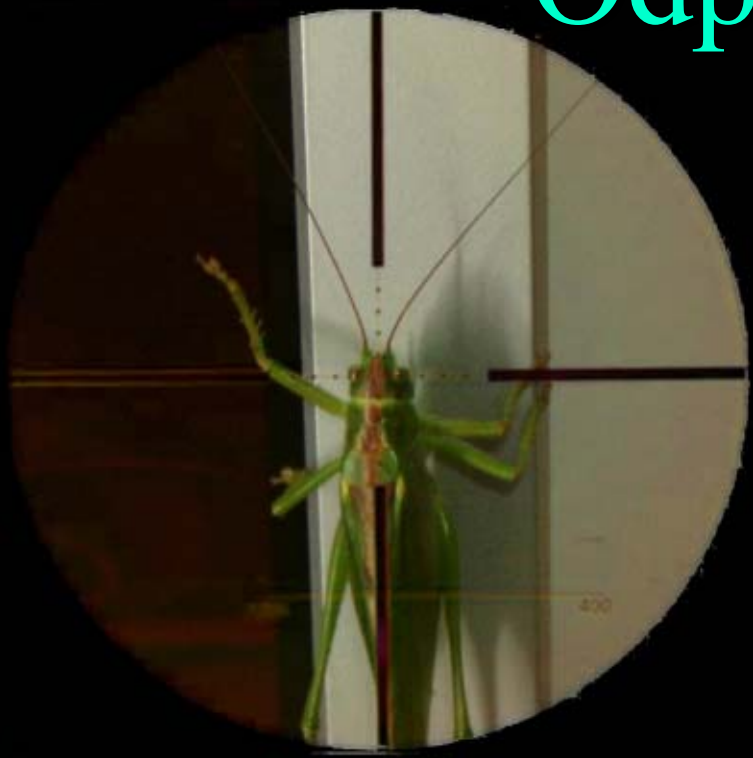
Odpluskwianie

Odpluskwianie



Odpluskwianie

- **Proces systematycznego znajdowania i usuwania błędów w programie.**
- **Przeciwieństwo programowania (procesu systematycznego wprowadzania błędów).**

Metody odpluskwiania

- Czytanie kodu
- Komunikaty kontrolne
- Debuger'y
- (Inne) rytuały magiczne

Czytanie kodu

- Można po prostu przeczytać kod i zobaczyć, gdzie jest błąd.
- Problem w tym, że nie zawsze można.
- Należy więc użyć pozostałych technik do przybliżonego zlokalizowania błędu.

Komunikaty kontrolne

- **Modyfikacja kodu programu tak, by wypisywał pewne informacje o stanie w trakcie działania.**
- **Miażdżąca krytyka:**
  - **Konieczność częstych rekompilacji**
  - **„Tymczasowe” instrukcje wpisane w kod mają tendencję do pozostawania w nim.**
  - **…**

Debuger'y

- **Debuger, jaki jest, każdy widzi. Powinien w szczególności umożliwiać**
  - **Uruchamianie, zatrzymywanie i restartowanie programu.**
  - **Wykonanie małymi krokami**
  - **Zatrzymywanie programu w ustalonych punktach**

Debuger'y cd.

- Podglądanie wartości rejestrów i pamięci odpluskwianego procesu
- Modyfikację powyższych
- W miarę możliwości korzystanie w ww. przypadkach z nazw zmiennych, funkcji etc. oraz numerów linii z kodu źródłowego.
- Analizę obrazu pamięci pozostawionego przez poległy w boju program.

Popularne* debuger'y

- dbx
- (k)gdb (+ ddd)
- (k)adb
- SoftICE
- Windbg
- CodeView (czyt. Visual Studio)
- Inne zintegrowane z IDE (Delphi, …)

* Mniej lub bardziej

Rodzaje debuger'ów

- **Maszynowe**
  - **„Rozumieją" program na poziomie kodu maszynowego, nie korzystają z kodu źródłowego (Przykład: kdb).**
- **Symboliczne**
  - **Potrafią korzystać z symboli pochodzących z kodu źródłowego (zmiennych, funkcji, …) oraz powiązać skompilowany kod z odpowiednimi liniami źródeł (Przykład: gdb).**

Rodzaje debuger'ów cd.

- Systemowe
  - Służą do odpluskwiania kodu działającego w trybie jądra.
  - Przykłady
    - SoftICE
    - kgdb
- „Zwykłe"
  - Do odpluskwiania normalnych aplikacji

Narzędzia debugeropodobne

- **Profiler'y**
  - **Służą do analizy czasu działania programu**
- **Wykrywacze wycieków**
  - **Pomagają wykrywać błędy przy zarządzaniu pamięcią**
- **I inne (np. gcov)**

"Zwykłe" debugowanie

Plan prezentacji

- Debugowanie w Linuksie
- gdb
- Debugowanie w Windows
- WinDbg

Debugery pod Linuksa

- **gdb**
  - **xxgdb**
  - **UPS**
  - **DDD**
  - **kdbg**

Gdb pozwala na…

- uruchomienie programu
- zatrzymanie działającego programu
- analizę tego, co się stało w momencie zatrzymania
- modyfikację elementów programu

Uruchamianie gdb

1. gdb
2. gdb program

---

3. gdb program core
4. gdb program <pid>

Koniec pracy...

5. quit lub C - d

Kompilacja i uruchamianie pod gdb

- **Kompilacja**

  [pk209469@Ost83m1509C gdb]$ gcc -g -o przyklad1 przyklad1.c

  – **opcja ‘-g’**

  – **opcja ‘-O’**

- **Uruchamianie**

  [pk209469@Ost83m1509C gdb]$ gdb przyklad1

Informacje dla programu

- **Wywołanie programu uzależnione jest od informacji:**
  - **argumenty programu**
    - *set args*
  - **środowisko**
    - *path <directory>*
    - *set environment varname[=value]*
  - **bieżący katalog roboczy**
    - *cd <directory>*
  - **standardowe wejście i wyjście**
    - *run > outfile*

Debugowanie działającego procesu

- *attach* – podłączenie działającego procesu
- *detach* – odłączenie procesu
- *kill* – zabicie procesu

Wiele procesów w programie

- gdb kontroluje proces macierzysty

Jak odpluskwiać potomka?

1. breakpoint na sleep
2. podpięcie potomka pod gdb
3. odpluskwianie potomka

Punkty kontrolne

- **breakpoints**
- **watchpoints**
- **catchpoints**

Breakpoints – ustawianie i usuwanie

- **break**
- **break function; break linenum; break +offset; break -offset**
- **break ... if cond**
  Np. '$foo-- <= 0'
- **watch expr**
- **catch event**
- **info break[n]**
- **clear**

(gdb) break density
Breakpoint 1 at 0x80484ad: file przyklad1.c, line 39.

Breakpoints – aktywowanie i deaktywowanie

- **Każdy breakpoint może być w jednym ze stanów:**
  - **aktywowany**
  - **nieaktywny**
  - **aktywowany jeden raz**
  - **aktywowany do usunięcia**

enable [breakpoints] [param] [range...]

disable [breakpoints] [range...].

Kontynuowanie wykonywania programu

- *continue [ignore-count]*
- *step [count]*
- *next [count]*
- *finish*

Breakpoints - przykład

(gdb) info break

Num Type Disp Enb Address What

1 breakpoint keep n 0x080484ad in density at przyklad1.c:39
breakpoint already hit 1 time

2 breakpoint keep y 0x080484be in density at przyklad1.c:40

--- dodaj warunek dla drugiego breakpoint-a ---

(gdb) condition 2 (x==1 && y ==1)

Analiza stosu wywołań

- **Stos i ramka**
- *frame args*
- *select-frame*
- *info f*
  - **adres**
  - **adres poprzedniej ramki oraz następnej**
  - **język, w którym został napisany kod**
  - **adres argumentów**
  - **adres zmiennych**
  - **licznik programu oraz które rejestry zostały zapisane**
- *info args, info locals, info catch*
- *backtrace*

Backtrace - przykład

(gdb) backtrace

#0 0x080484a5 in eval (x=0, y=5) at przyklad.c:31

#1 0x0804841c in main () at przyklad.c:21

Zmiana i analizowanie danych

- **Wyświetlanie**
  - *print*

    (gdb) print result

    $5 = 576
  - *x*
  - *info registers*
  - *info variables*
- **Zmiana wartości**
  - *print varname=newval*
  - *set var*
- **wsparcie dla lokalnych zmiennych**
  - **nazwy tych zmiennych rozpoczynają się od '$'**

Analiza zrzutu pamięci

- **Do najczęstszych powodów awarii należą:**
  - **próba odczytu lub zapisu pamięci, do której proces nie ma dostępu (ang. *segmentation fault*);**
  - **"błędy magistrali";**
  - **"wyjątki zmiennoprzecinkowe„**
- *ulimit –c unlimited*

[pk209469@Ost83m1509C gdb]$ gdb przyklad core.3073

Emacs i gdb

- aby uruchomić gdb w Emacs, należy użyć polecenia Emacs M-x gdb
- wiele okien
- dostosowanie do własnych potrzeb

Debugowanie w Windows

Debugery pod Windowsa

- **KD** - Kernel debugger. Użyteczny w zdalnym debugowaniu systemu operacyjnego, dotyczącym na przykład problemu "niebieskiego ekranu" Windows.
- **CDB** - Command-line debugger. Aplikacja na konsolę.
- **NTSD** - NT debugger. Debuger wykorzystywany w trybie użytkownika służący do debugowania aplikacji użytkownika. CDB wykorzystujący graficzny UI.
- **Windbg** - połączenie KD oraz NTSD wykorzystując graficzny UI.
- **Visual Studio, Visual Studio .NET** - używa tych samych "silników" co KD i NTSD, oferując przy tym dużo bardziej bogate UI niż Windbg.

