa tryby działania UML

- **TT** – Tracing Thread
  - Każdy proces na UML jest procesem na hoście.
  - Niewspierany i usuwany z nowych wersji jądra.
- **SKAS** - Single Kernel Address Space
  - Procesy uruchamiane na UML mają swoją własną przestrzeń adresową.
  - Ogranicza liczbę procesów widocznych w maszynie hosta.
  - Upraszcza widok UML z punktu widzenia hosta.
  - Niesie większe możliwości.

UML

Instalacja UML

- Instalacja z paczek – dostępna w większości dystrybucji.

UML

Instalacja UML

- Instalacja z paczek – dostępna w większości dystrybucji.
- Instalacja ze źródeł:

UML

Instalacja UML

- Instalacja z paczek – dostępna w większości dystrybucji.
- Instalacja ze źródeł:
  - Kernel Source: http://kernel.org/pub/linux/kernel/v2.6/

UML

Instalacja UML

- Instalacja z paczek – dostępna w większości dystrybucji.
- Instalacja ze źródeł:
    - Kernel Source:
      http://kernel.org/pub/linux/kernel/v2.6/
- Wirtualnego systemu plików:

UML

Instalacja UML

- Instalacja z paczek – dostępna w większości dystrybucji.
- Instalacja ze źródeł:
  - Kernel Source: http://kernel.org/pub/linux/kernel/v2.6/
- Wirtualnego systemu plików:
  - Duży wybór dostępny na: http://uml.nagafix.co.uk/

UML

Instalacja ze źródeł

- Korzystamy z jądra 2.6.23.1

UML

Instalacja ze źródeł

- Korzystamy z jądra 2.6.23.1
- Ściągami plik linux-2.6.23.1.tar.bz2

UML

Instalacja ze źródeł

- Korzystamy z jądra 2.6.23.1
- Ściągami plik linux-2.6.23.1.tar.bz2
  - wget http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.23.1.tar.bz2

UML

Instalacja ze źródeł

- Korzystamy z jądra 2.6.23.1
- Ściągami plik linux-2.6.23.1.tar.bz2
  - wget http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.23.1.tar.bz2
- Rozpakowujemy plik ze źródłami np.:

UML

Instalacja ze źródeł

- Korzystamy z jądra 2.6.23.1
- Ściągami plik linux-2.6.23.1.tar.bz2
  - wget http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.23.1.tar.bz2
- Rozpakowujemy plik ze źródłami np.:
  - tar xjf linux-2.6.23.1.tar.bz2

UML

Instalacja ze źródeł

- Korzystamy z jądra 2.6.23.1
- Ściągami plik linux-2.6.23.1.tar.bz2
  - wget http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.23.1.tar.bz2
- Rozpakowujemy plik ze źródłami np.:
  - tar xjf linux-2.6.23.1.tar.bz2
  - cd linux-2.6.23.1

UML

Konfiguracja jądra

- Zapisana w pliku .config

UML

Konfiguracja jądra

- Zapisana w pliku .config
- Można zmieniać ręcznie.

UML

Konfiguracja jądra

- Zapisana w pliku .config
- Można zmieniać ręcznie.
- Wygodniej korzystać z gotowych konfiguratorów.

UML

Konfiguracja jądra

- Zapisana w pliku .config
- Można zmieniać ręcznie.
- Wygodniej korzystać z gotowych konfiguratorów.
- make defconfig ARCH=um

UML

Konfiguracja jądra

- Zapisana w pliku .config
- Można zmieniać ręcznie.
- Wygodniej korzystać z gotowych konfiguratorów.
- make defconfig ARCH=um
    - Przygotowuje domyślny plik konfiguracyjnych dla architektury user mode linux.

