Odpluskwianie jądra Linuksa

Bartłomiej Bonarski, Kamil Lenartowicz, Paweł Zaborski

21 listopada 2007

Spis treści

- Podstawy debugowania jądra
- Omówienie KDB, KGDB
- User Mode Linux

Problemy

Pisanie programów nie jest łatwe. Podobnie jest z pisaniem kodu jądra. Trudności przy odpluskwianiu jądra:

- **wielowątkowość** jądra
- ciężko jest powtórzyć sytuację z błędem
- brak jakiejkolwiek kontroli nad jądrem, które samo kontroluje system

Błąd w trakcie działania jądra

- sygnalizacja błędu
- wykonanie akcji obsługi błędu
- powstaje błąd oops
- wiadomość trafia do bufora cyklicznego jądra (można go obejrzeć np za pomocą **dmesg**)

oops - zdjęcie

Video Session Viewer

```
Oops: 0000 [#1]
SMP
Modules linked in: iptable_nat
CPU:    1
EIP:    0060:[<c0181b16>]    Not tainted VLI
EFLAGS: 00210286   (2.6.13.4)
EIP is at do_task_stat+0x1f6/0x5e0
eax: 03200005   ebx: c28d3000   ecx: e003dc80   edx: 00000004
esi: e6790a20   edi: f6690300   ebp: c66d5f18   esp: c66d5ea4
ds: 007b   es: 007b   ss: 0068
Process ps (pid: 8133, threadinfo=c66d4000 task=d0be7020)
Stack: 00000000 c66d5f48 00000000 00000004 00000000 00000000 00000002 00000004
       00000000 00000000 00000000 000001cf 00000000 00000000 00000001 ffffffff
       ffffffff c013c324 5331a480 00000000 00566223 c131a480 00000000 ffffffff
Call Trace:
 [<c013c324>] buffered_rmqueue+0x134/0x1a0
 [<c0181f31>] proc_tgid_stat+0x11/0x20
 [<c017eaff>] proc_info_read+0x3f/0x70
 [<c0154542>] vfs_read+0x92/0x120
 [<c015483d>] sys_read+0x3d/0x70
 [<c0102c5d>] syscall_call+0x7/0xb
Code: 00 e8 6f d6 1e 00 8b 8e 5c 04 00 00 85 c9 0f 84 c4 00 00 00 8b 99 88 00 0
 00 85 db 74 3d 8b 83 b4 00 00 00 89 44 24 50 8b 43 04 <8b> 50 64 8b 68 68 8b 4
 08 c1 e2 14 09 ea 01 c2 89 d0 c1 e8 14
```

Wyjaśnienie

- Flagi procesora, zawartość rejestrów
- EIP - licznik instrukcji(ew nazwa modułu i nazwa symbolu, który spowodował błąd)
- zawartość wierzchołka stosu
- historia wykonanych instrukcji
- oops może doprowadzić do ’kernel panic’, ale nie zawsze

Jak to rozumieć?

- **Klogd** = Kernel Log Daemon
- **Syslogd**
- /boot/System.map
- /proc/kallsyms

System.map - zdjęcie

```
00000400 A __kernel_vsyscall
00000410 A SYSENTER_RETURN
00000420 A __kernel_sigreturn
00000440 A __kernel_rt_sigreturn
00100000 A phys_startup_32
c0100000 T _text
c0100000 T startup_32
c0101000 T _stext
c0101000 t run_init_process
c0101000 T stext
c0101014 t init_post
c01010ec t try_name
c01012b3 T name_to_dev_t
c01014f0 T calibrate_delay
c0102000 T thread_saved_pc
c010200a T disable_hlt
c0102011 T enable_hlt
c0102018 t poll_idle
c010201b T dump_task_regs
c0102106 T hard_disable_TSC
c0102110 T hard_enable_TSC
c010211a T select_idle_routine
c0102151 T arch_align_stack
c010217e T mwait_idle_with_hints
c01021b8 t mwait_idle
c01021c2 T sys_vfork
c01021f2 T sys_clone
c010222d T sys_fork
c010225d T release_thread
c010226f T kernel_thread
c0102301 T cpu_idle
c0102375 T dump_thread
c0102490 T sys_execve
c0102515 T get_wchan
```