Porównanie debugerów

Cecha	KD	NTSD	WinDbg	VS
K-mode deb	Y	N	Y	N
U-mode deb		Y	Y	Y
Unmanaged deb	Y	Y	Y	Y
Managed deb		Y	Y	Y
Remote deb	Y	Y	Y	Y
Attach to process	Y	Y	Y	Y
Detach from process	Y	Y	Y	Y
SQL deb	N	N	N	Y

WinDbg

- wykonywanie programu krok po kroku, przeglądając jego kod źródłowy
- ustawianie punktów kontrolnych
- podgląd rejestrów i flag CPU
- podgląd kodu w assemblerze
- podgląd wartości zmiennych
- podgląd stosu wywołań
- analizę bloków pamięci
- debugowanie za pośrednictwem sieci
- podłączenie do działającej aplikacji
- analizę zrzutu pamięci
- załadowanie pliku wykonywalnego
- załadowanie kodu źródłowego

Uruchamianie WinDbg

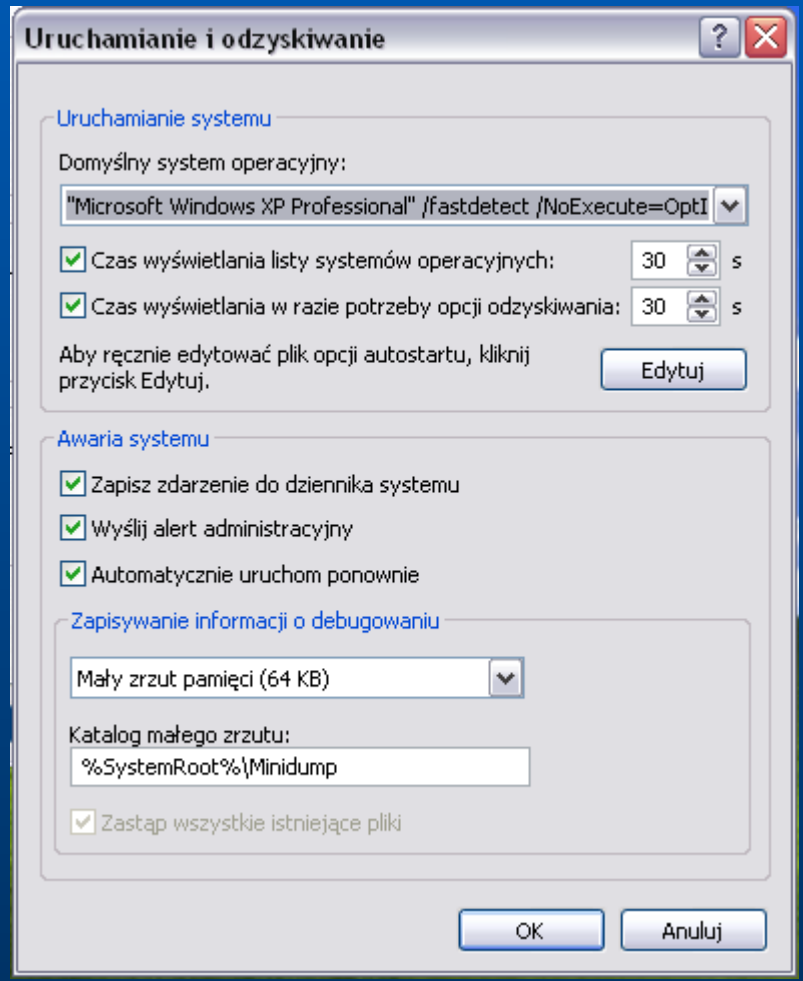
- **Ścieżka do symboli**

  File | Symbol File Path dodajemy:

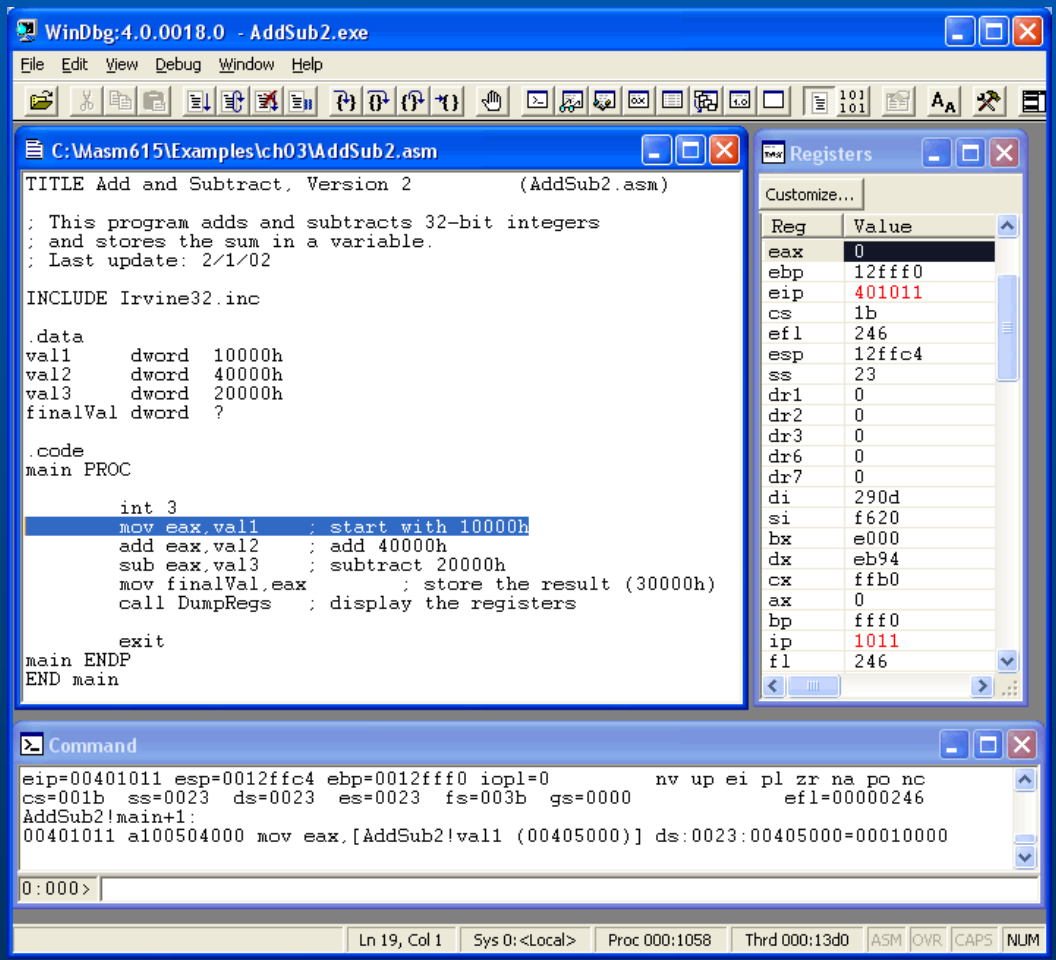
SRV*<c:\katalog_symboli>*http://msdl.microsoft.com/download/symbols

Plik zrzutu pamięci w Windows

- **Windows XP oferuje przygotowywanie trzech rodzajów zrzutów pamięci:**
  - **Zrzut pamięci jądra**
  - **Mały zrzut – 64 KB**
  - **Pełny zrzut pamięci**
- **Opcje te można znaleźć we *Właściwości* systemu w zakładce *Zaawansowane* pod pozycją *Uruchamianie i Odzyskiwanie*.**



Wygląd WinDbg



Debugowanie zdalne

- **Uruchamianie serwera:**

WinDbg –server npipe:pipe=*pipename* (note: multiple clients can connect)

lub z WinDbg:

.server npipe:pipe=*pipename* (note: single client can connect)

- **Uruchamianie klienta:**

  WinDbg -remote npipe:server=*Server*, pipe=*PipeName*[,password=*Password*]

lub z WinDbg:

File->Connect to Remote Session

dla połączenia, wpisz:

npipe:server=*Server*, pipe=*PipeName* [,password=*Password*]

Debugowanie „just-in-time”

- **Ustalanie WinDbg jako domyślnego debugera JIT:**

  Windbg –I

- **Domyślny „managed debugger”:**
  - HKLM\Software\Microsoft\.NETFramework\DbgJIT DebugLaunchSetting na 2
  - HKLM\Software\Microsoft\.NETFramework\DbgManagedDebugger na Windbg.
- **Uruchamianie WinDbg w przypadku, gdy aplikacja nie radzi sobie z obsłużeniem**

Debugowanie wyjątków

- '*first chance exception*' **- pierwsza szansa dla debugera (w momencie „włamania" aplikacji do debugera)**
- '*second-chance exception*' **– kolejna szansa dla debugera (w przypadku zignorowania pierwszej i nie znalezienia obsługi wyjątku przez system)**
- *gh* **– *obsłuż;*** *gn* ***- zignoruj***

- *.lastevent* **lub** *!analyze –v*
- .exr, .cxr **i** *.ecxr - informacje*
- sxe, sxd, sxn **i** *sxi – sposób obsługi przez debuger*

Analiza zrzutu pamięci

- **Otwarcie pliku core dump: File | Open Crash Dump**
- **WinDbg zlokalizuje instrukcję, która była wykonywana w momencie, gdy nastąpiło utworzenie pliku core (zatem w czasie awarii)**
- **Wykorzystanie poleceń WinDbg w celu zlokalizowania błędu w programie. Należy także rozważyć użycie *!analyze -v*, które to rozszerzenie przeprowadza szereg analiz mających na celu ustalenie przyczyny awarii, a następnie wyświetla raport**