UML

Konfiguracja jądra

- Zapisana w pliku .config
- Można zmieniać ręcznie.
- Wygodniej korzystać z gotowych konfiguratorów.
- make defconfig ARCH=um
    - Przygotowuje domyślny plik konfiguracyjnych dla architektury user mode linux.
- make menuconfig ARCH=um

UML

Konfiguracja jądra

- Zapisana w pliku .config
- Można zmieniać ręcznie.
- Wygodniej korzystać z gotowych konfiguratorów.
- make defconfig ARCH=um
    - Przygotowuje domyślny plik konfiguracyjnych dla architektury user mode linux.
- make menuconfig ARCH=um
    - Uruchamia wygodny konfigurator tekstowy.

menuconfig - początek

```
.config - Linux Kernel v2.6.23.1 Configuration

                    Linux Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

          UML-specific options  --->
          General setup  --->
      [*] Enable loadable module support  --->
      --- Enable the block layer  --->
          Character Devices  --->
          Generic Driver Options  --->
          Networking  --->
          UML Network Devices  --->
      [*] Network device support  --->
      < > Connector - unified userspace <-> kernelspace linker  --->
      v(+)

             <Select>    < Exit >    < Help >
```

UML

Co powinniśmy ustawić?

- W menu głównym:

UML

Co powinniśmy ustawić?

- W menu głównym:
  - Enable loadable module support – włączamy obsługę modułów jądra.

UML

Co powinniśmy ustawić?

- W menu głównym:
    - Enable loadable module support – włączamy obsługę modułów jądra.
    - Network supporting – obsługę sieci.

UML

Co powinniśmy ustawić?

- W menu głównym:
    - Enable loadable module support – włączamy obsługę modułów jądra.
    - Network supporting – obsługę sieci.
    - Ciekawostka – dla chcących pisać zadanie nr 2 przy użyciu UML należy włączyć Crypto-API.

UML

Co powinniśmy ustawić?

- W menu głównym:
  - Enable loadable module support – włączamy obsługę modułów jądra.
  - Network supporting – obsługę sieci.
  - Ciekawostka – dla chcących pisać zadanie nr 2 przy użyciu UML należy włączyć Crypto-API.
  - Warto zainteresować się podmenu:
  - UML-specific options

UML

Co powinniśmy ustawić?

- W menu głównym:
  - Enable loadable module support – włączamy obsługę modułów jądra.
  - Network supporting – obsługę sieci.
  - Ciekawostka – dla chcących pisać zadanie nr 2 przy użyciu UML należy włączyć Crypto-API.
  - Warto zainteresować się podmenu:
  - UML-specific options
  - Kernel Hacking

UML

Co powinniśmy ustawić?

- W menu głównym:
  - Enable loadable module support – włączamy obsługę modułów jądra.
  - Network supporting – obsługę sieci.
  - Ciekawostka – dla chcących pisać zadanie nr 2 przy użyciu UML należy włączyć Crypto-API.
  - Warto zainteresować się podmenu:
  - UML-specific options
  - Kernel Hacking
- Po wybraniu każdego polecenia możemy wybrać help, aby dowiedzieć się czego dotyczy dana opcja.

UML-specific options

```
.config - Linux Kernel v2.6.23.1 Configuration

                         UML-specific options
 Arrow keys navigate the menu.  <Enter> selects submenus --->.
 Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
 <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
 for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

     [ ] Force a static link
         Host processor type and features  --->
         Host memory split (Default split (3G/1G user/kernel host spli
     [ ] Three-level pagetables (EXPERIMENTAL)
         Memory model (Flat Memory)  --->
     [ ] 64 bit Memory and IO resources (EXPERIMENTAL)
     [*] Networking support
     [*] Kernel support for ELF binaries
     <M> Kernel support for MISC binaries
     < > Host filesystem
     v(+)

              <Select>    < Exit >    < Help >
```

UML

UML-specific options

- Wybrane opcje:

UML

UML-specific options

- Wybrane opcje:
  - Host filesystems – umożliwia proste podmontowanie systemu plików hosta.