/proc/kallsyms - zdjęcie

```
c0100000 T _text
c0100000 T startup_32
c0101000 T _stext
c0101000 t run_init_process
c0101000 T stext
c0101014 t init_post
c01010ec t try_name
c01012b3 T name_to_dev_t
c01014f0 T calibrate_delay
c0102000 T thread_saved_pc
c010200a T disable_hlt
c0102011 T enable_hlt
c0102018 t poll_idle
c010201b T dump_task_regs
c0102106 T hard_disable_TSC
c0102110 T hard_enable_TSC
c010211a T select_idle_routine
c0102151 T arch_align_stack
c010217e T mwait_idle_with_hints
c01021b8 t mwait_idle
c01021c2 T sys_vfork
c01021f2 T sys_clone
c010222d T sys_fork
c010225d T release_thread
c010226f T kernel_thread
c0102301 T cpu_idle
c0102375 T dump_thread
c0102490 T sys_execve
c0102515 T get_wchan
c010257c T sys_set_thread_area
c0102729 T sys_get_thread_area
c0102854 T show_regs
c0102a19 T default_idle
```

Porównanie plików

- System.map zawiera statyczne adresy symboli w jądrze
- /proc/kallsyms jest dynamicznie zmieniany w trakcie ładowania/usuwania modułów
- zawierają symbole oraz ich adresy
- klogd korzysta z obydwu

Klogd

- przy każdorazowym ładowaniu/usuwaniu modułu trzeba informować o tym **klogd** (opcja -i lub -I)
- funkcje **insmod**, **rmmod**, **modprobe** robią to automatycznie
- w trakcie błędu oops **klogd** szuka wpisu w pliku System.map
- jeśli nie znajdzie, sięga do /proc/kallsyms
- następuje translacja adresów

Trudne!

- wymagana opcja
  [*] Compile the kernel with debug info w jądrze
- Znalezienie pliku (czasami nawet funkcji) na podstawie EIP
- Deasemblacja modułu odpowiedzialnego za błąd
- Użycie programu do deasemblacji, np **objdump**
- Lokalizacja linijki w kodzie źródłowym dla odpowiadającej jej linii z pliku przesuwalnego

- Został wywołany błąd oops. Odczytujemy wartość **EIP**: c2483069
- Przeszukujemy pliki System.map oraz /proc/kallsyms
- Odczytujemy nazwę symbolu:

```
<adres> <symbo> <[nazwa_modułu]>

c2483060 test_read_proc [test]
c2483000 __insmod_test_O/home/ross/prog/test.o_M3 [test]
c2483110 __insmod_test_S.rodata_L68     [test]
c2483060 __insmod_test_S.text_L176      [test]
c2483080 foo    [test]
de79c340 ip6_frag_mem    [ipv6]
```

W naszym przypadku będzie to symbol test_read_proc, z modulu ’test’

- Obliczamy adres:
  (EIP) - (adres bazowy funkcji) = 0xc2483069 - 0xc2483060 = 0x9
- Używając przykładowo programu **objdump**, deasemblujemy plik ’test.o’

```
00000000 <test_read_proc>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
   6:   83 c4 f8                add    $0xfffffff8,%esp
   9:   a1 00 00 00 00          mov    0x0,%eax
   e:   50                      push   %eax
   f:   68 00 00 00 00          push   $0x0
```

Oto fragment pliku źródłowego dla modułu ’test.o’:

```
int test_read_proc(char *buf, char **start, off_t offset, int count, int *eof, void *data)
{
.       int *ptr;
.       ptr=0;
.       printk("%d\n",*ptr);
.       return 0;
}
```

Domyślamy się, że mov 0x0,%eax odpowiada ptr=0

Debugowanie za pomocą **printk**

- Jest to funkcja podobna do **printf**
- Bardzo prosty i szybki sposób na podglądanie zmiennych w trakcie działania jądra
- Na dłuższą metę mało skuteczny sposób
- Składnia: printk(<0-9> ‘tekst’, argumenty...);

Priorytety wiadomości

Oto fragment pliku nagłówkowego:

```
#define    KERN_EMERG      "<0>"      /* system is unusable                 */
#define    KERN_ALERT      "<1>"      /* action must be taken immediately*/
#define    KERN_CRIT       "<2>"      /* critical conditions                */
#define    KERN_ERR        "<3>"      /* error conditions                   */
#define    KERN_WARNING    "<4>"      /* warning conditions                 */
#define    KERN_NOTICE     "<5>"      /* normal but significant condition   */
#define    KERN_INFO       "<6>"      /* informational                      */
#define    KERN_DEBUG      "<7>"      /* debug-level messages               */
```

(filmik)

/proc/sys/kernel/printk

1 4 1 7

- console_loglevel - oznacza, że komunikaty o numerze większym niż ta wartość będą wypisywane na konsoli
- default_message_loglevel - domyślny priorytet dla wiadomości bez ustalonego priorytetu
- minimum_console_level - najmniejsza wartość, jaką można ustawić dla 'console_loglevel'
- default_console_loglevel - domyślna wartość dla console_loglevel

Domyślnie syslogd wypisuje zawartość bufora kernela na tty12, aby mieć wynik na konsoli można dopisać do pliku /etc/syslog.conf linię:
$ xconsole -file /dev/console

