

термоядерная отладка в Linux и xBSD

обзор отладчиков ядерного уровня

крис касперски, aka мышцх, a.k.a. nezumi, a.k.a. souriz, a.k.a. elraton, no-email

отладчиков уровня ядра под никсы — много, хороших из них мало (если такие вообще есть) и нужно быть нереально крутым хакером, чтобы с первого напаса выбрать такой дебагер, которым можно отлаживать, а не обламываться. перепробовав кучу отладчиков, мышцх решил составить внятный обзор для начинающих, рассказывающий чем один отладчик отличается от другого и какой из них торкает, а какой никуда не канает кроме как в /dev/nul

введение

Существует множество интегрированных отладчиков для Linux'a, но ни один из них не включен в основную ветвь ядра, что выглядит странно, если не сказать подозрительно, особенно если учесть, что xBSD-системы включают в себя ядерный отладчик изначально (правда, во всех известных мне дистрибутивах он по умолчанию задисаблен и его активация требует перекомпиляции ядра).

Причина в том, что Линус Торвальдс (до сих пор стоящий у руля и принимающий решение о включении тех или иных компонентов в ядро) не доверяет интерактивным отладчикам и считает, что у "правильных" программистов таких потребностей просто не возникает. Типа, есть же отладочная печать (см. `map printk`) и ее, типа, вполне достаточно.

С Линусом, однако, согласны далеко не все разработчики и в определенных ситуациях без дебагера не обойтись, особенно если приходится отлаживать чужие модули, поставляемые без исходных текстов или ломать защиты, противостоящие отладчикам прикладного уровня. Так что, хочет того Торвальдс или нет, но "термоядерные" отладчики для Линуса все-таки есть, причем в количестве намного большим одного, причем, практически все они распространяются в исходных текстах и не требуют денег. Казалось бы какая проблема — скачал, поставил, запустил....

Между тем, проблемы все-таки есть. Это и плохая совместимость неофициальных отладчиков в различными версиями официальных ядер и сложность выбора хорошего отладчика из кучи заброшенных проектов... Ситуация усугубляется тем, что различные отладчики предназначены для решения различных задач и потому на вопрос: "какой ядерный отладчик самый лучший" даются сильно неодинаковые ответы, зачастую без всяких пояснений! Ну и какая польза от таких советов?!

типы отладчиков

Классические ядерные отладчики (как для UNIX, так и для Windows) требуют наличия `_двух_` компьютеров, на одном из которых устанавливается отлаживаемое ядро, а на другом — сам отладчик. Обмен данными обычно осуществляется по последовательному порту через нуль-модем, хотя можно встретить и другие варианты. Такие отладчики называются **удаленными** и к ним, в частности, относится знаменитый `gdb` (клиентская часть). Другая часть отладчика находится непосредственно в ядре и если там ее нет (а Linux'e ее нет), у нас ничего не получится. Очевидный недостаток удаленных отладчиков — необходимость приобретения второго компьютера, что далеко не всегда приемлемо (особенно для "домашних" хакеров). Виртуальные машины в какой-то мере снижают остроту проблемы, но...

Локальные отладчики выгодно отличаются тем, что позволяют отлаживать ядро на одной машине с отладчиком. Чаще всего они работают только в текстовом режиме. Поддержка консоли в графическом режиме (не говоря уже про X'ы) требует специальных "агентов", работающих далеко не везде и не всегда, а потому в ряде случаев приходится прибегать к удаленной отладке.

Конструктивно отладчики могут быть реализованы либо как неофициальный патч ядра (требующий его перекомпиляции), либо как драйвер, загружаемый в ядро налету зачастую даже без перезагрузки системы. Примером отладчиков первого типа служит `KDB`, второго — `LinIce`.

Однако, независимо от особенностей своей реализации, все Linux-отладчики страдают хронической проблемой совместимости с ядрами, поскольку, разработчики ядра не координируют свои действия с разработчиками отладчиков и потому последние поддерживают

только фиксированный набор версий ядер, причем, поддержка новых версий как правило происходит с большим опозданием.