Windows Debugging Tools for Windows

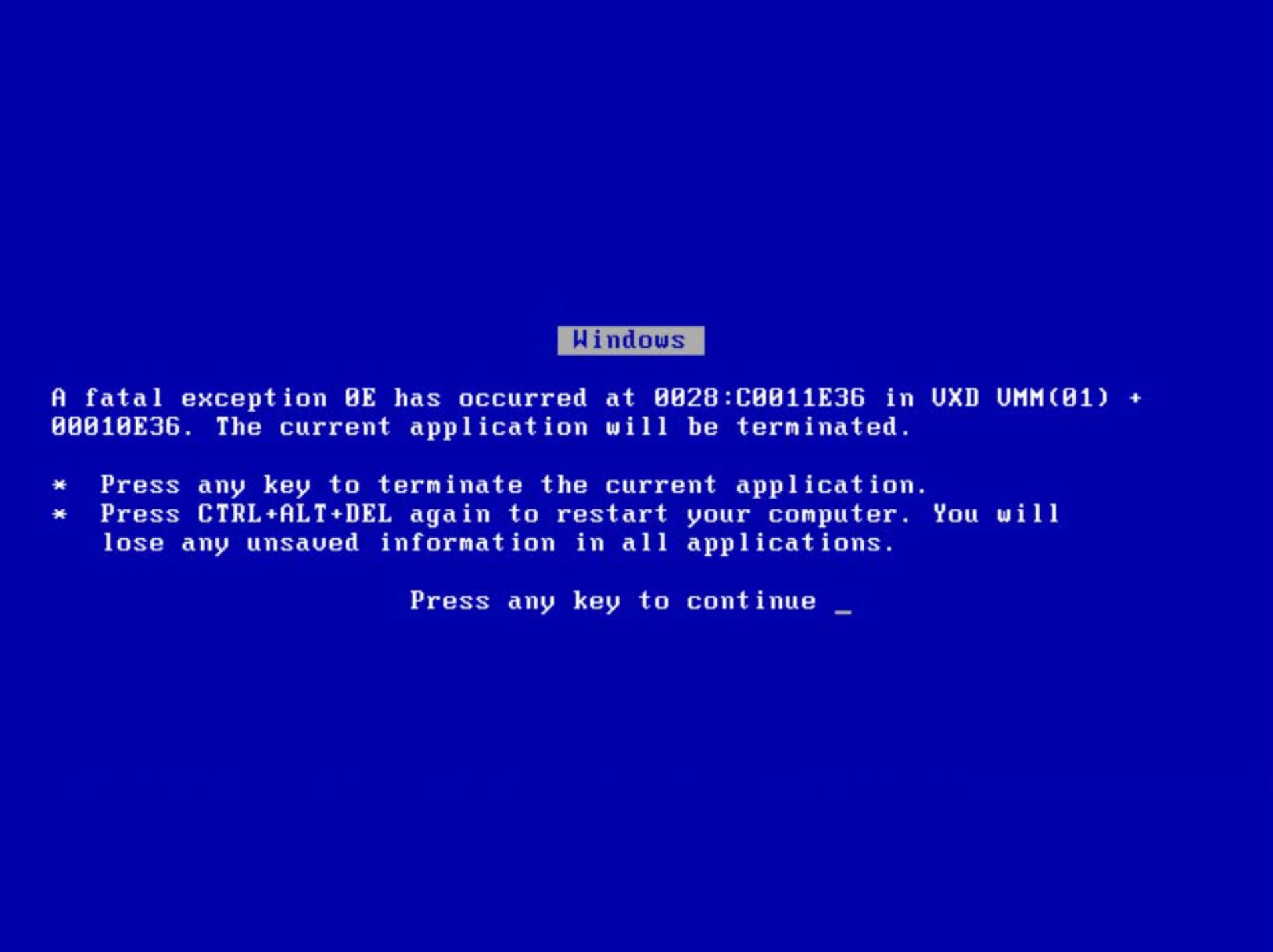
- WinDbg, KD, CDB, NTSD
- Dodatkowo:
  - Logger
  - DbgRpc
  - KDdbCtrl
  - …
- Dokumentacja


Jak to działa ?

Prosty debuger

Trzeba umieć

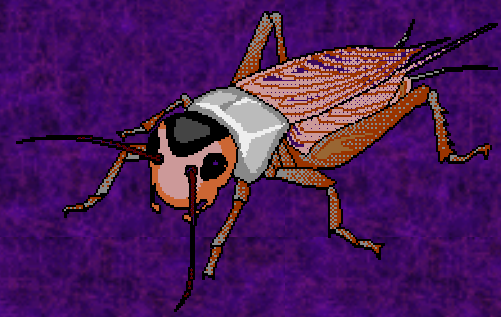
- **Czytać i modyfikować pamięć oraz rejestry debugowanego procesu**
- **Ustawiać w nim punkty przerwań**
- **Reagować na zachodzące w nim zdarzenia (w szczególności wznawiać działanie).**


Windows
A fatal exception 0E has occurred at 0028:C0011E36 in VXD VMM(01) +
00010E36. The current application will be terminated.
* Press any key to terminate the current application.
* Press CTRL+ALT+DEL again to restart your computer. You will
lose any unsaved information in all applications.
Press any key to continue _

Windows

Schemat prostego odpluskwiacza w windows może wyglądać tak

```
void main ( void ) {
    CreateProcess (..., DEBUG_ONLY_THIS_PROCESS, ...) ;
    while ( 1 == WaitForDebugEvent ( ... ) ) {
            if ( EXIT_PROCESS ) { break; }
            ContinueDebugEvent ( ... ) ;
    }
}
```



CreateProcess

- DEBUG_ONLY_THIS_PROCESS
  - Mówi, że będziemy kontrolować tworzony proces, ale nie jego procesy potomne.
  - System będzie teraz informował o zajściu odpowiednich zdarzeń
- DEBUG_PROCESS
  - Pozwala kontrolować także potomstwo odpluskwianego procesu.

WaitForDebugEvent

- Czeka na zatrzymanie odpluskwianego procesu (z uwagi na zakończenie, breakpoint, …).
- W sytuacjach praktycznych należy wołać w osobnym wątku aby uniknąć zablokowania odpluskwiacza.

ContinueDebugEvent

- Wznawia działanie procesu odpluskwianego.
- Nic innego nie może wznowić rzeczonego procesu.

Zdarzenia

System przekazuje informacje o przyczynie zatrzymania procesu w formie poniższej struktury:

```
typedef struct _DEBUG_EVENT {
    DWORD dwDebugEventCode;
    DWORD dwProcessId; DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_PROCESS_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
        RIP_INFO RipInfo;
    } u;
} DEBUG_EVENT
```

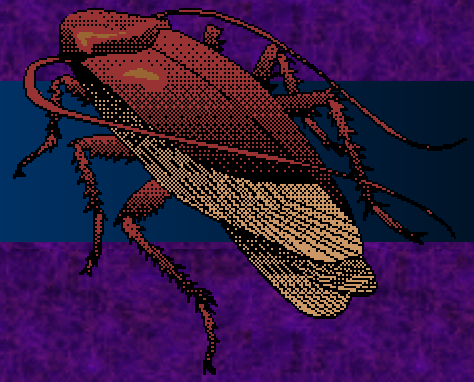
Pamięć

- Pamięć odpluskwianego procesu można czytać funkcją *ReadProcessMemory*
- Zapis umożliwia *WriteProcessMemory*
  - Pamięć może być oznaczona jako niezapisywalna, trzeba to poprawić za pomocą *VirtualProtectEx*
  - Po zapisie trzeba przywrócić ochronę.

Rejestry

- Wartości rejestrów odpluskwianego wątku można odczytać funkcją *GetThreadContext*
- Do modyfikacji służy *SetThreadContext*
- Parametrem tych funkcji jest struktura *CONTEXT*, której kształt zależy oczywiście od procesora.

Linux



```
int main() {
    int status;
    pid_t child = fork();
    if (child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl(...);
    }
    else {
        while (1) {
            wait(&status);
            if (WIFEXITED(status)) break;
            ptrace(PTRACE_CONT, child, NULL, NULL);
        }
    }
    return 0;
}
```

ptrace

- Podstawowe wywołanie systemowe służące do odpluskwiania i pokrewnych operacji.
- Z parametrem *PTRACE_TRACEME* używane w procesie odpluskwianym – m.in. sprawia, że wszelkie sygnały (nawet blokowane) zostaną dostarczone procesowi śledzącemu.
- *PTRACE_ATTACH* – pozwala śledzić istniejący proces.

wait

- Blokuje do chwili, gdy proces potomny zakończy się lub (jeśli jest śledzony) przyjdzie sygnał.
- Odpluskwiany program jest wtedy zatrzymywany
- Sygnał nie został dostarczony do śledzonego procesu.

ptrace cd.

- Polecenie *PTRACE_CONT* wznawia wykonanie śledzonego procesu.
- Pozostałe parametry mówią, czy i jaki sygnał mu dostarczyć
- *PTRACE_SYSCALL* również wznawia proces, ale powoduje jego zatrzymanie przy następnym wywołaniu/powrocie z funkcji systemowej (sygnałem *SIGTRAP*).
  - Wywołania *exec* zawsze zatrzymują proces, nawet bez *PTRACE_SYSCALL*.

Pamięć

- ptrace z poleceniem *PTRACE_PEEKTEXT* lub *PTRACE_PEEKDATA* odczytuje słowo z pamięci procesu śledzonego.
- *PTRACE_POKETEXT* lub *PTRACE_POKEDATA* zapisują słowo w ww. pamięci.
- *PTRACE_PEEKUSER* i *PTRACE_POKEUSER* służą do zapisu/odczytu pamięci zawierającej kontekst procesu.

Rejestry

- *PTRACE_PEEKUSER* i *PTRACE_POKEUSER* mogą być użyte do odczytania i modyfikacji rejestrów.
- *PTRACE_GETREGS* i *PTRACE_GETFPREGS* pozwalają odczytać wartości rejestrów.
- *PTRACE_SETREGS* i *PTRACE_SETFPREGS* modyfikują wartości rejestrów.

Punkty przerwań

W przypadku procesorów x86 wygląda to mniej więcej tak:

- Debuger wstawia w kod instrukcję INT3
- Gdy program ją wykona, system zatrzyma go i poinformuje debuger
  - Windows – zdarzeniem *EXCEPTION_DEBUG_EVENT* o wyjątku *EXCEPTION_BREAKPOINT*
  - Linux – sygnałem *SIGTRAP*
- Przed wznowieniem trzeba przywrócić nadpisaną instrukcję i cofnąć licznik

Problem

- Aby wznowić program musimy przywrócić instrukcję nadpisaną przez INT3 – ale to usunie breakpoint.
- Jeśli procesor umożliwia wykonanie krokowe (single-step execution), rozwiązanie jest łatwe.

Narzędzia do wykrywania wycieków pamięci i profilowania kodu

Bugi związane z pamięcią

Najczęściej są to:

- wycieki pamięci (gdy pamięć zaalokowana malloc nie jest potem zwalniana przez free), wielokrotne lub błędne zwalnianie pamięci,
- używanie niezaalokowanej pamięci.