[ ] Kernel hacking

```
Linux Kernel v2.6.17.13 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  < > module capable

          [ ] Show timing information on printks
          [*] Magic SysRq key
          [*] Kernel debugging
          (14)  Kernel log buffer size (16 => 64KB, 17 => 128KB) (NEW)
          [*]   Detect Soft Lockups (NEW)
          [ ]   Collect scheduler statistics (NEW)
          [ ]   Debug slab memory allocations (NEW)
          [*]   Debug preemptible kernel (NEW)
          [ ]   Mutex debugging, deadlock detection (NEW)
          [ ]   Spinlock debugging (NEW)
          [ ]   Sleep-inside-spinlock checking (NEW)
          [ ]   kobject debugging (NEW)
          [ ]   Highmem debugging (NEW)
          [ ]   Verbose BUG() reporting (adds 70K) (NEW)
          [ ]   Compile the kernel with debug info (NEW)
          [ ] Debug Filesystem
          [ ] Debug VM (NEW)
          [ ] Compile the kernel with frame pointers (NEW)
          [ ] Compile the kernel with frame unwind information
          [*] Force gcc to inline functions marked 'inline' (NEW)
          < > torture tests for RCU (NEW)
          [*] Early printk (NEW)
          [ ] Check for stack overflows (NEW)
          [ ] Stack utilization instrumentation (NEW)
          (2) Stack backtraces per line (NEW)
          [ ] Write protect kernel read-only data structures (NEW)
          [ ] Use 4Kb for kernel stacks instead of 8Kb (NEW)
          [*] Enable doublefault exception handler

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

Magiczny SysRq

[ ] Magic SysRq key

Alt + SysRq + ...

- H - wypisuje pomoc
- B - natychmiastowy restart systemu, bez odmontowania dysków i zapisania ich buforów. Zazwyczaj uszkadza system plików
- E - wysyla sygnal SIGTERM do wszystkich procesow z wyjatkiem INIT
- I - wysyla sygnal SIGKILL do wszystkich procesow z wyjatkiem INIT
- K - zabija wszystkie procesy na biezacym terminalu (przydatne np gdy padnie serwer X)

Magiczny SysRq - c.d.

- L - wysyła sygnał SIGKILL do wszystkich procesow (INIT też)
- M - wypisuje informacje o pamięci. Przydatne tylko do debugowania.
- O - wyłącza system operacyjny
- P - wypisuje zawartość rejestrów i flag procesora
- R - przełącza tryb klawiatury (np jeśli przestanie działać w X serwerze)
- S - pozwala zapisać zawartość buforów na dysk (np gdy chcemy zrobić restart)
- (0 - 9) zmienia console_loglevel na [0-9]

[ ] Show timing information of printks - opcja ta umożliwia wyświetlanie czasu razem z tekstem w funkcji printk(). Jest to pomocne w celu sprawdzenia długich opóźnień w działaniu jądra (np podczas startu).

[*] Kernel debugging

- [ ] Kernel log buffer size (16 => 64KB, 17 => 128KB) - pozwala na zmianę bufora cyklicznego jądra
- [ ] Detect Soft Lockups - pozwala na wykrycie śoft lockup", czyli stanu w którym kernel ’zapętla się’ na minimum 10s w trybie jądra nie oddając procesora innym zadaniom
- [ ] Collect scheduler statistics - przydatne w tuningowaniu schedulera, pozwala na zbieranie statystyk do /proc/schedstat

[*] Kernel debugging - c.d.

- [ ] Debug slab memory allocations - kontroluje przydzielanie i zwalnianie pamieci, jednak moze znacznie spowolnić działanie systemu
- [ ] Mutex debugging, deadlock detection - pozwala na automatyczne wykrywanie i raportowanie mutexów oraz blokad semaforowych
- [ ] Spinlock debugging - umożliwia wykrywanie niezainicjalizowanych spinlockow (aktywne oczekiwanie) i innych, często wykonywanych na nich błędów
- [ ] Verbose BUG() reporting (adds 70K) - pozwala na wykrycie nazwy pliku oraz linii wywołania, które wygenerowało błąd na podstawie EIP z oops (kosztem dodatkowych 70-100KB pamięci)

[*] Kernel debugging - c.d.

- [ ] Compile the kernel with debug info - zwiększa rozmiar kernela, ale musowo to właczyc przy uruchamianiu jądra w debugerze (zwieksza rozmiar jądra)

Pozostałe opcje:

- **[ ] Compile the kernel with frame pointers** - należy włączyć jeśli chcemy używać zewnętrznych debugerów jądra (zwiększa rozmiar jądra i spowalnia działanie)
- **[ ] Compile the kernel with frame unwind information** - podobnie jak wyżej (zwiększa rozmiar, ale nie spowalnia jądra)

- **[ ] Early printk** - jest to opcja przydatna, gdy jądro dosyć wcześnie napotyka na błędy. Pozwala na wypisywanie informacji bezpośrednio na konsole, jednak konfliktuje z wszelkimi daemonami typu klogd, syslogd
- **[ ] Check for stack overflows** - sprawdza czy nie nastąpiło przepełnienie stosu
- **[ ] Use 4Kb for kernel stacks instead of 8Kb** - pozwala na zmianę domyślnej wielkości stosu z 8Kb na 4Kb

- GNU Debugger
- standardowy debbuger dla systemów GNU
- napisany w 1986 roku przez Richarda Stallmana
- dostępny dla wielu architektur, w tym dla x86, x86-64, IA-64, Motoroli 68000, PowerPC i SPARCa
- umożliwia debuggowanie programów napisanych w C, C++, Object-C, Fortranie, Pascalu, Moduli-2 i Adzie

GDB umożliwia debuggowanie za pomocą:

- ustalenia zbioru zdarzeń (breakpointów, watchpointów), których zajście ma wstrzymać działanie programu
- możliwości odczytu stanu programu w momencie przerwania działania
- stałej modyfikacji kodu i analizy zmian zachodzących w programie
- istnieje możliwość zdalnego debuggowania, przydatna przy debuggowaniu systemów wbudowanych lub jądra linuksa(z pomocą KDGB, ale o tym później)

Najczęściej używane komendy w GDB:

- gdb <nazwa pliku wykonywalnego> - uruchomienie gdb
- r (run) <lista argumentów> - uruchomienie programu z opcjonalną listą argumentów
- b (break) <nazwa funkcji lub numer linii> - ustawienie breakpointa na podanej funkcji lub w podanej linii debuggowanego kodu
- n (next) - jeden krok programu (w kodzie źródłowym)
- s (step) - jeden krok programu
- signal <sygnał> wysłanie sygnału do debuggowanego procesu
- bt (backtrace) - wyświetlenie aktualnego stosu wywołań

GDB - Podstawowe komendy

- p (print) <nazwa zmiennej> - wypisuje wartość podanej zmiennej
- d <numer breakpointa> - usunięcie breakpointa, bez argumentu usuwa wszystkie breakpointy
- c (continue) - wznowienie działania programu
- return <wyrażenie> - powrót z funkcji z wartością wyrażenia
- help <komenda> - najważniejsza komenda, bez podania komendy wyświetla spis treści pomocy

KDB (Built-in Kernel Debugger) - wbudowany debugger jądra. Dzięki niemu można:

- wypisywać zawartości struktur danych systemu według podanego adresu
- wykonywać jeden krok polecenia procesora
- zatrzymywać się z powodu wykonania konkretnej instrukcji
- odczytywać stos wywołań danego procesu

KDB - Możliwości

KDB zostaje przywołane w następujących przypadkach:

- jeśli wystąpi kernel panic
- jeśli KDB napotka na wcześniej zdefiniowany breakpoint
- przez naciśnięcie przycisku PAUSE/BREAK przez użytkownika

1. KDB instaluje się poprzez spatchowanie i skompilowanie jądra z wybranymi opcjami
2. ściągamy dwa patche dla naszej wersji jądra ze strony http://oss.sgi.com/projects/kdb/
    - kdb-v4.4-wersja-jądra-common-X.bz2
    - kdb-v4.4-wersja-jądra-arch-X.bz2

   na przykład pliki
    - kdb-v4.4-2.6.22-common-4.bz2
    - kdb-v4.4-2.6.22-i386-2.bz2

   są przeznaczone dla KDB w wersji 4.4 dla jądra 2.6.22 na architektury i386

KDB - Instalacja