UML

UML-specific options

- Wybrane opcje:
  - Host filesystems – umożliwia proste podmontowanie systemu plików hosta.
  - Nesting level – ustawiamy na więcej niż 0, jeżeli chcemy uruchamiać tego UML wewnątrz innego UML.

UML

UML-specific options

- Wybrane opcje:
  - Host filesystems – umożliwia proste podmontowanie systemu plików hosta.
  - Nesting level – ustawiamy na więcej niż 0, jeżeli chcemy uruchamiać tego UML wewnątrz innego UML.
  - Kernel support for ELF binaries – obowiązkowo, obsługi binariów w formacie ELF.

Kernel Hacking

```
.config - Linux Kernel v2.6.23.1 Configuration

                         Kernel hacking
 Arrow keys navigate the menu.  <Enter> selects submenus --->.
 Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
 <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
 for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

     [*] Show timing information on printks
     [*] Enable __must_check logic
     [ ] Enable unused/obsolete exported symbols
     [ ] Debug Filesystem
     [*] Kernel debugging
     [ ]   Debug shared IRQ handlers
     [*]   Detect Soft Lockups
     [*]   Collect scheduler debugging info
     [ ]   Collect scheduler statistics
     [ ]   Collect kernel timers statistics
     v(+)

              <Select>    < Exit >    < Help >
```

UML

Kernel Hacking

- Wybrane opcje:

UML

Kernel Hacking

- Wybrane opcje:
  - Kernel Debugging

UML

Kernel Hacking

- Wybrane opcje:
  - Kernel Debugging Compile with debug info – skompilowanie z informacjami potrzebnymi gdb do pracy.

UML

Kernel Hacking

- Wybrane opcje:
  - Kernel Debugging Compile with debug info – skompilowanie z informacjami potrzebnymi gdb do pracy.
  - Show timing info on printk – wyświetlanie informacji w przypadku wywołania printk().

UML

Kernel Hacking

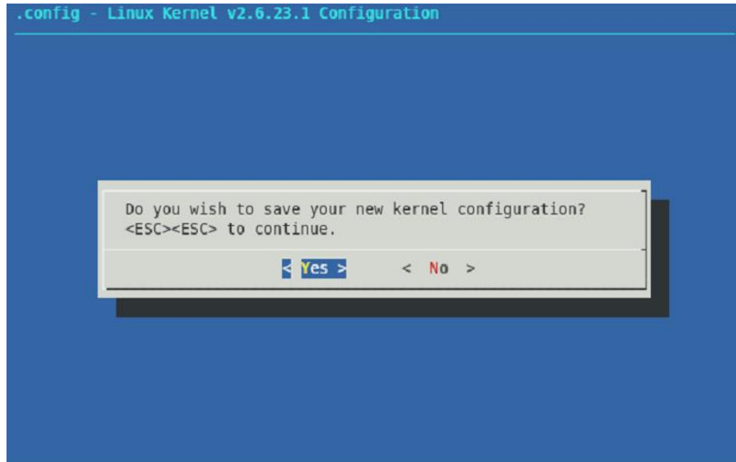
- Wybrane opcje:
  - Kernel Debugging Compile with debug info – skompilowanie z informacjami potrzebnymi gdb do pracy.
  - Show timing info on printk – wyświetlanie informacji w przypadku wywołania printk().
  - Detect soft lockup – wykrywanie sytuacji, gdy jądro nie dopuszcza pewnych procesów.

UML

Kernel Hacking

- Wybrane opcje:
  - Kernel Debugging Compile with debug info – skompilowanie z informacjami potrzebnymi gdb do pracy.
  - Show timing info on printk – wyświetlanie informacji w przypadku wywołania printk().
  - Detect soft lockup – wykrywanie sytuacji, gdy jądro nie dopuszcza pewnych procesów.
  - Opcje dotyczące mutexów, spinlocków itp. – pozwalają włączyć, w zależności od potrzeb wykrywanie pewnych zakleszczeń itp. ...

menuconfig - potwierdzenie



UML

Kompilacja jądra

- Jądro kompilujemy następującą komendą:
  - make linux ARCH=um

UML

Kompilacja jądra

- Jądro kompilujemy następującą komendą:
  - make linux ARCH=um
- Po poprawnym skompilowaniu jądra w katalogu linux-2.6.23.1 znajduje się plik linux, który jest uruchamialnym jądrem linuxa w formacie ELF.