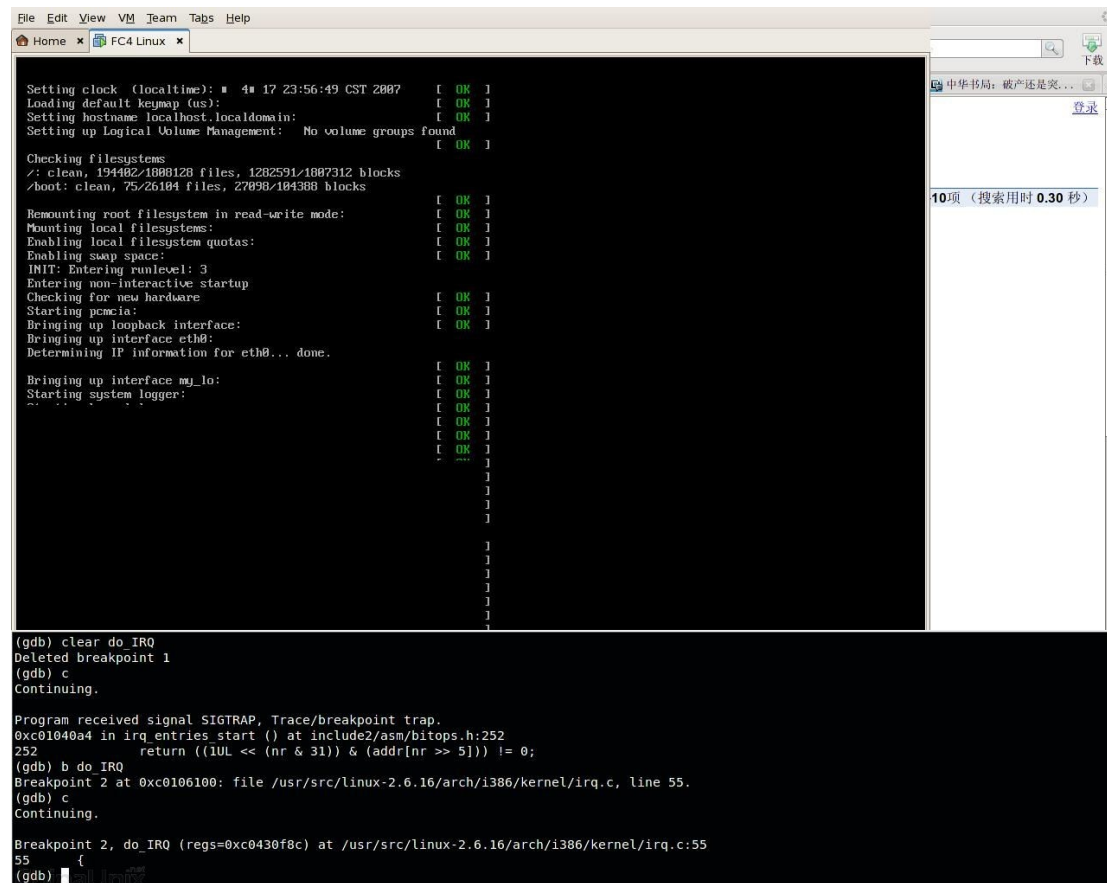
У xBSD таких проблем вообще по жизни нет, поскольку ядерный отладчик разрабатывается (и поставляется) вместе с самими ядрами, и нормально поддерживает все графические режимы, в которых только работает сама xBSD.

использование отладочных возможностей VMWare

Начиная с версии 6.0 RC2 в популярной виртуальной машине VM Ware появился механизм Record/Replay, позволяющий (среди прочих возможностей) осуществлять удаленную ядерную отладку даже для тех операционных систем, которые поставляются без интегрированного отладчика: Linux, xBSD с выключенным отладчиком, NT, etc.

Просто добавляем в vmx-файл (описывающий конфигурацию виртуальной машины) строку "debugStub.listen.guest32=1" (или "debugStub.listen.guest64=1" для 64-разрядных платформ), после чего в vmware.log файле появляется следующая запись "VMware Workstation is listening for debug connection on port 8832", означающая, что виртуальная машина слушает 8832-порт с которым готова общаться по gdb-протоколу. Остается запустить сам gdb, приконнектиться к порту и приступить к отладке безо всяких танцев с бубном, без наложения заплаток, без перекомпиляции ядра, etc.

Отладчик gdb может быть запущен как на хосте (основной операционной системе), так и на соседней виртуальной машине.



```
File Edit View VM Team Tabs Help
Home x FC4 Linux x
Setting clock (localtime): 4 17 23:56:49 CST 2007 [ OK ]
Loading default keymap (us): [ OK ]
Setting hostname localhost.localdomain: [ OK ]
Setting up Logical Volume Management: No volume groups found [ OK ]
Checking filesystems
/: clean, 194462/1888128 files, 1282591/1887312 blocks
/boot: clean, 75/26184 files, 27898/104388 blocks
Remounting root filesystem in read-write mode: [ OK ]
Mounting local filesystems: [ OK ]
Enabling local filesystem quotas: [ OK ]
Enabling swap space: [ OK ]
INIT: Entering runlevel: 3
Entering non-interactive startup
Checking for new hardware [ OK ]
Starting pcmcia: [ OK ]
Bringing up loopback interface: [ OK ]
Bringing up interface eth0:
Determining IP information for eth0... done.
Bringing up interface my_lo: [ OK ]
Starting system logger: [ OK ]
...
(gdb) clear do_IRQ
Deleted breakpoint 1
(gdb) c
Continuing.
Program received signal SIGTRAP, Trace/breakpoint trap.
0xc01040a4 in irq_entries_start () at include2/asm/bitops.h:252
252 return ((1UL << (nr & 31)) & (addr[nr >> 5])) != 0;
(gdb) b do_IRQ
Breakpoint 2 at 0xc0106100: file /usr/src/linux-2.6.16/arch/i386/kernel/irq.c, line 55.
(gdb) c
Continuing.
Breakpoint 2, do_IRQ (regs=0xc0430f8c) at /usr/src/linux-2.6.16/arch/i386/kernel/irq.c:55
55 {
(gdb)
```

Рисунок 1 использование отладочных возможностей VM Ware 6.0 для отладки Linux'a без интегрированного отладчика

Для отладки нам потребуется два комплекта ядер. Одно – установленное на отлаживаемой системе и другое – установленное на машине (реальной или виртуальной) где работает gdb. Отлаживаемое ядро может быть скомпрессировано и пострипано (как обычно и бывает), а вот ядро для gdb в обязательном порядке должно быть разжато (ну не понимает gdb сжатых ядер — что тут поделаешь!) и желательно откомпилировано с отладочной

информацией. Как минимум, на машине с gdb должен присутствовать файл System.map. Естественно, версии обоих ядер должны совпадать, иначе начнется полный хаос.