Najpopularniejsze narzędzia:

- MEMWATCH,
- YAMD,
- Electric Fence,
- memcheck i adrcheck w Valgrindzie,
- MemCheck Deluxe - http://prj.softpixel.com/mcd/,
- Checker - http://www.gnu.org/software/checker/.

o tych chwilę poopowiadam

MEMWATCH

- open-source'owe narzędzie do wykrywania błędów związanych z alokacją i zwalnianiem pamięcią dla C
- http://www.linkdata.se/sourcecode.html
- użycie polega po prostu na dodaniu pliku nagłówkowego do kodu testowanego programu i zdefiniowaniu MEMWATCH w poleceniu gcc. Powstanie wtedy plik z logami wykrytych podwójnych i błędnych zwolnień oraz o niezwolnionej pamięci.

MEMWATCH (przykład użycia)

```
#include < stdlib.h>
#include < stdio.h>
#include "memwatch.h"

int main(void)
{
  char *ptr1;
  char *ptr2;

  ptr1 = malloc(512);
  ptr2 = malloc(512);

  ptr2 = ptr1;
  free(ptr2);
  free(ptr1);
}
```

Kompilujemy:

```
gcc -DMEMWATCH -DMW_STDIO test1.c memwatch.c -o test1
```

MEMWATCH (wynik)

Uruchamiamy:

```
./test1
```

Dostajemy (memwatch.log):

```
============= MEMWATCH 2.71 Copyright (C) 1992-1999 Johan Lindh =============

Started at Fri Jan  6 17:01:16 2006

Modes: __STDC__ 32-bit mwDWORD==(unsigned long)
mwROUNDALLOC==4 sizeof(mwData)==32 mwDataSize==32

double-free: <4> test1.c(15), 0x91a56b4 was freed from test1.c(14)

Stopped at Fri Jan  6 17:01:16 2006

unfreed: <2> test1.c(11), 512 bytes at 0x91a58e4         {FE FE FE FE FE FE FE
FE FE FE FE FE FE FE FE FE ................}

Memory usage statistics (global):
 N)umber of allocations made: 2
 L)argest memory usage      : 1024
 T)otal of all alloc() calls: 1024
 U)nfreed bytes totals      : 512
```

MEMWATCH (jak to działa?)

- za pomocą preprocesora C podmienia nasze wywołania funkcji związanych z alokowaniem i zwalnianiem pamięci swoimi,
- w których zapamiętuje wszystkie alokacje i zwolnienia
- i na tej podstawie potem generuje raport.

YAMD (Yet Another Memory Debugger)

- kolejne narzędzie do wykrywania błędów związanych z pamięcią
- ze względu na operowanie na dość niskopoziomowych funkcjach działa zarówno dla C, jak i dla C++ wykrywając wszystkie operacje na pamięci
- http://www.cs.hmc.edu/~nate/yamd/
- należy rozpakować, zmake'ować i zainstalować (make install)

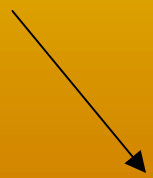
YAMD (przykład)

```
#include < stdlib.h>
#include < stdio.h>

int main(void)
{
  char *ptr1;
  char *ptr2;
  char *chptr;
  int i = 1;
  ptr1 = malloc(512);
  ptr2 = malloc(512);
  ptr2 = ptr1;
  free(ptr2);
  free(ptr1);
  chptr = (char *)malloc(512);
  for (i; i <= 512; i++) {
    chptr[i] = 'S';
  }
  free(chptr);
}
```

Kompilujemy:

```
gcc -g test2.c -o test2
```



Uruchamiamy:

```
yamd-run ./test2
```

YAMD (wynik)

```
YAMD version 0.32
Starting run: ./test2
Executable: /home/michal/yamd-0.32/test2
Virtual program size is 1528 K
Time is Fri Jan  6 18:26:15 2006

default_alignment = 1
min_log_level = 1a
repair_corrupted = 0
die_on_corrupted = 1
check_front = 0

INFO: Normal allocation of this block
Address 0xb7ff2e00, size 512
Allocated by malloc at
        /lib/tls/libc.so.6(malloc+0x35)[0x410045]
        ./test2[0x8048b48]
        /lib/tls/libc.so.6(__libc_start_main+0xe4)[0x3c4ad4]
        ./test2[0x8048a79]

INFO: Normal allocation of this block
Address 0xb7fefe00, size 512
Allocated by malloc at
        /lib/tls/libc.so.6(malloc+0x35)[0x410045]
        ./test2[0x8048b5b]
        /lib/tls/libc.so.6(__libc_start_main+0xe4)[0x3c4ad4]
        ./test2[0x8048a79]

INFO: Normal deallocation of this block
Address 0xb7ff2e00, size 512
Allocated by malloc at
        /lib/tls/libc.so.6(malloc+0x35)[0x410045]
        ./test2[0x8048b48]
        /lib/tls/libc.so.6(__libc_start_main+0xe4)[0x3c4ad4]
        ./test2[0x8048a79]
Freed by free at
        /lib/tls/libc.so.6(__libc_free+0x35)[0x4126e5]
        ./test2[0x8048b72]
        /lib/tls/libc.so.6(__libc_start_main+0xe4)[0x3c4ad4]
        ./test2[0x8048a79]
a
```

```
ERROR: Multiple freeing
At
        /lib/tls/libc.so.6(__libc_free+0x35)[0x4126e5]
        ./test2[0x8048b80]

/lib/tls/libc.so.6(__libc_start_main+0xe4)[0x3c4ad4]
        ./test2[0x8048a79]
free of pointer already freed
Address 0xb7ff2e00, size 512
Allocated by malloc at
        /lib/tls/libc.so.6(malloc+0x35)[0x410045]
        ./test2[0x8048b48]

/lib/tls/libc.so.6(__libc_start_main+0xe4)[0x3c4ad4]
        ./test2[0x8048a79]
Freed by free at
        /lib/tls/libc.so.6(__libc_free+0x35)[0x4126e5]
        ./test2[0x8048b72]

/lib/tls/libc.so.6(__libc_start_main+0xe4)[0x3c4ad4]
        ./test2[0x8048a79]

INFO: Normal allocation of this block
Address 0xb7fece00, size 512
Allocated by malloc at
        /lib/tls/libc.so.6(malloc+0x35)[0x410045]
        ./test2[0x8048b90]

/lib/tls/libc.so.6(__libc_start_main+0xe4)[0x3c4ad4]
        ./test2[0x8048a79]

ERROR: Crash
        ./test2[0x8048ba5]

/lib/tls/libc.so.6(__libc_start_main+0xe4)[0x3c4ad4]
        ./test2[0x8048a79]
Tried to write address 0xb7fed000
Seems to be part of this block:
Address 0xb7fece00, size 512
Allocated by malloc at
        /lib/tls/libc.so.6(malloc+0x35)[0x410045]
        ./test2[0x8048b90]

/lib/tls/libc.so.6(__libc_start_main+0xe4)[0x3c4ad4]
        ./test2[0x8048a79]
Address in question is at offset 512 (out of bounds)
Will dump core after checking heap.
```

Electric Fence

- biblioteka do wykrywania błędów związanych z pamięcią,
- wchodzi ona w skład wielu dystrybucji Linuxa,
- wykrywa zarówno czytanie, jak i pisanie poza zaalokowanym obszarem pamięci i zwolnionego już obszaru pamięci,
- można spokojnie używać go razem z gdb

jak to działa?

- biblioteka libefence.a implementuje na nowo funkcje:

```
void * malloc (size_t size);
void free (void *ptr);
void * realloc (void *ptr, size_t size);
void * calloc (size_t nelem, size_t elsize);
void * memalign (size_t alignment, size_t size);
void * valloc (size_t size);
```

- malloc w Electric Fence działa tak, że alokuje dwie lub więcej strony, z tym, że ostatnią z nich czyni niedostępną, zwracając taki wskaźnik, żeby pierwszy bajt za żądanym rozmiarem wypadał na początek niedostępnej strony.
- są pewne problemy – czasami, zby niektóre biblioteki działały dobrze malloc musi zaalokować nieco więcej pamięci, bo powoduje, że sprawdzanie jest mniej dokładne
- podobnie rozwiązywane jest sprawdzanie pamięci przed zaalokowaną i sprawdzanie zwolnionej pamięci

Electric Fence (ustawienia)

Zmienne środowiskowe:

EF_ALIGNMENT	Ustala do ilu bajtów dopełnia alokowaną pamięć. Domyślnie do sizeof(int)
EF_PROTECT_BELOW	Ustawione na 0 lub 1 odpowiedio wyłącza i włącza zabezpieczenie też pamięć przed zaalokowaną pamięcią
EF_PROTECT_FREE	Ustawione na 0 lub 1 odpowiedio wyłącza i włącza zabezpieczenie pamięci zwolnionej
EF_ALLOW_MALLOC_0	Ustawione na 0 lub 1 odpowiedio wyłącza i włącza tolerancję dla wywołań malloca na 0 bajtów.
E_FILL	Jeśli jest ustawione na wartość z pomiędzy 0 i 255 każdy bajt alokowanej pamięci jest ustawiny na tę wartość.