UML

Kompilacja jądra

- Jądro kompilujemy następującą komendą:
  - make linux ARCH=um
- Po poprawnym skompilowaniu jądra w katalogu linux-2.6.23.1 znajduje się plik linux, który jest uruchamialnym jądrem linuxa w formacie ELF.
- Moduły kompilujemy następującą komendą:
  - make modules ARCH=um

UML

Instalacja modułów

- Ściągamy system plików np.:
  - Slackware-11-root_fs.bz2

UML

Instalacja modułów

- Ściągamy system plików np.:
  - Slackware-11-root_fs.bz2
- Rozpakowujemy do Slackware-11-root_fs.

UML

Instalacja modułów

- Ściągamy system plików np.:
  - Slackware-11-root_fs.bz2
- Rozpakowujemy do Slackware-11-root_fs.
- Montujemy go w systemie hosta:
  - mount slackware-11-root-fs /mnt/f –o loop

UML

Instalacja modułów

- Ściągamy system plików np.:
  - Slackware-11-root_fs.bz2
- Rozpakowujemy do Slackware-11-root_fs.
- Montujemy go w systemie hosta:
  - mount slackware-11-root-fs /mnt/f –o loop
- Instalujemy moduły na system plików:
  - Make modules_install INSTALL_MOD_PATH=/mnt/f ARCH=um

UML

Instalacja modułów

- Ściągamy system plików np.:
  - Slackware-11-root_fs.bz2
- Rozpakowujemy do Slackware-11-root_fs.
- Montujemy go w systemie hosta:
  - mount slackware-11-root-fs /mnt/f –o loop
- Instalujemy moduły na system plików:
  - Make modules_install INSTALL_MOD_PATH=/mnt/f ARCH=um
- Odmontowujemy wirtualny system plików:
  - Umount /mnt/f

UML

Uruchomienie UML

- Tak przygotowane jądro możemy uruchomić wydając komendę:
  - ./linux ubd0=Slackware-11-root_fs

UML

Uruchomienie UML

- Tak przygotowane jądro możemy uruchomić wydając komendę:
  - ./linux ubd0=Slackware-11-root_fs
- Możemy podać dodatkowe parametry np. mem=<ilość_pamięci>.

UML

Install własnego modułu

- Każdy moduł jest kompilowany pod konkretną wersję jądra.

UML

Install własnego modułu

- Każdy moduł jest kompilowany pod konkretną wersję jądra.
- Aby skompilować na hoście moduł do UML, należy wykonać:
  - make modules SUBDIRS=path ARCH=um

UML

Install własnego modułu

- Każdy moduł jest kompilowany pod konkretną wersję jądra.
- Aby skompilować na hoście moduł do UML, należy wykonać:
  - make modules SUBDIRS=path ARCH=um
- Path jest ścieżką do źródeł modułu.

UML

Install własnego modułu

- Każdy moduł jest kompilowany pod konkretną wersję jądra.
- Aby skompilować na hoście moduł do UML, należy wykonać:
  - make modules SUBDIRS=path ARCH=um
- Path jest ścieżką do źródeł modułu.
- Tak przygotowany moduł po wgraniu do vfs może być włączony na platformie UML (np. przez modprobe)

UML

UML + gdb

- Jądro uml jest traktowane przez gdb jak normalne proces, zatem uruchamiamy je dokładnie tak samo jak dowolny inny program.