Пара примеров работы с gdb представлена ниже:

```
# запускаем gdb
% gdb

# указываем путь к несжатому 32-битному ядру
(gdb) file vmlinux-2.4.69-27.EL.debug

# подключаемся к отлаживаемому ядру
(gdb) target remote localhost:8832

# все! с этого момента можно начинать
# отладку ядра!!!
```

Листинг 1 отладка x86-ядра Linux'a под VM Ware

```
# запускаем gdb
% gdb

# указываем путь к несжатому 64-битному ядру
(gdb) file vmlinux-2.6.96-17.EL.debug

# переводим gdb в 64-разрядный режим
(gdb) set architecture i386:x86-64

# подключаемся к отлаживаемому ядру
(gdb) target remote localhost:8832

# все! с этого момента можно начинать
# отладку ядра!!!
```

Листинг 2 отладка x86-64 ядра Linux'a под VM Ware

Естественно, ядро запущенное на эмуляторе, "видит" только виртуальное железо (исключение составляют USB-устройства, жесткие диски и сетевые карты) к которым VM Ware позволяет давать прямой доступ, однако, например, отладить драйвер видео-карты таким образом уже не получится.

Подробнее на эту тему можно почитать: <http://stackframe.blogspot.com/> и <http://blogs.vmware.com/sherrod/2007/04/index.html>, а триальную версию VMWare скачать — <http://www.vmware.com/download/ws>

использование отладочных возможностей QEMU

Эмулятор QEMU так же позволяет отлаживать ядра без интегрированных отладчиков, но, в отличие от VM Ware, он бесплатен и распространяется вместе с исходными тестами, которые находятся на <http://fabrice.bellard.free.fr/qemu/>

Пример командной строки, реализующий форсированную отладку, приведен ниже:

```
# запускаем QEMU с ядром, которое мы собираемся отлаживать
$ qemu -kernel /boot/bzImage -append "root=/dev/hda" -std-vga -m 256m -s -hda hdd.img &

# запускаем gdb на основной машине и подключаемся на порт 1234
$ gdb (gdb) target remote localhost:1234

# подключаем образ ядра (должен совпадать с отлаживаемым ядром)
(gdb) file vmlinux
```

Листинг 3 отладка Linux'a без интегрированного отладчика под QEMU

```
memory: 08000000 @ 00000000 (usable)
Initial ramdisk at: 0x80800000 (3586098 bytes)
Built 1 zonelists. Total pages: 32768
Kernel command line: rd_start=0x80800000 rd_size=3586098 root=/dev/ram console=ttyS0
Primary instruction cache 2kB, physically tagged, 2-way, linesize 16 bytes.
Primary data cache 2kB, 2-way, linesize 16 bytes.
Synthesized TLB refill handler (19 instructions).
Synthesized TLB load handler fastpath (31 instructions).
Synthesized TLB store handler fastpath (31 instructions).
Synthesized TLB modify handler fastpath (30 instructions).
PID hash table entries: 1024 (order: 10, 4096 bytes)
Using 100.000 MHz high precision timer.
Console: colour VGA+ 80x25
Dentry cache hash table entries: 16384 (order: 4, 65536 bytes)
Inode-cache hash table entries: 8192 (order: 3, 32768 bytes)
Memory: 118224k/131072k available (2101k kernel code, 12832k reserved, 450k
data, 132k init, 0k highmem)
Security Framework v1.0.0 initialized
SELinux: Disabled at boot.
Capability LSM initialized
Mount-cache hash table entries: 512
Checking for 'wait' instruction... available.
checking if image is initramfs...
```

Рисунок 2 загрузка Linux-ядра на виртуальной машине бесплатного эмулятора QEMU

Novell Linux Kernel Debugger (NLKD)

На сегодняшний день, NLKD является, пожалуй, самым продвинутым и мощным ядерным отладчиком для Linux, поддерживающим как локальную, так и удаленную отладку. Другие его достоинства — бесплатность и наличие исходных текстов. К сожалению, он работает только с **SUSE Linux Enterprise Server v9 SP1/SP2** и требует перекомпиляции ядра, что является существенным недостатком, ограничивающим область его применения. Зато NLKD имеет документированный расширяемый интерфейс плагинов и свободно работает в как в текстовом, так и в графическом режиме.