Electric Fence (użycie)

- wystarczy zlinkować nasz program z biblioteką libefence.a, z reguły wystarczy opcaja -lefence
- i uruchomić program, w przypadku wystąpienia błędu dostaniemy odpowidni komunikat, np.:

```
Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens
ElectricFence Aborting: free(b7ee5e00): address not from malloc().
```

ogólny schemat debuggowania:

- zlinkować program z biblioteką Electric Fence,
- uruchomić program pod debuggerem, poprawić błędy,
- włączyć EF_PROTECT_BELOW,
- uruchomić program pod debuggerem, poprawić błędy,
- ewentualnie ustawić EF_ALIGMENT na 0 i powtórzyć procedurę.

memcheck i adrcheck w Valgrindzie

- Valgrind to kolejne narzędzie zorientowane przede wszystkim na kompleksową kontrolę błędów związanych z pamięcią, ale nie tylko.
- http://valgrind.org/
- teraz tylko krótkie wprowadzenie
- Valgrind to zbiór kilku narzędzi
- nas chwilowo będą interesowały dwa -- memcheck i adrcheck.
- obydwa dokonują kompleksowej kontroli programu jeżeli chodzi o pamięć, ale memcheck dodatkowo sprawdza, czy nigdzie nie czyta się z niezainicjowanych komórek pamięci, za to adrcheck jest dużo szybszy.
- aby sprawdzić program wystarczy napisać:

```
valgrind --tool=nazwa_narzędzia program [argumenty]
```

memcheck w Valgrindzie (przykładowy wynik)

```
==12926== Memcheck, a memory error detector.
==12926== Copyright (C) 2002-2005, and GNU GPL'd, by Julian Seward et al.
==12926== Using LibVEX rev 1471, a library for dynamic binary translation.
==12926== Copyright (C) 2004-2005, and GNU GPL'd, by OpenWorks LLP.
==12926== Using valgrind-3.1.0, a dynamic binary instrumentation framework.
==12926== Copyright (C) 2000-2005, and GNU GPL'd, by Julian Seward et al.
==12926== For more details, rerun with: -v
==12926==
==12926== My PID = 12926, parent PID = 5295.  Prog and args are:
==12926==    ./test2
==12926==
==12926== Invalid free() / delete / delete[]
==12926==    at 0x4003F62: free (vg_replace_malloc.c:235)
==12926==    by 0x8048403: main (in /home/michal/valgrind-3.1.0/test2)
==12926==  Address 0x401F028 is 0 bytes inside a block of size 512 free'd
==12926==    at 0x4003F62: free (vg_replace_malloc.c:235)
==12926==    by 0x80483F5: main (in /home/michal/valgrind-3.1.0/test2)
==12926==
==12926== Invalid write of size 1
==12926==    at 0x8048429: main (in /home/michal/valgrind-3.1.0/test2)
==12926==  Address 0x401F688 is 0 bytes after a block of size 512 alloc'd
==12926==    at 0x400346D: malloc (vg_replace_malloc.c:149)
==12926==    by 0x8048413: main (in /home/michal/valgrind-3.1.0/test2)
==12926==
==12926== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 12 from 1)
==12926== malloc/free: in use at exit: 512 bytes in 1 blocks.
==12926== malloc/free: 3 allocs, 3 frees, 1,536 bytes allocated.
==12926== For counts of detected errors, rerun with: -v
==12926== searching for pointers to 1 not-freed blocks.
==12926== checked 42,548 bytes.
==12926==
==12926== LEAK SUMMARY:
==12926==    definitely lost: 512 bytes in 1 blocks.
==12926==      possibly lost: 0 bytes in 0 blocks.
==12926==    still reachable: 0 bytes in 0 blocks.
==12926==         suppressed: 0 bytes in 0 blocks.
==12926== Use --leak-check=full to see details of leaked memory.
```

Inne narzędzia

- MemCheck Deluxe - http://prj.softpixel.com/mcd/,
- Checker - http://www.gnu.org/software/checker/.

Profilowanie kodu

metody programistyczne

- blokady
- liczniki

```
extern struct timeval time;

boolean_t   mymutex_try(mymutex_t *lock) {
    int ret;
    ret=mutex_try(lock->mutex);
    if (ret) {
        lock->tryfailcount++;
    }
    return ret;
}

void    mymutex_lock(mymutex_t *lock) {
    if (!(mymutex_try(lock))) {
        mutex_lock(lock->mutex);
    }
    lock->starttime = time.tv_sec;
}

void    mymutex_unlock(mymutex_t *lock) {
    lock->lockheldtime += (time.tv_sec -
                lock->starttime);
    lock->heldcount++;
    mutex_unlock(lock->mutex);
}
```

co to i po co to?

- profilowanie kodu to badanie jak często który kawałek kodu jest wykonywany.
- dzięki takiej wiedzy, możemy zdecydować optymalizacji i udoskonalaniu wydajności której części kodu powinniśmy poświęcić najwięcej czasu.
- przy skomplikowanych programach, z wieloma modułami, sprawa nie jest wcale prosta.

```
#ifdef PROFILING
            counter++;
#endif
```

w szczególności do profilowania sekcji krytycznych

gprof

- gprof jest narzędziem załączanym do większości destrybucji Linuxa
- jego działanie opisuje odpowiednia strona man
- aby użyć gprofa trzeba skompilować swój program z opcjami -pg i -g i uruchomić go
- powstanie wtedy plik gmon.out z profilami
- aby go zinterpretować wystarczy uruchomić gprofa:

```
gprof nazwa_pliku_wykonywalnego
```

- opcje (m.in.):
  - -p to flat profile
  - -q to call graph

gprof (flat profile)

sumy czasów, które program spędził na wykonaniu konkretnych funkcji

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
```

gprof (call graph)

jak wiele czasu spędziła funkcja na siebie i jak wiele czasu zajęły jej podwywołania

```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1          main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1          report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp < cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01        0.02       8+472     < cycle 2 as a whole>   [4]
                0.01        0.02     244+260         offtime < cycle 2> [7]
                0.00        0.00     236+1           tzset < cycle 2> [26]
-----------------------------------------------
```

gcov

- gcov pozwala na wypisanie ile razy została użyta każda linia kodu,
- Aby go użyć kompilujemy program z opcjami -fprofile-arcs -ftest-coverage
- Uruchomienie programu powoduje stworzenie informacji w plikach ".da" dla każdego pliku skompilowanego przy użyciu opisanych opcji
- Wtedy wystarczy zapuścić gcov dla naszego kodu źródłowego: `gcov tescik.c`
- I dostaniemy następujący plik tescik.c.gcov:

```
main() {
 1          int i, total;
 1          total = 0;
 11         for (i = 0; i < 10; i++)
 10         total += i;
 1          if (total != 45)
######      printf ("Failure\n"); else
 1          printf ("Success\n");
 1          }
```

Urządzenie prof

- prof jest urządzeniem służącym do profilowania modułów,
- działanie opisuje odpowiednia strona man,
- dostarcza dwupoziomową strukturę katalogową,
- na pierwszym poziomie mamy plik kontrolny ctl i zero lub więcej ponumerowanych katalogów, każdy związany z jakimś modułem, który profilujemy

- do pliku ctl można pisać:

module nazwa	dodaje moduł do profilowania
start	rozpoczyna profilowanie dodanych modułów, a gdy ich nie ma, to wszystkich modułów jądra, poprzez próbkowanie
startcp	rozpoczyna profilowanie pokrywające (ang. coverage), to znaczy każda wywoływana funkcja w załączonych modułach jest liczona
startmp	jak wyżej, ale profilowanie pamięci
stop	zatrzymuje profilowanie
end	zatrzymuje profilowanie, zwalnia pamięć modułów i wyłącza je spod profilowania
interval i	zmiena odstępy między próbkowaniem na i ms. Domyślnie 100 ms.

- katalog dotyczący konkretnego modułu zawiera pliki:

name	z którego można przeczytać nazwę modułu
path	z którego można przeczytać ścieżkę do modułu
pctl	nieużywany
histogram	z którego można przeczytać wyliczoną statystykę

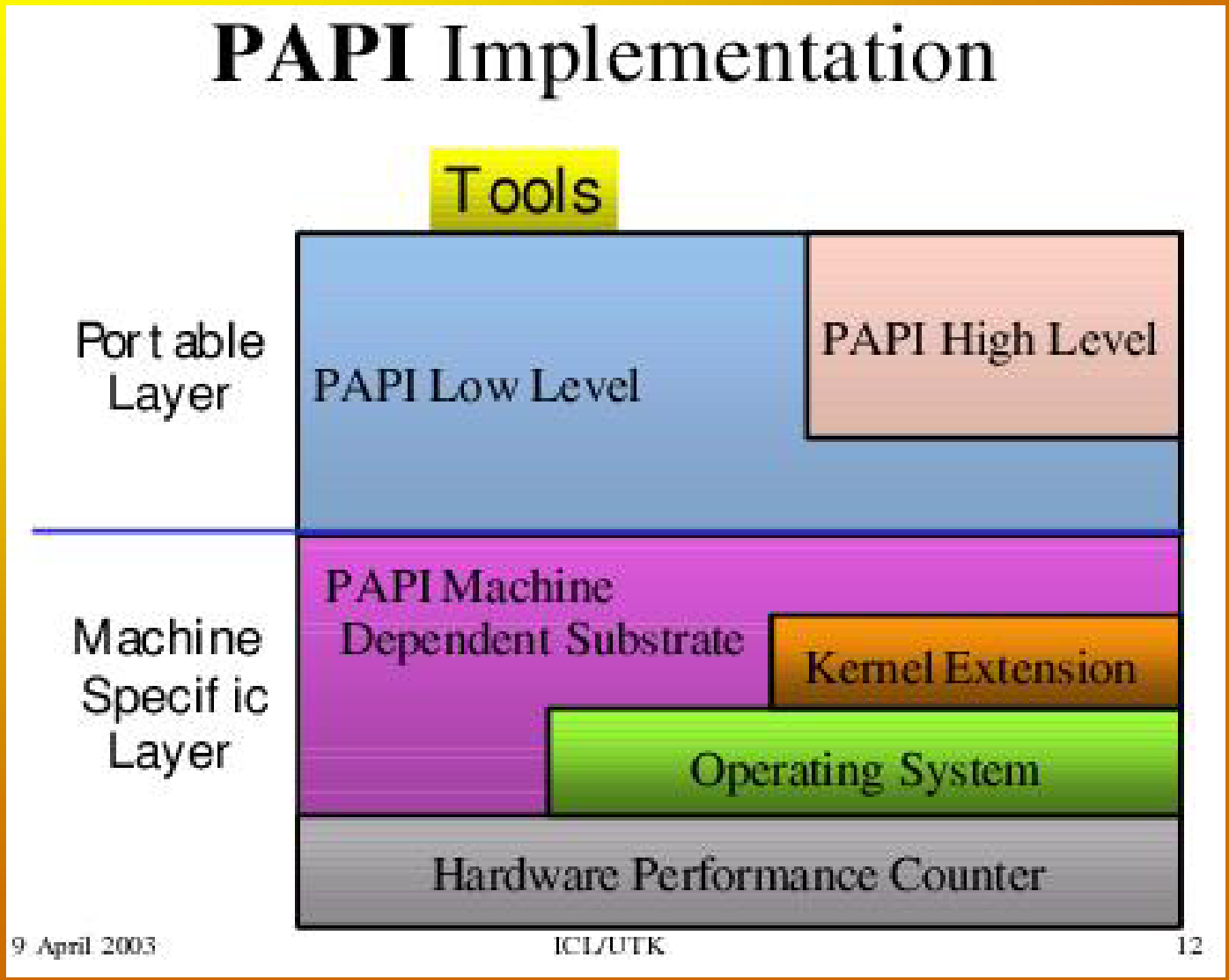
- istnieje też uboższe, ale popularniejsze urządzenie profile powiązane z narzędziem readprofile

PAPI (Performance Application Programing Interface)

- PAPI (Performance Application Programing Interface) to dość duże narzędzie a właściwie duży projekt
- http://icl.cs.utk.edu/papi/index.html
- sedno stanowi dostarczenie interfejsu (do C i Fortrana) do obsługi różnych liczników
- PAPI dysponuje wersjami na różne platformy i systemy operacyjne. Zajmiemy się tutaj linuxową wersją.
- PAPI zapewnie trzy interfejsy:
  - low-level, dla wymagających i zainteresowanych użytkowników,
  - high-level,
  - graficzny do wizualizacji.

do C i Fortrana

PAPI (Architektura)



PAPI (zdarzenia rzeczywiste i predefiniowane)

- PAPI w pliku papiStdEventsDefs.h definiuje zbiór najbardziej wpływających na wydajność zdarzeń. To tak zwane zdarzenia predefiniowane (ang. preset). Zdarzenia te są mapowane na rzeczywiste zdarzenia (po ang. nazywa się to native) dla danej platformy. Zdarzenia to wystąpienia określonych sygnałów związanych z funkcjami procesora. Każdy procesor ma pewną liczbę rzeczywistych zdarzeń dla niego.
- Rzeczywiste zdarzenia obejmują wszystkie zdarzenia zliczane przez CPU, czyli zależne od platformy. PAPI zapewnia dostęp do tych zadrzeń przez low-level interface (interfejs jest niezależny od platformy).
- Zdarzenia predefiniowane (ok. stu!) też z reguły są zliczane przez CPU zapewniające liczniki wydajności, więc z reguły mapowanie zdarzeń predefiniowanych na rzeczywiste jest proste. Zdarzenia te zapewniają informacje o wykorzystaniu hierarchii pamięci, przetwarzania potokowego, dają dostęp do licznika instrukcji itp. Niekiedy są one jakąś kombinacją zdarzeń rzeczywistych.
- Bardzo dużą tabelę zamieszczemy w materiałach, tu tylko jej fragment ;)

PRESET NAME	DESCRIPTION	AMD ATHLON K7	IBM POWER3	INTEL/HP ITANIUM	INTEL PENTIUM III	MIPS R12K	U SI
PAPI_L1_DCM	Level 1 data cache misses	v	v	v	v	v	x
PAPI_L1_IC	Level 1 instruction cache						

PAPI (high-level)

- plik nagłówkowy: papi.h

PAPI_num_counters()	funkcja inicjalizujuje PAPI (jeśli trzeba), zwraca optymalną interfejsu high-level, związaną z liczbą liczników.
PAPIF_start_counters(*events, array_length)	inincjalizuje PAPI (jeśli trzeba) i startuje liczenie zdarzeń z tablicy events o długości array_length.
PAPI_flops(*real_time, *proc_time, *flpins, *mflops)	po prostu inicjalizuje PAPI (jeśli trzeba), i ustawia wartości wskazywane przez wskaźniki (od ostatniego wywołania PAPI_flops): real_time -- czas rzeczywisty, proc_time -- czas wirtualny (tyle ile dany proces był wyonywany), flpins -- liczba wykonanych instrukcji zmiennoprzecinkowych, mflops -- Mflop/s rating.
PAPI_read_counters(*values, array_length)	Wstawia do tablicy values bieżące wartości liczników i je wyzerowuje.
PAPI_accum_counters(*values, array_length)	Dodaje do tablicy values bieżące wartości wskaźników i je wyzerowuje.
PAPI_stop_counters(*values, array_length)	Wstawia bieżące wartości licznikow do tablicy values i kończy ich wyliczanie.

PAPI (low-level)

- interfejs nisko-poziomowy udostępnia dużo więcej, nisko-poziomowych funkcji, także do obsługi rzeczywistych zdarzeń

PAPI (przykład 1.)

```
#include <papi.h>

main()
{
     int Events[2] = { PAPI_TOT_CYC, PAPI_TOT_INS };
     int num_hwcntrs = 0;

     /* Initialize the PAPI library and get the number of
     counters available */
     if ((num_hwcntrs = PAPI_num_counters()) <= PAPI_OK)
         handle_error(1);

     printf("This system has %d available counters.",
        num_hwcntrs);

     if (num_hwcntrs > 2)
         num_hwcntrs = 2;

     /* Start counting events */
     if (PAPI_start_counters(Events, num_hwcntrs) != PAPI_OK)
         handle_error(1);
}
```

PAPI (przykład 2.)

```
#include <papi.h>

#define NUM_EVENTS 2

main()
{
     int Events[NUM_EVENTS] = {PAPI_TOT_INS, PAPI_TOT_CYC};
     long_long values[NUM_EVENTS];

     /* Start counting events */
     if (PAPI_start_counters(Events, NUM_EVENTS) != PAPI_OK)
        handle_error(1);

     /* Do some computation here*/

     /* Read the counters */
     if (PAPI_read_counters(values, NUM_EVENTS) != PAPI_OK)
        handle_error(1);

     /* Do some computation here */

     /* Stop counting events */
     if (PAPI_stop_counters(values, NUM_EVENTS) != PAPI_OK)
        handle_error(1);
}
```

DynaProf

- DynaProf jest open-source'owym, ciągle rozwijanym narzędziem do profilowania aplikacji
- Dynaprof mierzy zarówno po prostu czas, jak i dane dające się zmierzyć poprzez PAPI (korzysta z PAPI)
- Działa na plikach wykonywalnych, niepotrzebne mu źródło
- http://www.cs.utk.edu/~mucci/dynaprof/

Inne narzędzia

Wiele ich jest:

- IBM: prof, gprof, tprof,
- SGI: ssrun, prof, perfex,
- HP/Compaq: prof, pixie, gprof, hiprof, uprofile,
- Giude View,
- Vampir,
- Vprof,
- HPCView,
- po części Valgrind

Odpluskwianie w jądrze

Use the Source, Luke, use the Source. Be one with the code.'. Think of Luke Skywalker discarding the automatic firing system when closing on the deathstar, and firing the proton torpedo (or whatever) manually. _Then_ do you have the right mindset for fixing kernel bugs.

Linus

W kwestii samych błędów

(a raczej tego, co widzi użytkownik systemu, gdy następi błąd), to należy rozróżnić cztery sytuacje:

- wszystko działa, ale nie tak, jak trzeba,
- „się zawiesił" (lockups)
- zgłaszany jest błąd oops (normalny sposób informowania użytkonkia o nieprawidłowościach w działaniu jądra)
- kernel panic -- błąd paniczny powodujący załamanie się systemu.

Opcje konfiguracji

W testowaniu kodu jądra i debuggowaniu przydatne mogą się okazać dostępne liczne opcje konfiguracyjne kompilacji jądra. Opcje te są zebrane w dziale Kernel hacking. Aktywność wszystkich tych opcji uzależniona jest od włączenia ogólnej opcji CONFIG_DEBUG_KERNEL. Jeżeli zamierzamy modyfikować jądro warto włączyć je wszystkie. W niniejszej prezentacji opisane są niektóre (częściej używane) z tych opcji.

wyprowadzanie informacji

- funkcja printk(), demony klogd i syslogd
- makro BUG()
- wywołanie panic()
- wydruk śladu stosu

printk()

- funkcja printk() działa niemal identycznie, jak funkcja printf() z biblioteki dostępnej przy programowaniu przestrzeni użytkownika.
- jej przydatność polega właśnie na tym, że jest niezawodna i wszechstronna
- z jednym wyjątkiem: nie da się jej wywołać we wczesnej fazie rozruchu jądra, niektórzy programiści wyprowadzają wtedy komunikaty na sprzęt, który działa zawsze, np. na port szeregowy. Istnieje jednak jeszcze słabo przenośna i nie na każdej architekturze zaimplementowana fukcja early_printk()
- zdolność tej pierwszej do określania poziomu rejestrowania (ang. loglevel)

```
printk(KERN_WARNING "Minister zdrowia ostrzega: palenie tytoniu
powoduje choroby płuc.\n");

printk(KERN_INFO "To jest prezentacja o odpluskwianiu.\n");

printk("Nie został określony poziom rejestrowania.\n");
```

printk() (loglevel bardziej szczegółowo)

- stałe zdefiniowane są w pliku linux/kernel.h
- wartości dołączane są po prostu na początek komunikatu. Jądro orównuje poziom z bieżącym poziomem rejestrowania (console_loglevel) i na tej podstawie decyduje, czy komunikat powinien być wyświetlony w konsoli, czy przekierowany gdzie indziej

KERN_EMERG	<0>	Sytuacja awaryjna
KERN_ALERT	<1>	Problem wymagający natychmiastowej interwencji
KERN_CRIT	<2>	Sytuacja krytyczna
KERN_ERR	<3>	Błąd
KERN_WARNING	<4>	Ostrzeżenie
KERN_NOTICE	<5>	Sytuacja normalna, warta odnotowania
KERN_INFO	<6>	Informacja
KERN_DEBUG	<7>	Komunikat diagnostyczny

- w przykadku nieokreślenia poziomu, za jego wartość przyjmowany jest DEFAULT_MESSAGE_LOGLEVEL.

bufor cykliczny, klogd, syslogd

- komunikaty jądra umieszczane są w cyklicznym buforze o rozmiarze LOG_BUF_LEN. Można go konfigurować przy kompilacji bodajże za pomocą opcji CONFIG_LOG_BUF_SHIFT. Z reguły jest to 16kB.
- za pobieranie komunikatów jądra z bufora odpowiedzialny jest demon przestrzeni użytkownika klogd. Demon ten zapisuje komunikaty do systemowego pliku dziennika, korzystając przy tym z pomocy demona syslogd. klogd w celu odczytywania komunikatów korzysta z pliku /proc/kmsg (bądź też z systemowego wywołania syslog()). klog budzi się, gdy przychodzi nowy komunikat i przekazuje go do demona syslogd.
- syslogd natomiast dostarcza wywołane komunikaty do pliku (domyślnie jest to plik /var/log/messages). Działanie demona można konfigurować w pliku /etc/syslog.conf

makro BUG()

- powoduje zgłoszenie błędu oops wraz z wszelkimi związanymi informacjami
- w większości architektur BUG() jest po prostu rozwijane do jakiejś niedozwolonej operacji i w ten sposób generowany jest błąd oops