3. kopiujemy ściągnięte patche do rozpakowanego źródła i rozpakowujemy
   - bzip2 -d kdb-v4.4-2.6.22-common-4.bz2
   - bzip2 -d kdb-v4.4-2.6.22-i386-2.bz2

   - patch -p1 < kdb-4.4-2.6.22-common-4
   - patch -p1 < kdb-4.4-2.6.22-i386-2

   Nie powinien pojawić się żaden błąd.
4. w konfiguracji jądra, w zakładce “Kernel Hacking”, zaznaczamy opcję “Built-in Kernel Debugger” support. Aby KDB poprawnie wyświetlał stos wywołań procesu, należy zaznaczyć opcję “Compile the kernel with frame pointers”.

Jeśli CONFIG_KDB_OFF nie było zaznaczone podczas kompilacji, KDB będzie domyślnie włączony. W przeciwnym przypadku trzeba go uruchomić, ustawiając flagę kdb=on (w lilo/grubie) lub przez polecenie

- echo “1” > /proc/sys/kernel/kdb

Aby wyłączyć KDB, wystarczt ustawić flagę kdb=off lub wykonać następujące polecenie

- echo “0” > /proc/sys/kernel/kdb

- md <adres/symbol> <k> - wyświetla wartość pamięci zaczynającej się w podanym adresie dla k linii, jeśli k nie jest podana, wykorzystywane są zmienne środowiskowe, jeśli adres nie jest podany, md kontynuuje wyświetlanie od ostatniego wyświetlonego miejsca.
- mm <adres/symbol> <nowa wartość> - zmienia wartość zapisaną pod danym adresem/symbolem na nową wartość.

Przykłady:

- aby wyświetlić 15 linii pamięci zaczynającej się od 0xc000000 wpisać md 0xc000000 15
- aby zmienić wartość pamięci zapisanej pod adresem 0xc000000 na 0x10 wpisać mm 0xc000000 0x10

- bp <adres/symbol> - ustawia breakpoint na podanym adresie
- bd <numer> - wyłącza breakpoint o podanym numerze, breakpointy są numerowane począwszy od 0
- be <numer> - włącza breakpoint o podanym numerze
- bl - wypisuje listę wszystkich (włączonych i wyłączonych) breakpointów
- bc <numer> - usuwa breakpoint o podanym numerze z listy breakpointów, jeśli jako parametr podano *, usuwa wszystkie breakpointy

Przykłady

- aby ustawić breakpoint na funkcji sys_write(), wpisać bp sys_write
- aby usunąć breakpoint o numerze 1, wpisać bc 1

KDB - Podstawowe komendy

- rd - wyświetla wartości rejestrów procesora
- rm <nazwa> <nowa wartość> - zmienia wartość rejestru o podanej nazwie na nową wartość

Przykłady:

- aby zmienić wartość rejestru ebx na 0x25 wpisać rm %ebx 0x25

KDB - Podstawowe komendy

- bt - wypisuje zawartość stosu wywołań bieżącego wątku
- btp <numer procesu> - wypisuje wartość stosu wywołań dla procesu o podanym pidzie

- KGDB - kolejny debugger
- instaluje się przez spatchowanie jądra i wybranie odpowiednich opcji
- do debuggowania potrzebne dwa komputery połączone kablem szeregowym
- na jednym działa zwykły system, na drugim jest uruchomiony jest gdb
- gdb może debuggować jądro zdalnego systemu jak zwykły program
- można próbować uruchomić na wirtualnych systemach

- wszystko, co oferuje gdb, czyli
  - breakpointy
  - wyświetlanie wartości zmiennych, rejestrów
  - modyfikacja pamięci
  - oglądanie stosu wywołań funkcji systemowych
- obsługa systemów wieloprocesorowych

- do poprawnego uruchomienia - dwa komputery połączone kablem szeregowym
- możemy też użyć dwóch wirtualnych maszyn

1. Uruchamiamy VMWare i instalujemy dwa wirtualne systemy. Pozostawiamy wyłączony ten, który będzie debuggowany
2. Ściągamy źródła linuksa i rozpakowujemy je:
    - cd /usr/src
    - wget “http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.23.tar.bz2”
    - tar xjvf linux-2.6.23.tar.bz2
3. Ściągamy patche kgdb dla naszej wersji jądra i aplikujemy je. Jeśli patche aplikujemy za pomocą polecenia patch, to należy pamiętać o poprawnej kolejności patchowania, podanej w pliku series Możemy też zainstalować patche za pomocą polecenia quilt, więcej w pliku README

KGDB - Instalacja

4. W konfiguracji jądra, w zakładce “Kernel hacking” zaznaczamy:
   - “Compile the kernel with frame pointers”
   - “Compile the kernel debug info”
   - “KGDB: kernel debugging with remote gdb”
   - “KGDB: Console messages through gdb”
   - “Simple selection of KGDB serial port”
   - w “Serial port number for KGDB” wpisujemy 0
5. Kompilujemy jądro
6. Po kompilacji powstaną dwa pliki: vmlinux i arch/i386/boot/bzImage. bzImage należy przenieść na drugą maszynę wirtualną

KGDB - Instalacja

7. Na drugiej maszynie, po przeniesieniu bzImage, instalujemy jądro, dodając odpowiedni wpis do menu.lst
8. Należy pamiętać, aby do opcji tego jądra dodać parametr kgdbwait
9. Do systemu z zainstalownym nowym jądrem dodajemy port szeregowy. Ma on łączyć się z named pipe o nazwie x. należy w zaawansowanych opcjach wybrać "yield CPU on poll"
10. Z http://lass.cs.umass.edu/ pjd/nptp_setup.zip ściągamy i uruchamiamy program "Named Pipe TCP Proxy"
11. W programie tym dodajemy nowe połączenie między portem lokalnym na naszym komputerze oraz x. Ważne jest, aby zaznaczyć opcję "enable non-local systems access"

12. Uruchamiamy drugą maszynę, oczywiście na nowym jądrze
13. Na pierwszej maszynie przechodzimy do katalogu z plikiem vmlinux
    - cd /usr/src/linux-source-2.6.23/
14. I uruchamiamy gdb
    - gdb vmlinux
15. nawiązujemy połączenie z drugim systemem
    - target remote ip_adrr:port

    gdzie ip_adrr to ip hosta a port to numer portu, jaki wybraliśmy podczas tworzenia łącza nazwanego w programie Named Pipe TCP Proxy
16. możemy teraz możemy debuggować jądro przy pomocy gdb tak, jakby to był zwykły program

Jądro Linux GUEST jest uruchamiane w trybie użytkownika jako zwykły proces goszczący na jądrze HOST-a. Zalety:

- możemy bezpiecznie uruchamiać niestabilny system używając tylko jednego komputera
- daje nam to możliwość przyjrzenia się działającemu procesowi Linux w szczególności debugowania go.
- nie tracimy zbyt wiele zasobów na HOST
- działający system GUEST wyróżnia się dużo szybkościa w porównianiu do innych technik wirtualizacji

UML może być uruchamiany w dwóch różnych trybach które znacząco się różnią.

Tryb TT (Tracing Thread)- przeszłość

- każdy proces działający jako na systemie gościa jest widziany z systemu host
- jądro gościa ma możliwość nadpisania czegość w pamięci systemu host, a przecież bywa niestabilne
- uruchomiony wątek przechwytuje wywołania niskopoziomowe z procesów, także z jądra systemu gościa - nieoptymalne

Tryb ten nie będzie omawiany na tej prezentacji z racji tego ,że nie działa na obecnie używanych jądrach , wyszedł z użycia.

SKAS (Single Kernel Address Space)

- systemy (host , guest) mają rozdzieloną przestrzeń adresową
- zasadą SKAS jest mała liczba procesów widocznych z poziomu host-a (w SKAS3 zawsze 4) które przełączają się obsługując zadania gościa - nie zajmują kolejek na host

Tryb SKAS3 jest możliwy do uruchomienia po zainstalowaniu specjalnej łatki na jądro gospodarza, tryb SKAS0 jest dostępny od razu na współczesnych jądrach, przykłady na tej prezentacji będą robione w trybie SKAS0.

Instalacja UML dla trybu SKAS0 sprowadza się do skompilowania jądra gościa z odpowiednimi flagami i podpięcie mu systemu plików, teoretycznie. Do działania będzie więc potrzebne:

- skompilowane jądro z flagami ARCH=um
- system plików - wirtualna partycja na której będzie działał system gościa

1. Na tej prezentacji zajmiemy się jądrem 2.6.17.13 , jądro musi być bez żadnych łatek, tak więc ściągamy jądro z kernel.org