Скачать последнюю версию отладчика (вместе с документацией) можно по ссылке: <http://forge.novell.com/modules/xfmod/project/?nlkd>

```

schedule:
%0120760 push    ebp
%0120761 mov     ebp, esp
%0120763 push    edi
%0120764 push    esi
%0120765 push    ebx
%0120766 sub     esp, 3C
%0120769 mov     eax, ldump_oncpu1
%0120770 jnz     %C0121019
%0120776 mov     edx, FFFFE000
%0120778 and     esp, edx
%012077D cmp     [edx], edx
%012077F mov     eax, edx
%0120781 jz      %C0121009
%0120787 mov     eax, [eax]
%0120789 mov     eax, [eax]
%012078B test    al, 19
%012078D jnz     %C0120A55
%0120793 mov     eax, FFFFE000
%0120798 and     eax, esp
%012079A cmp     leax1, eax
%012079C mov     edx, eax
%012079E jz      %C0121011
%01207A4 mov     eax, [edx+14]
%01207A7 test    eax, eax
%01207A9 jz      %C0120A55
%01207AF jmp     %C012103E
%01207B4 call   get_default_cpu_class
%01207B9 mov     ecx, [eax+00000080]
%01207BF mov     eax, [per_cpu_runqueues]
%01207C4 test    eax, eax
%01207C6 jz      %C0120F75
%01207CC mov     edx, [ecx+28]

st(4) <empty>      st(1) <empty>      st(2) <empty>      st(3) <empty>
st(5) <empty>      st(6) <empty>      st(7) <empty>
xmm0_32 0          xmm0_32 0          xmm0_32 0          xmm0_32 0
xmm1_32 0          xmm1_32 0          xmm1_32 0          xmm1_32 0
xmm2_32 0          xmm2_32 0          xmm2_32 0          xmm2_32 0
xmm3_32 0          xmm3_32 0          xmm3_32 0          xmm3_32 0

%04D1F5C 00000046 00000000 00000001 C04D1F74 C0129C05 C04C8200 C04D1F98 C010AFCA esp 00000046
%04D1F5E C7E80D00 C045DA00 00000001 C04E129C C04D0000 00099100 C050E560 C04D1FD4 +04 00000000
%04D1F9C C01090EC C04D0000 00000000 C04D0000 00099100 C050E560 C04D1FD4 00000000 +08 00000000
%04D1FB6 0000007B 0000007B FFFFFFF0 C010634C 00000000 00000246 C04D1FE0 C0107184 +0C C04D1F74
%04D1FDC 00000000 C04D1F7B C04D265B C03C13C4 C04D2139 00000000 C050E560 005A9007 +10 C0129C05
%04D1FFC C010019F 0001B855 E5890000 8505C65D 00C045F7 900DEBC3 90909090 90909090 +14 C04C8200
%04D201C 90909090 0001B855 E5890000 DE7CA35D 01B8C050 C3000000 0000B68D BF3D0000 +18 C04D1F58
%04D2038 00000000 0F5925 05C0C020 C0451786 0EC83000 5DC0220F 000001B8 B3BD3300 +1C C019AF7A
%04D205C 00000000 89D23155 01B8DE5E 89000000 45EA2C15 B4BDC3C0 00000026 27BCD000 +20 C7E80D00
%04D207C 00000000 89D23155 0038B0E5 0AB91075 BA000000 00000001 15A40D83 895DC046 +24 C045DA00
%04D2098 F689C3D0 89D23155 0038B0E5 04B80F75 BA000000 00000000 4615A4A3 D0895DC0 +28 00000001
%04D20BC 0076BDC3 A3E58955 C05D0003 000001B8 0076B000 148D9231 45EA4085 F88340C0 +2C C04E129C
%04D20D8 5DF1761F 000001B8 76BDC300 27BCD000 00000000 0001B955 E5890000 0001B85D +30 C04D0000
%04D20FC 0D890000 C04FB068 A2748DC3 27BCD000 00000000 EA2CB855 E589C045 8904EC83 +34 00099100

bn
?
bc hd be h1 hoot hpa hpi hpio hpo hpr hprw
bpx cls e code cpu d da du da
ef4 es fa fi f2 f4 f8 mod o ol of ef4 ef8 ef8 ef8
hboot i il i2 i4 mod o ol of ef4 ef8 ef8 ef8
bpi r radix rs st su tss sa su si s2 s4 s4 s4 s8
sfb scr size stack st su tss sa su si s2 s4 s4 s4 s8
>

```

Рисунок 3 внешний вид отладчика NLKD

отладчик KDB

KDB – самый популярный ядерный отладчик для Linux-систем, разработанный компанией SGI, распространяющий его в исходных текстах на бесплатной основе как побочный продукт своей деятельности — <http://oss.sgi.com/projects/kdb/>

Отладчик реализован как патч к ядру версий 2.[234], устанавливаемый следующим образом:

```

$ cd linux
$ patch -p1 < kdb-xxx
$ make *config

```

Листинг 4 установка отладчика KDB путем патчка исходных текстов ядра

Активируем флаги CONFIG_KDB и CONFIG_FRAME_POINTER (что можно сделать при помощи утилит типа xconfig, menuconfig, oldconfig или ручного задания опций make-файла), перекompилируем ядро и радуемся жизни.

KDB поддерживает локальный/удаленный режимы отладки и работает на процессорах семейства x86 и IA64, однако, в режиме локальной отладки имеет большие проблемы с различными графическими режимами и некоторыми контроллерами клавиатурами, что не есть хорошо, поэтому использование NLKD в ряде случаев оказывается все же предпочтительнее.

Переход в текстовый режим осуществляется по ALT-CTRL-F1, а возвращение — ALT-CTRL-F7. Клавиша <Pause> вызывает всплытие локальной консоли отладчика (аналог <CTL-D> в soft-ice), позволяя нам просматривать память, устанавливать точки останова, дизассемблировать машинный код и т.д. Короче, практически все как в soft-ice, вот только отладки на уровне исходных текстов KDB не представляет, что является существенным недостатком для обычных разработчиков, но нас, хакеров, совершенно не волнует, поскольку, если бы у нас были исходные тексты, разве стали бы мы заниматься онанизмом и часами прочитывать в отладчике?!