```
if (zle_sie_dzieje) BUG();
```

- równoważne

```
BUG_ON(zle_sie_dzieje);
```

wywołanie panic()

- błędy bardziej krytyczne mogą być zgłaszane wywołaniem panic()
- powoduje wyświetlenie komunikatu o błędzie i zatrzymanie systemu

```
if (straszny_i_okropny_blad) panic("Maksymalna kicha!\n");
```

ślad stosu

```
if (trzeba_analizowac_slad_stosu) dump_stack();
```

Analiza błędów oops

Reakcja systemu, oops vs. kernel panic

- Blad oops to standardowy sposób informowania użytkownika o nieprawidłowościach w działaniu jądra. Zgłoszenie błędu oops polega na wyświetleniu komunikatu o błędzie wraz z zawartością rejestrów i śladem wykonania (ang. backtrace). Niekiedy po zakończeniu obsługi pojawia się niespójność jądra. Konieczne jest wtedy ostrożne wycofanie się do poprzedniego kontekstu i przywrócenie kontroli nad systemem.

- Czasem jest to niemożliwe. Jeśli błąd wystąpi w kontekście przerwania, jądro nie może nic zrobić i „panikuje" dając w efeckie błąd paniczny (kernel panic). Pojawia się on też przy błędzie podczas wykonywania procesu jałowego lub procesu init. Jedynie wystąpienie oopsa w kontekście jednego ze zwykłych procesów daje możliwość unicestwienia tego procesu i kontynuowania działania reszty systemu.

Co nam mówi 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
```

Opis

Licznik oopsów

wykonywana instr.

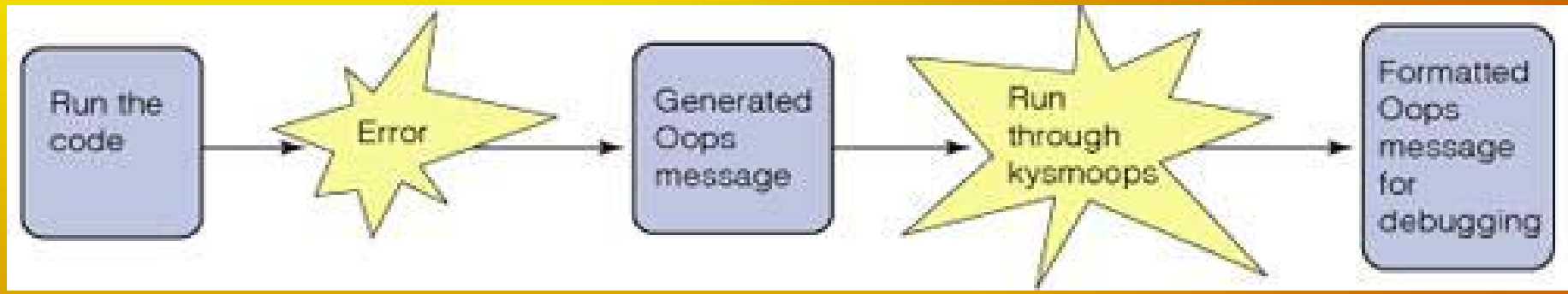
flagi i rejestry

Stos

Ślad wykonania

ksymoops

- EIP = adres bazowy funkcji + offset instrukcji
- ślad wykonania to łańcuch wywołania funkcji prowadzący do błędu
- aby dokładnie go przeanalizować możemy zdekodować naszego oopsa za pomocą polecenia ksymoops `ksymoops zapis_oops.txt`



- wtedy adresy wymienione w śladzie wykonania zostaną przetłumaczone na nazwy symboliczne funkcji

System.map

- ksymoops korzysta w tym celu z pliku System.map generowanego w czasie kompilacji jądra oraz informacji o załadowanych dynamicznie modułach
- oto przykładowy fragment pliku System.map:

```
....
c017cdf0 T reiserfs_dir_fsync
c017ce80 t reiserfs_readdir
c017d2f0 t create_virtual_node
c017d780 t check_left
c017d8d0 t check_right
....
```

ksymoops (wynik)

```
>>EIP; c017d558 <create_virtual_node+298/490>   <=====
Trace; c017eb4f <ip_check_balance+34f/ae0>
Trace; c017fc44 <reiserfs_kfree+14/50>
Trace; c0180115 <fix_nodes+115/450>
Trace; c018a1c8 <reiserfs_insert_item+88/110>
Trace; c017bb3a <reiserfs_new_inode+3da/500>
Trace; c018738f <pathrelse+1f/30>
Trace; c0177a13 <reiserfs_lookup+73/d0>
Trace; d0871044 <END_OF_CODE+9a77/???
Trace; c0178274 <reiserfs_mkdir+d4/1d0>
Trace; c0142e36 <d_alloc+16/160>
Trace; c013c75f <vfs_mkdir+7f/b0>
Trace; c013c7f8 <sys_mkdir+68/b0>
Trace; c0108f77 <system_call+33/38>
Trace; c010002b <startup_32+2b/139>

Code;  c017d558 <create_virtual_node+298/490>
00000000 <_EIP>:
Code;  c017d558 <create_virtual_node+298/490>   <=====
   0:   8b 40 14                 mov    0x14(%eax),%eax   <=====
Code;  c017d55b <create_virtual_node+29b/490>
   3:   ff d0                    call   *%eax
```

kallsyms

- od wersji rozwojowej jądra 2.5
- aktywowana przy użyciu opcji konfiguracyjnej CONFIG_KALLSYMS
- uaktywnienie tej opcji powoduje umieszczenie w obrazie jądra odwzorowań nazw symbolicznych na adresy pamięci, dzięki czemu jądro może wyprowadzać zdekodowane ślady wykonania
- i wtedy ksymoops nie jest już potrzebny

co dalej?

- możemy użyć informcji o offsecie (u nas 298) do odnalezienia konkretnej instrukcji, która błąd spowodowała
- w tym celu potrzebujemy kopii problematycznej funkcji w postaci kodu assemlerowego
- możemy ją uzyskać za pomocą narzędzia objdump
- przydatna może być także zawartość rejestrów. Można na podstawie zawartości rejestrów odtworzyć stan wykonania w momencie wystąpienia błędu.
- a potem będziemy się zastanawiać od jakiej linii z kodu w C to pochodzi

objdump

```
objdump -d plik.o
```

```
25d:    b8 02 00 00 00          mov    $0x2,%eax
262:    74 05                   je     269 <create_virtual_node+0x269>
264:    b8 0f 00 00 00          mov    $0xf,%eax
269:    89 c1                   mov    %eax,%ecx
26b:    eb 10                   jmp    27d <create_virtual_node+0x27d>
26d:    8d 76 00                lea    0x0(%esi),%esi
270:    0f b6 4a 0f             movzbl 0xf(%edx),%ecx
274:    c0 e9 04                shr    $0x4,%cl
277:    0f b6 c1                movzbl %cl,%eax
27a:    0f b7 c8                movzwl %ax,%ecx
27d:    8b 5c 24 24             mov    0x24(%esp,1),%ebx
281:    89 c8                   mov    %ecx,%eax
283:    8b 04 85 00 00 00 00    mov    0x0(,%eax,4),%eax
28a:    8b 8b 24 01 00 00       mov    0x124(%ebx),%ecx
290:    51                      push   %ecx
291:    8b 54 24 08             mov    0x8(%esp,1),%edx
295:    52                      push   %edx
296:    57                      push   %edi
297:    56                      push   %esi
298:    8b 40 14                mov    0x14(%eax),%eax
29b:    ff d0                   call   *%eax
29d:    89 c2                   mov    %eax,%edx
29f:    8b 06                   mov    (%esi),%eax
2a1:    83 c4 10                add    $0x10,%esp
2a4:    01 c2                   add    %eax,%edx
```

gdzie to jest w C?

- nie ma jakiejś prostej metody
- należy zwrócić uwagę, że z reguły:
  - if-y, while-e i case-y objawiają się poprzez porównania i skoki
  - return-y objawiają się jako długie skoki na koniec funkcji
  - wywołania funkcji poprzez odkładanie na stos i wołanie funkcji
- dość charakterystyczne jest także korzystanie ze spinlocków, użycie wskaźników, operacje na pamięci i operacje arytmetyczne.

Radzenie sobie z lockupami

Rodzaje lockupów:

- lockupy związane z hardwarem,
- lockupy z umożliwionymi przerwaniami,
- lockupy z zablokowanymi przerwaniami.

Lockupy z umożliwionymi przerwaniami

• mamy z nimi do czynienia, gdy utknęliśmy w nieskończonej pętli lub wisimy na jakimś locku

• bardzo łatwo sprawdzić, czy przerwania nie są zablokowane. Choćby świadczy o tym reakcja światełek na klawiaturze (np. na klawisz Caps Lock).

klawisze sysinfo

shift + scroll lock	informacje o pamięci
ctrl + scroll lock	informacje o stanie procesora
prawy alt + scroll lock	informacje o rejestrach i ślad wykonania

Magic sysrq key

- można aktywować za pośrednicrtwem opcji konfiguracyjnej CONFIG_MAGIC_SYSRQ
- można ją też włączyć za pomocą odpowiedniego pliku procowego:

```
echo 1 > /proc/sys/kernel/sysrq
```

- na większości klawiatur znajduje się klawisz SysRq (jest to z reguły alt + print screen).