```
mkdir ~/kernel
cd ~/kernel
wget http://www.eu.kernel.org/pub/linux/kernel/
v2.6/linux-2.6.17.13.tar.bz2
```

2. Rozpakowujemy archiwum

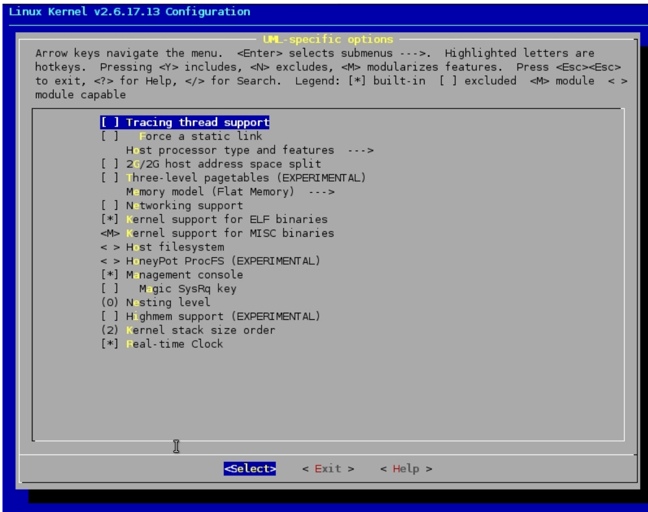
```
tar -jxvvf linux-2.6.17.13.tar.bz2
cd linux-2.6.17.13
```

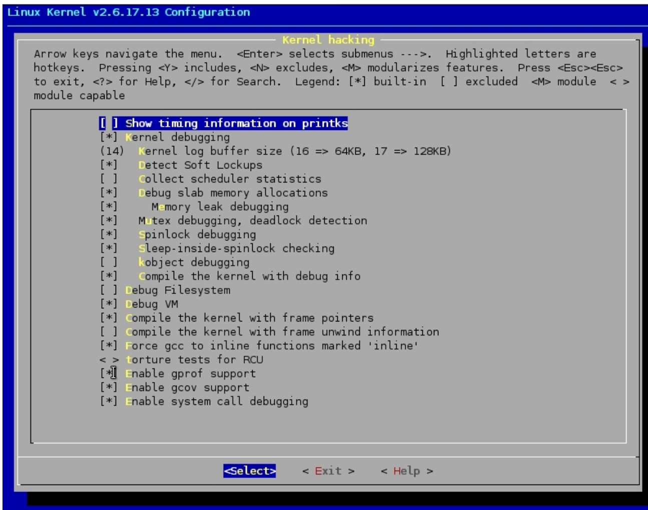
3. Tworzymy domyślną konfigurację, należy pamiętać o opcji ARCH=um (kompilujemy jądro na architekturę um) a następnie zmieniamy kilka ustawień.