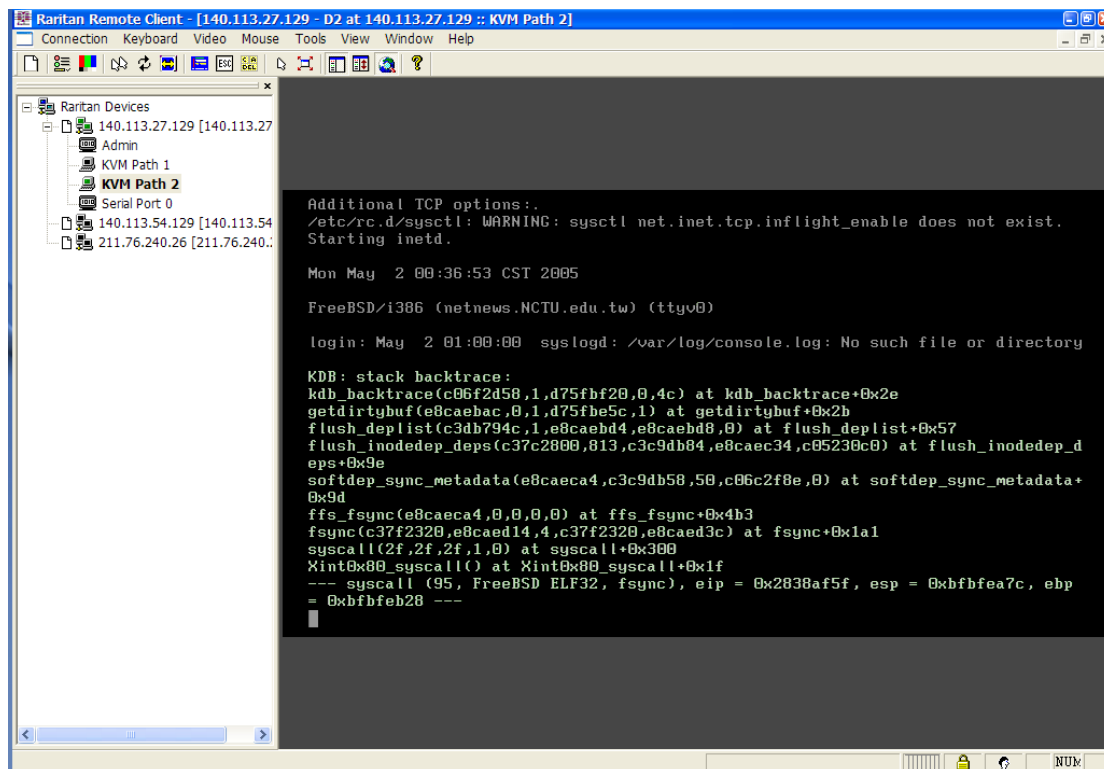


Рисунок 4 сеанс удаленной отладки в KDB

отладчик KGDB

Читая обзоры, можно подумать, что KGDB – это конкурент (или, если не конкурент, то альтернатива) KDB. На самом деле это не так и KGDB представляет собой интегрированный отладчик удаленного (не локального!!!) типа, поддерживающий несколько версий ядер от 2.4.6 до 2.6.0 и реализованный на платформах i386, x86_64, PPC и S390.

KGDB устанавливается путем патча ядра с последующей перекомпиляцией и реализует только серверную часть. В качестве клиента обычно используется gdb или другой отладчик, поддерживающий его нуль-модемный протокол.

Если на удаленной машине (там, где находится gdb) положить ядро, откомпилированное с отладочной информацией (ключ -g), мы получим отладку на уровне исходных текстов, обеспечиваемую средствами gdb, а отнюдь не KGDB, как утверждают некоторые руководства.