UML

UML + gdb

- Jądro uml jest traktowane przez gdb jak normalne proces, zatem uruchamiamy je dokładnie tak samo jak dowolny inny program.
- Jądro UML używa wewnętrznie sygnałów SIGSEGV oraz SIGUSR1, zatem aby nie widzieć ciągłych komunikatów o nich piszemy:
  - handle SIGSEGV pass noprint nobreak
  - handle SIGUSR1 pass noprint nobreak

UML

UML - przykład pierwszy

- gdb ./linux

UML

UML - przykład pierwszy

- gdb ./linux
- handle SIGSEGV pass noprint nobreak

UML

UML - przykład pierwszy

- gdb ./linux
- handle SIGSEGV pass noprint nobreak
- handle SIGUSR1 pass noprint nobreak

UML

UML - przykład pierwszy

- gdb ./linux
- handle SIGSEGV pass noprint nobreak
- handle SIGUSR1 pass noprint nobreak
- b start_kernel

UML

UML - przykład pierwszy

- gdb ./linux
- handle SIGSEGV pass noprint nobreak
- handle SIGUSR1 pass noprint nobreak
- b start_kernel
- r ubd0=file_system

UML

UML - przykład pierwszy

- gdb ./linux
- handle SIGSEGV pass noprint nobreak
- handle SIGUSR1 pass noprint nobreak
- b start_kernel
- r ubd0=file_system
- Wykonanie jądra systemu zatrzyma się na funkcji start_kernel() – możemy np. podejrzeć jej kod przy pomocą komendy l

UML

UML - przykład drugi

- b schedule

UML

UML - przykład drugi

- b schedule
- r ubd0=file_system

UML

UML - przykład drugi

- b schedule
- r ubd0=file_system
- System uruchomi się i po zainicjowanie podstawowych struktur itp. zatrzyma się.

UML

UML - przykład drugi

- b schedule
- r ubd0=file_system
- System uruchomi się i po zainicjowanie podstawowych struktur itp. zatrzyma się.
- s – wykonanie schedule krok po kroku

UML

UML - przykład drugi

- b schedule
- r ubd0=file_system
- System uruchomi się i po zainicjowanie podstawowych struktur itp. zatrzyma się.
- s – wykonanie schedule krok po kroku
- c – po każdym continue jądro zatrzyma się niemal natychmiastowo – pokazuje to częstotliwość wywoływania funkcji schedule()

UML - przerwanie pracy jądra

Aby przejść do konsoli gdb bez ustawionego breakpointa należy:

- Uruchomić konsolę na hoście

UML - przerwanie pracy jądra

Aby przejść do konsoli gdb bez ustawionego breakpointa należy:

- Uruchomić konsolę na hoście
- Wykonać ps –A | grep linux

UML - przerwanie pracy jądra

Aby przejść do konsoli gdb bez ustawionego breakpointa należy:

- Uruchomić konsolę na hoście
- Wykonać ps –A | grep linux
- Wybrać najmniejszy pid procesu – jest to pid naszej instancji jądra UML

UML - przerwanie pracy jądra

Aby przejść do konsoli gdb bez ustawionego breakpointa należy:

- Uruchomić konsolę na hoście
- Wykonać ps –A | grep linux
- Wybrać najmniejszy pid procesu – jest to pid naszej instancji jądra UML
- Wykonać kill –INT <pid>

UML - przerwanie pracy jądra

Aby przejść do konsoli gdb bez ustawionego breakpointa należy:

- Uruchomić konsolę na hoście
- Wykonać ps –A | grep linux
- Wybrać najmniejszy pid procesu – jest to pid naszej instancji jądra UML
- Wykonać kill –INT <pid>
- Wykonanie jądra zostanie zatrzymane, a gdb pokaże informację o przechwyceniu sygnału