```
make defconfig ARCH=um
make menuconfig ARCH=um
```

4. W sekcji ustawień specyficznych dla UML oraz w Kernel hacking zaznaczamy opcje które mogą się nam przydać w trakcie debugowania i odznaczamy te które nie będą na pewno potrzebne:

Instalacja




Linux Kernel v2.6.17.13 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 < > module capable
[ ] Show timing information on printks
[*] Kernel debugging
(14) Kernel log buffer size (16 => 64KB, 17 => 128KB)
[*] Detect Soft Lockups
[ ] Collect scheduler statistics
[*] Debug slab memory allocations
[*] Memory leak debugging
[*] Mutex debugging, deadlock detection
[*] Spinlock debugging
[*] Sleep-inside-spinlock checking
[ ] kobject debugging
[*] Compile the kernel with debug info
[ ] Debug Filesystem
[*] Debug VM
[*] Compile the kernel with frame pointers
[ ] Compile the kernel with frame unwind information
[*] Force gcc to inline functions marked 'inline'
< > torture tests for RCU
[*] Enable gprof support
[*] Enable gcov support
[*] Enable system call debugging
<Select>
< Exit >
< Help >

5. Kompilujemy jądro

```
make linux ARCH=um
```

6. Mamy już gotowe jądro , teraz zajmiemy się systemem plików, w tym celu pobieramy gotowy dla UML system plików Linux Slackware (jest mały) , rozpakowujemy i układamy jądro, pliki w jednym katalogu