Скачать последнюю версию отладчика можно с <http://kgdb.linsyssoft.com/>

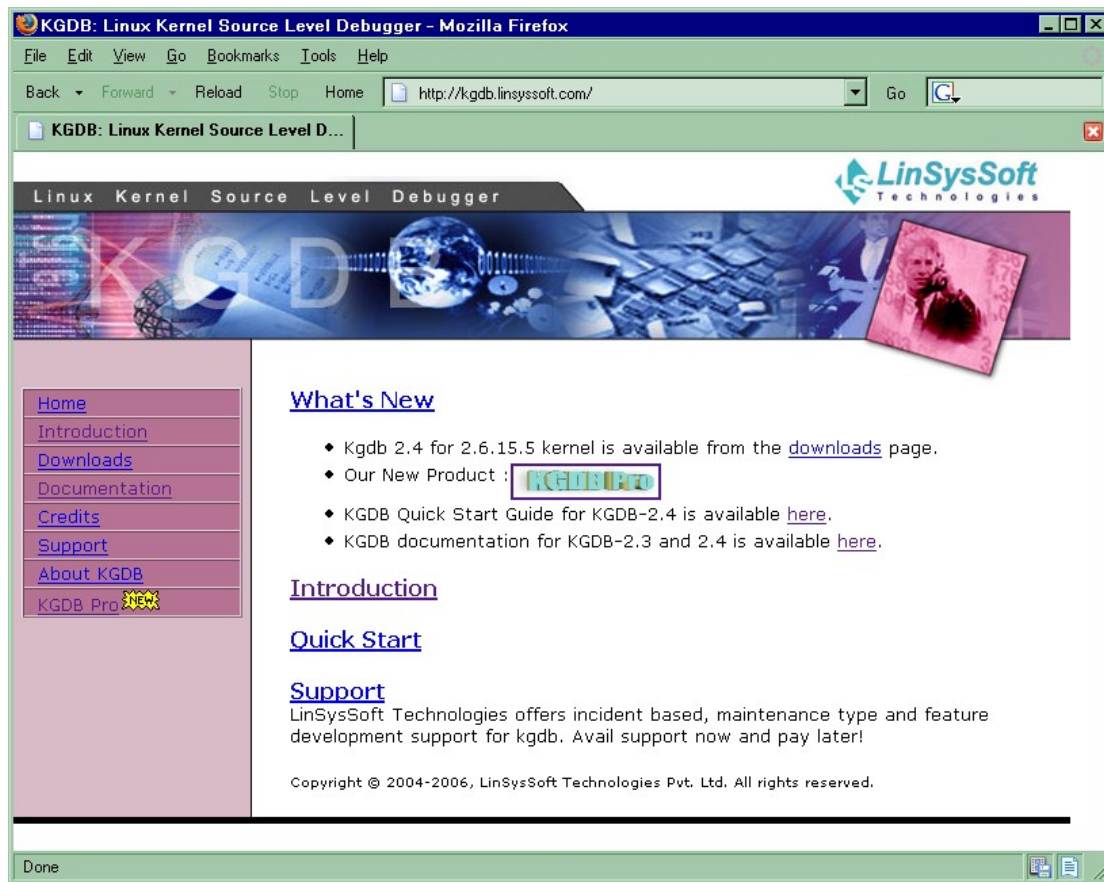


Рисунок 5 здесь можно надыбать KGDB

отладчик LinIce

LinIce представляет собой некоторую пародию на soft-ice под Linux, конструктивно реализованную как загружаемый модуль ядра, не требующую ни наличия второй машины, ни перекомпиляции, что дает ему сто очков вперед. К сожалению, у LinIce имеется множество проблем. Ядра 2.6.9 и выше не поддерживаются, как не поддерживаются Super-VGA и framebuffer режимы. К тому же имеется множество проблем с различными клавиатурными контроллерами...

Тем не менее, LinIce вполне пригоден для хакерства, особенно если необходимо что-то быстро отломать, а времени/желания/возможности перекомпиляции ядра у нас нет. Однако, следует помнить, что по своим функциональным возможностям LinIce самый бедный отладчик, из всех рассмотренных выше и потому для серьезного хакинга все-таки лучше поставить NLKD или KDB.

Последнюю версию отладчика можно бесплатно скачать с <http://www.linice.com>.

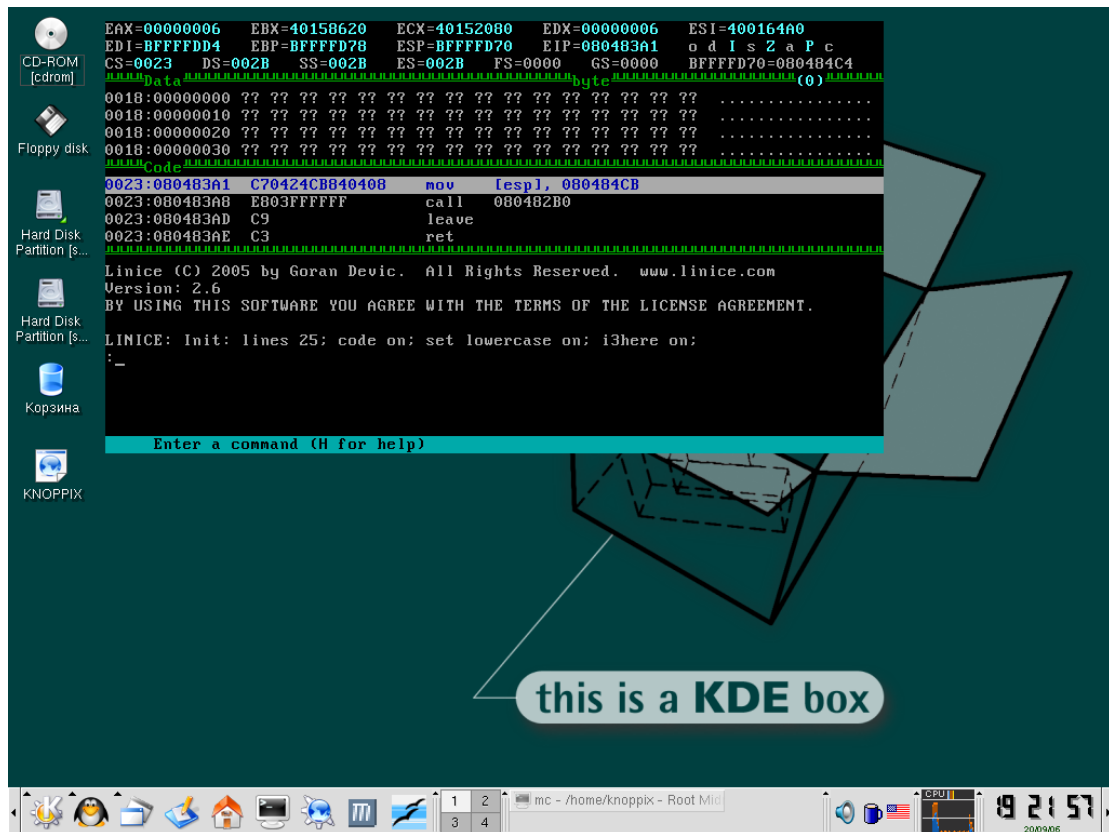


Рисунок 6 внешний вид отладчика LinIce

ядерная отладка в xBSD системах

Ядро xBSD систем включает в себя интегрированный отладчик по имени DDB, поддерживающий как локальную, так и удаленную отладку. По умолчанию ядро собирается без отладчика и чтобы исправить эту вопиющую несправедливость необходимо добавить строку "options DDB" в файл конфигурации ядра (обычно находится в /usr/src/sys/i386/conf/GENERIC) и перекомпилировать его.

Вызывать отладчик можно различными путями: набрав флаг "-d" в приглашении загрузчика мы попадем в DDB на самой ранней стадии загрузки ядра до начала распознавания и подключения устройств, что очень полезно для отладки драйверов.

А вот если хочется взломать какую-нибудь программу, то для вхождения в DDB с консоли (как текстовой, так и графической) достаточно отдать команду "sysctl debug.enter_debugger=ddb" или (если в конфигурационном файле обозначена опция "options BREAK_TO_DEBUGGER") нажать на <CTRL-ALT-ESC>. (**Внимание:** в некоторых раскладках клавиатуры эта комбинация изменена!)