Breaking the schedule()

```
Welcome to Linux 2.6.23.1 (tty0)

darkstar login:
Program received signal SIGINT, Interrupt.
0xffffe410 in __kernel_vsyscall ()
(gdb) b schedule
Breakpoint 3 at 0x818e0bb: file kernel/sched.c, line 3482.
(gdb) c
Continuing.

Breakpoint 3, schedule () at kernel/sched.c:3482
3482            prev = rq->curr;
(gdb) c
Continuing.

Breakpoint 3, schedule () at kernel/sched.c:3482
3482            prev = rq->curr;
(gdb) c
Continuing.

Breakpoint 3, schedule () at kernel/sched.c:3482
3482            prev = rq->curr;
(gdb) c
Continuing.
```

UML - debugowanie modułów

- Trudność w debugowaniu modułów wynika z faktu, że moduły są dynamicznymi fragmentami kodu, zatem adres modułu w pamięci jest przydzielany dynamicznie przez jądro UML.

UML - debugowanie modułów

- Trudność w debugowaniu modułów wynika z faktu, że moduły są dynamicznymi fragmentami kodu, zatem adres modułu w pamięci jest przydzielany dynamicznie przez jądro UML.
- Dlatego musimy wyciągnąć ten adres, a potem podać go do gdb :)

UML - debugowanie modułu

Aby móc debuggować moduł należy:

- Załadować moduł do pamięci na maszynie UML (przez modprobe lub insmod).

UML - debugowanie modułu

Aby móc debuggować moduł należy:

- Załadować moduł do pamięci na maszynie UML (przez modprobe lub insmod).
- Pod gdb wypisać strukturę modułów
  - p modules

UML - debugowanie modułu

Aby móc debuggować moduł należy:

- Załadować moduł do pamięci na maszynie UML (przez modprobe lub insmod).
- Pod gdb wypisać strukturę modułów
  - p modules
- Otrzymamy listę linii w postaci :
  - $i = next = addr, prev = addr
  - np.: $1 = next = 0x3502cea4, prev = 0x3502cea4

UML - debugowanie modułu cz. 2

- Wypisujemy kolejne struktury modułów na ekran - p *((struct module *) addr)

UML - debugowanie modułu cz. 2

- Wypisujemy kolejne struktury modułów na ekran - p *((struct module *) addr)
- Otrzymujemy długą listę atrybutów. Interesują nas name – nazwa służąca do zidentyfikowania właściwego modułu oraz module_core – adres w pamięci uml.

UML - debugowanie modułu cz. 2

- Wypisujemy kolejne struktury modułów na ekran
  - p *((struct module *) addr)
- Otrzymujemy długą listę atrybutów. Interesują nas name – nazwa służąca do zidentyfikowania właściwego modułu oraz module_core – adres w pamięci uml.
- Dodajemy przestrzeń symboli modułu
  - add_symbol_file host_path_ko addr

UML - debugowanie modułu cz. 2

- host_path_ko to ścieżka do pliku modułu na maszynie hosta (!)

UML - debugowanie modułu cz. 2

- host_path_ko to ścieżka do pliku modułu na maszynie hosta (!)
- addr to wartosc pobrana z module_core

UML - debugowanie modułu cz. 2

- host_path_ko to ścieżka do pliku modułu na maszynie hosta (!)
- addr to wartosc pobrana z module_core
- Teraz możemy debuggować podobnie jak poprzednio – podając symbole (takie jak nazwy funkcji) do breakpointów, printów itp. ...

UML - umlgdb

- Istnieje skrypt umlgdb, który automatyzuje proces debuggowania modułów ładowanych do jądra uml.

UML - umlgdb

- Istnieje skrypt umlgdb, który automatyzuje proces debuggowania modułów ładowanych do jądra uml.
- Niestety, skrypt ten działa tylko z jądrem 2.4, dlatego nie opisuje tu jego działania.

UML - umlgdb

- Istnieje skrypt umlgdb, który automatyzuje proces debuggowania modułów ładowanych do jądra uml.
- Niestety, skrypt ten działa tylko z jądrem 2.4, dlatego nie opisuje tu jego działania.
- Zainteresowanych odsyłam do http://user-mode-linux.sourceforge.net/old/.

Zapraszamy na prezentację:

UML + gdb oraz kgdb w praktyce