```
mkdir ~/uml
cd ~/uml
wget http://heanet.dl.sourceforge.net/sourceforge/
user-mode-linux/root_fs_slack8.1.bz2
bzip2 -fd root_fs_slack8.1.bz2
mv root_fs_slack8.1 fs
rm root_fs_slack8.1.bz2
cp ~/kernel/linux linux
```

7. Musimy nieco skonfigurować system plików a dokładnie plik /etc/fstab, uruchamiając jądro zdefiniujemy ze urządzeniem ubd0 jest właśnie ten system plików dla tego w nim musimy skonfigurować ze ubd0 ma być zamontowany jako główny katalog, w tym celu montujemy system plików i edytujemy /etc/fstab

```
mkdir fs_dir
mount fs fs_dir -o loop
vi fs_dir/etc/fstab
```

plik powinniśmy doprowadzić do takiej postaci:

```
/dev/ubd0 / ext2 defaults 1 1
none /dev/pts devpts gid=5,mode=620 0 0
none /proc proc defaults 0 0
```

jak widać nie zajmujemy się tu partycją SWAP, po prostu nie będziemy testować działania jądra związanego z obsługą SWAP, po zakończeniu odmontowywujemy system plików

```
cd ../
umount fs_dir
```

8. Możemy już przejść do uruchamiania Linuksa , jeśli chcemy uruchomić zwyczajnie system to uruchamiamy kernel jak każdy inny program

```
./linux OPCJE
```

w przypadku uruchamiania debuggerem ładujemy plik bez żadnych opcji a następnie wewnątrz debuggera uruchomimy go z odpowiednimi opcjami