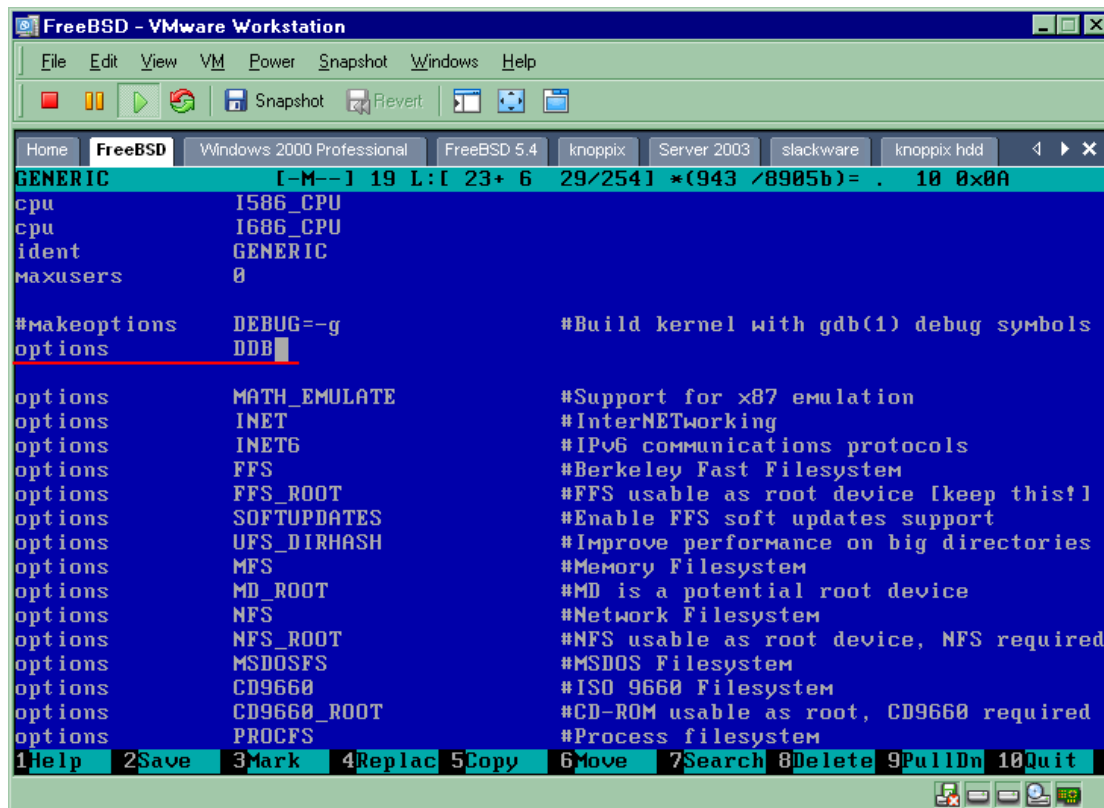


Рисунок 7 изменение конфигурации ядра FreeBSD для подключения отладчика DDB

При желании (и наличии второй машины) можно осуществить удаленную отладку, если возможностей локальной вдруг окажется недостаточно (мне трудно представить такую ситуацию, но... чего в жизни не случается). Официальная документация по FreeBSD говорит, что для этого нам понадобится две копии ядра: одна — установленная на отлаживаемой системе (пострипанная), другая — удаленная, положенная в один каталог с gdb (откомпилированная с отладочной информацией). На самом деле, наличие отладочной информации совершенно необязательно, gdb будет работать и без нее, пускай и не сможет выйти на уровень исходных текстов — а оно нам нужно? Достаточно, чтобы версии ядер совпадали. Более того, в качестве удаленной системы не обязательно использовать именно FreeBSD. Пригодна любая система, под которую имеется порт gdb (например, Linux, или даже NT), главное — скопировать копию ядра на удаленную систему и загрузить ее в gdb через команду "file".

Но прежде, чем запускать gdb необходимо соединить обе машины COM-шнурком, причем в файле конфигурации ядра потребуется исправить строку, отвечающую за параметры данного порта (она находится в строке "device siox", где x — номер последовательно порта, считая от нуля), оставив флаги в значении 0x80.

Включаем отлаживаемую машину. В командной строке загрузчика набираем "-d", чтобы остановить загрузку системы и войти в отладчик. Включаем удаленную машину и набираем в командной строке "gdb -k kernel" (или же "kgdb kernel" при использовании KGDB), где "kernel" — имя (с путем) к файлу ядра.

Цепляемся отладчиком к последовательному порту, набрав следующую команду "target remote /dev/cuaa0", где cuaa0 — первый последовательный порт, после чего возвращаемся к отлаживаемой машине (которая находится в состоянии ожидания загрузки внутри DDB) и говорим "gdb", сообщая системе, что мы передаем бразды правления удаленному отладчику gdb. Вернуть управление обратно можно при помощи той же самой команды "gdb", фактически представляющий собой триггер между локальным и удаленным режимом отладки.

Заключение

Разумеется, мы описали далеко не все существующие ядерные отладчики и оставили без ответа вопрос: какой же из них все-таки самый лучший. Но ведь отладчик — это не религия. Использование нескольких отладчиков вполне нормальное явление.