```
gdb linux
<...w dbg...>
gdb> r OPCJE
```

OPCJE dają nam możliwość konfiguracji uruchamianego systemu, należy znać takie jak:

- ubd<n><flagi>=<system plików> jak się później okaże opcja konieczna, jądro musi mieć odpowiednie programy do uruchomienia
  <flagi> = (r = readonly | s = O_SYNC | c = shared) - raczej mało dla nas istotne , może być potrzebne przy korzystaniu przez kilka systemów z jednej partycji
- mem=<rozmiar przydzielonej pamięci RAM>(K | M | G)
- mode=(tt | skas0 | skas3)

możemy także konfigurować sieć w kilku trybach , inne urządzenia lecz nie będzie nam to potrzebne.

Uwagi do uruchamiania poprzez gdb:

- GDB wysyła w trakcie działania różne sygnały do procesów, w efekcie jak się później okaże jądro wyłącza się zaraz po starcie w wyniku sygnałów SIGSEGV i SIGUSR1, aby temu zapobiec należy zapobiec docierania sygnałów do procesów jądra, w trakcie sesji gdb będziemy musieli najpierw przekonfigurować te sygnały:

```
gdb> handle SIGSEGV pass nostop noprint
gdb> handle SIGUSR1 pass nostop noprint
```

- Jeśli chcemy po uruchomieniu jądra pod gdb powrócić do gdb musimy wyslać sygnał SIGINT do procesu o najmniejszym pid spośród procesów uruchomionego przez gdb systemu. Będzie to oczywiście proces którego ojcem jest gdb więc przerwanie go zwolni konsole gdb.

W praktyce: sprawdzę jak można uruchamiać jądro, które z opcji są konieczne i co się dzieje kiedy nie zdefiniujemy koniecznych opcji

Teraz coś rzeczywiście praktycznego: chcę się dowiedzieć kilku rzeczy o startowaniu jądra

Użyta technika: założę breakpoint na funkcje jądra do_fork, użyje nakładki graficznej na GDB, mianowicie DDD, w DDD mamy standardowo:

- ramka z obserwowanymi zmiennymi
- ramka z kodem
- konsola GDB

Debugować można w trybach:

- TT - używa się do tego specjalnego skryptu umlgdb, po skonfigurowaniu wszystkiego i uruchomieniu mamy SEGFAULT
- SKAS - ta opcja nawet czasem działa, wybieram tą

1. Musimy zainstalować moduły (a potem najlepiej właściwy dla nas moduł) na naszym systemie plików dla jądra gościa

```
cd ~uml
mount fs fs_dir -o loop
cd ~kernel/linux-2.6.17.13
make modules ARCH=um
make modules_install INSTALL_MOD_PATH=~uml/fs_dir ARCH=um
```

2. Zakładam że mamy odpowiednio skompilowany nasz moduł "mod.ko". Odpowiednio:

   - W Makefile ustawiliśmy pobranie nagłówków z ∼kernel/linux-2.6.17.13 - ze źródeł skompilowanego jądra na architekture um identycznego do tego na którym będziemy moduł debugować.
   - W Makefile ustawiliśmy opcje ARCH=um
   - Kompilowaliśmy go tym samym kompilatorem co do wersji itd. co jądro, w tym samym środowisku, najlepiej zaraz po sobie

Uwaga: Każda niezgodność prędzej czy później da o sobie znać, w ostateczności "można" też edytować plik mod.ko (pamiętając o strukturze ELF-a!). Jeśli ktoś z jakiś powodów nigdy nie miał okazji pisać kompilatora asemblera może mieć trudniej.

3. Kopiujemy nasz moduł

```
cp ~module/mod.ko
~uml/fs_dir/lib/modules/2.6.17.13/kernel/drivers/misc/mod.ko
```

4. Musimy zainstalować na naszym systemie plików programy do obsługi modułów. Ściągamy paczki:

```
cd ~uml/fs_dir
wget http://www10.frugalware.org/pub/linux/zenwalk/i486/
zenwalk-2.8/packages/zenwalk/a/module-init-tools-3.3pre1-i486-1z28.tgz
wget http://riksun.riken.go.jp/pub/pub/Linux/slackware
/slackware-11.0/slackware/l/glibc-2.3.6-i486-6.tgz
cd ..
umount fs_dir
./linux ubd0=fs
GUEST> cd /
GUEST> installpkg module-init-tools-3.3pre1-i486-1z28.tgz
GUEST> installpkg glibc-2.3.6-i486-6.tgz
GUEST> halt
```

Koniecznie w tych wersjach!

1. Uruchamiamy debugger z Linuksem i wstawiamy breakpoint-a na moment ładowania modułu - żeby wydobyć adres modułu i przypisać tam symbole z modułu.

```
gdb linux
GDB> handle SIGSEGV pass nostop noprint
GDB> handle SIGUSR1 pass nostop noprint
GDB> break module.c: 1775
GDB> r ubd0=fs
GUEST> depmod
GUEST> modprobe mod
GDB> print ((struct module *) _mod)->module_core
GDB> add-symbol-file ~module/mod.ko <TU ADRES Z PRINT-a>
GDB> break mod.c: <linia z pliku lub funkcja ogolnie>
GDB> c
GUEST>
GDB> <debugujemy moduł>
```

W praktyce: próba debugowania modułu