Лично мышкх предпочитает такую стратегию: если ломаемая программа запускается под FreeBSD 4.5 (любимая версия мышкх'а), то задействуется DDB, если же нет, то загружается виртуальная машина с SuSE Linux, где установлен NLDK. Под остальными системами приходится использовать KDB или же KGDB, соединенный с соседней виртуальной машиной, на которой запущен gdb.

LinIce используется в основном для несложных экспериментов (например, наблюдением за rootkit'ми и прочей малварью). Ломать программы в нем мышкх перестал как только въехал в gdb, рядом с которым soft-ice и рядом не валялся.

Отладочные возможности эмуляторов мышкх пока еще не раскурил до конца и только начинает их использовать, но чем больше он их использует, тем больше они ему нравятся.

>>> врезка магические SysRq клавиши

Ядро Linux'а, начиная с версии 2.2 поддерживает ряд магических комбинаций клавиш, вызываемых по <ALT-SysRq-Key>, где "Key" – магическая клавиша, например, <s>. Магические клавиши полезны для отладки и борьбы с малварью (например, клавиша <k> – убивает все процессы в текущей консоли).

Чтобы включить магические SysRq клавиши при компиляции ядра параметр CONFIG_MAGIC_SYSRQ необходимо установить в состояние "yes" (в большинстве дистрибутивов это уже сделано за нас, если же это не так, а перекомпилировать ядро лень, то включить магические клавиши можно и на лету, отдав команду "echo 1 > /proc/sys/kernel/sysrq").

```

Enabling DMA acceleration for: hdc      [UMware Virtual IDE CDROM Drive]
Scanning for Harddisk partitions and creating /etc/fstab... Done.
Using swap partition /dev/sda2.
Network device eth0 detected, DHCP broadcasting for IP. (Backgrounding)
Automounter started for: floppy cdrom.
(Re)starting network services.
SysRq : Show State

      free      sibling
task   PC  stack  pid father child younger older
init   S C1195F24 3216 1 0 370 (NOTLB)
Call Trace: [c0118329] [c011825c] [c0144e2c] [c014e23d] [c0108997]
keventd S C7Ea2664 5880 2 1 3 (L-TLB)
Call Trace: [c0129165] [c0107114]
ksoftirqd_CPU S C7E9E000 6152 3 1 4 2 (L-TLB)
Call Trace: [c012093c] [c0107114]
kswapd S C7E96000 6360 4 1 5 3 (L-TLB)
Call Trace: [c011e99e] [c011e6ef] [c0135d26] [c0107114]
bdflush S 00000286 6308 5 1 6 4 (L-TLB)
Call Trace: [c0118c47] [c0142503] [c0107114]
kupdated S C7E93F34 5620 6 1 57 5 (L-TLB)
Call Trace: [c08b1e5ca] [c0118329] [c08c3d020] [c011825c] [c01425dd]
[c010899a] [c0142508] [c0107114]
kjournald S 00000286 0 57 1 62 6 (L-TLB)
Call Trace: [c0118c47] [c08b1df9] [c08b1de10] [c0107114]
init S 00000000 0 62 1 63 140 57 (NOTLB)
Call Trace: [c011f38c] [c011f3cf] [c0108997]
rcS S 00000000 0 63 62 371 (NOTLB)
Call Trace: [c011f38c] [c011f3cf] [c0108997]
khubd S 00000001 4 140 1 370 62 (L-TLB)
Call Trace: [c08bf080b] [c08bf9cac] [c08bf9cac] [c0107114]
automount S 7FFFFFFF 24 370 1 140 (NOTLB)
Call Trace: [c01182cb] [c014e4e6] [c014e51b] [c014e72d] [c0108997]
S01knoppix-hd S 00000000 0 371 63 457 (NOTLB)
Call Trace: [c011f38c] [c011f3cf] [c0108997]
sleep S C6833F88 B 457 371 (NOTLB)
Call Trace: [c012cbbf] [c0118329] [c011825c] [c012481e] [c0108997]
Starting daemons...
Cleaning: /etc/network/ifstate.
Setting up IP spoofing protection: rp_filter.
Configuring network interfaces...done.
Starting printing system service: cupsys.
Mounting local filesystems...
mount: usbdevfs already mounted or /proc/bus/usb busy
mount: according to mtab, /proc/bus/usb is already mounted on /proc/bus/usb
mount: fs type sysfs not supported by kernel
INIT: Entering runlevel: 5

```

Рисунок 8 вывод списка процессов при помощи магической комбинации <ALT-SysRq-t>

Перечень магических клавиш лежит в файле /src/Documentation/sysrq.txt, а так же может быть получен по ссылке: <http://www.gelato.unsw.edu.au/lxr/source/Documentation/sysrq.txt>

- 'r' – отключение сырого клавиатурного ввода;
- 'k' —процедура "Secure Access Key" (сокращенно — SAK), убивающая все процессы на текущей виртуальной консоли;
- 'b' — немедленная перезагрузка без размонирования дисков;
- 'c' — создание crashdump'а, пригодного для последующего анализа;
- 'o' — нормальный шатдаун системы;

- 's' — сброс дисковых кэшей для всех смонтированных томов;
- 'u' — перемонтирование всех томов "только-на-чтение";
- 'p' — отображение содержимого регистров процессора;
- 't' — отображение текущего списка процессов;
- 'm' — отображение об использовании памяти;
- '0'-'9' — установка уровня отладочного вывода printk;
- 'e' — посылка сигнала SIGTERM всем процессам кроме init;
- 'i' — посылка сигнала SIGKILL всем процессам кроме init;
- 'I' — посылка сигнала SIGKILL всем процессам, включая init;
- 'h' — вывод списка магических SysRq клавиш;