SysRq + :

h	pomoc dla sysrq key
s	sync
u	przemontowuje wszystkie partycje tylko do odczytu
b	rebootuje system
o	wyłącza komputer
r	zmienia tryb klawiatury na XLATE (wyłącza raw)
p	informacje o rejestrach i ślad wykonania
t	wypisuje procesy i informacje o nich
k	killuje proces na bieżącej konsoli
e	wysyła wszystkim oprócz init SIGTERM
i	wysyła wszystkim oprócz init SIGKILL
l	wysyła wszystkim SIGKILL
m	wypisuje informacje o pamięci
0-8	zmienia console_loglevel

Lockupy z zablokowanymi przerwaniami

• NMI Watchdog potrafi wykrywać sytuacje, w których mamy do czynienia z takim lockupem (sprawdza czy w przeciągu kilka sekund było jakieś przerwanie) i w razie stwierdzenia takiej sytuacji automatycznie generuje oopsa.

• Jedną z opcji konfiguracyjnych (począwszy od wersji rozwojowej jądra 2.5) jest opcja określana w menu konfiguracyjnym sleep-inside-spinlock-checking -- wtedy też dostaniemy automatycznego oopsa w przypadku zawieszenia podczas przetrzymywainia spin_locka.

Debugger do jądra

Brak ,,oficjalnego'' debuggera jądra

I'm afraid that I've seen too many people fix bugs by looking at debugger output, and that almost inevitably leads to fixing the symptoms rather than the underlying problems.

Linus

Co można zdziałać zwykłym debuggerem gdb?

- można uruchomić debugger wobec jądra (tak samo, jak wobec zwykłych procesów)

```
gdb vmlinux /proc/kcore
```

- po uruchomieniu debuggera można korzystać z dowolnych implementowanych w nim poleceń podglądu danych, np.:

```
p zmienna_globalna
```

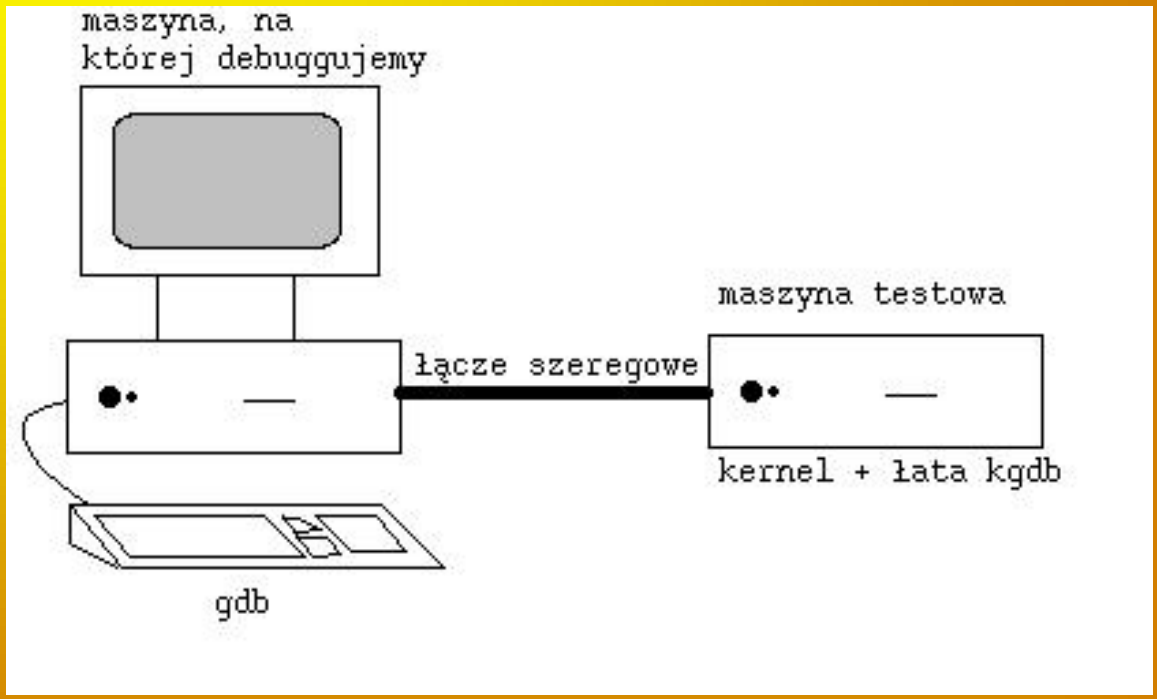
- do deasemblacji funkcji należy użyć polecenia dissassemble:

```
dissassemble moja_funkcja
```

- jeżeli skompilujemy jądro z opcją -g, debugger może wydobyć oczywiście znacznie więcej informacji. Możliwe jest choćby podglądanie zawartości struktur i wyłuskiwanie wskaźników. Oczywiście jednak opcja -g powoduje znaczny rozrost jądra, więc absolutnie nie należy jej stosować przy kompilacji do celów innych niż debuggowanie.
- **ALE** na tym wyczerpuje się lista możliwości gdb jeżeli chodzi o jądro. Nie da się za jego pomocą modyfikować danych działającego jądra. Nie można wykonać pojedyńczej instrukcji kodu, ani też ustawić punktów wstrzymania wykonania. To poważne wady.

kgdb

• idea jest taka, że na jednej maszynie odpalamy testowane jądro z nałożoną łatą kgdb i odpluskwiamy je za pomocą gdb z drugiej maszyny połączonej z pierwszą łączem szeregowym (tak zwanym kablem null-modem),



• potrzebna jest druga maszyna, żeby było gdzie postawić gdb do obsługi kodu źródłowego i informacji dla debuggera pozostawionych przez gcc.

• http://kgdb.linsyssoft.com/

kgdb (To jak się za to wszystko wziąć?)

1. łączymy maszyny poprzez null-modem (szeregowy kabel 26-152B (Female DB9 - Female DB9) i adapter Null Modem: 26-264B (Female DB9 - Male DB9))
2. Nakładamy łatę kgdb na jądro, które będziemy testować.
3. Przy konfigurowaniu włączyć opcję o zdalnym debuggowaniu z grupy kernel hacking.
4. Kompilujemy jądro i dodajemy je do gruba (dodajemy też specjalne opcje w linii poleceń).

```
kgdbwait kgdb8250=< port number>,< port speed>
```

5. Kopiujemy vmlinux na maszynę, z której będziemy debuggować
6. Ustawiamy prędkość połączenia na komputerze, z którego będziemy debuggować:

```
stty ispeed < port speed> ospeed <port speed> < /dev/ttyS<port number>
```

kgdb (cd.)

7. Uruchamiamy gdb. Jeżeli chcemy debuggować też moduły, musimy zaopatrzyć się w specjalny gdb (tu nazwany gdbmod), można go ściągnąc ze strony kgdb. Jeżeli nie chcemy, można skorzystać z normalnego gdb:

```
gdbmod vmlinux
GNU gdb 20000204 Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it
under certain conditions. Type "show copying" to see the
conditions. There is absolutely no warranty for GDB. Type "show
warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb)
```

8. Na maszynie testowej wybieramy z boot loadera jądro z kgdb. Dostaniemy napis:

```
Waiting for connection from remote gdb...
```

9. Na maszynie, na której debuggujemy łączymy się za pomocą komendy

```
(gdb) target remote /dev/ttyS1
Remote debugging using /dev/ttyS1 breakpoint () at
gdbstub.c:1153 1153 }
(gdb)
```

I JUŻ!

Można też debuggować moduły – szczegóły w materiałach.

Słów kilka o kdb

• Zmieniamy podejście do problemu. Będziemy debuggować lokalnie a nie zdalnie. Nie potrzebujemy dwóch maszyn, ale za to nasze możliwości będą bardziej ograniczone.

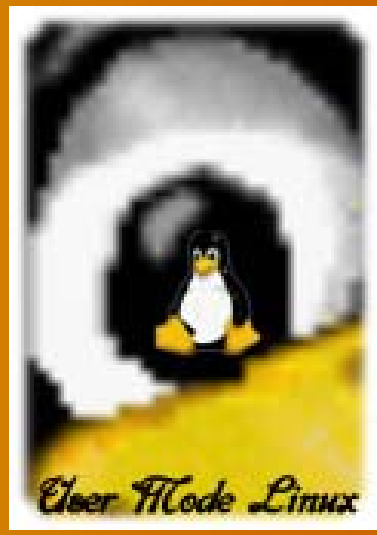
• kdb (Build-in Kernel Debugger) to łata na jądro, która pozawala badać pamięć i struktury system podczas gdy system chodzi. Zbiór komend umożliwia mięedzy innymi:

- wykonywanie pojedyńczych kroków (procesorowych),
- zatrzymanie na wykonaniu danej instrukcji,
- zatrzymanie na dostępie (lub zmianie) w zadanym miejscu pamięci,
- modyfikacje zmiennych
- deasemblację instrukcji.

• Ale nie umożliwia np. debuggowania na poziomie kodu.

User Mode Linux

i jego użycie do debuggowania jądra



Ogólnie o UMLu:

• Najkrócej można powiedzieć, że UML to tak zmodyfikowane jądro Linuxa, że można je uruchomiś jako zwykły proces w przestrzeni użytkownika. Powstaje zamknięte środowisko, które "emuluje" Linuksa.

• Procesy uruchomione pod kontrolą emulatora nie mogą oczywiście zdawać sobie sprawy, że są "oszukiwane" tzn. z punktu widzenia procesu nie da się rozróżnić że komunikacja przez wywołania systemowe odbywa się z mniejszym systemem.

• Podstawowe zastosowanie UML to oczywiście testowanie samego jądra, modułów, aplikacji. Dokonywanie zmian w prawdziwym środowisku zawsze wiąże się z niebezpieczeństwem destabilizacji pracy systemu, zawieszenia go, utraty danych itp. W przypadku UML-a nie ma tego niebezpieczeństwa.

• http://user-mode-linux.sourceforge.net

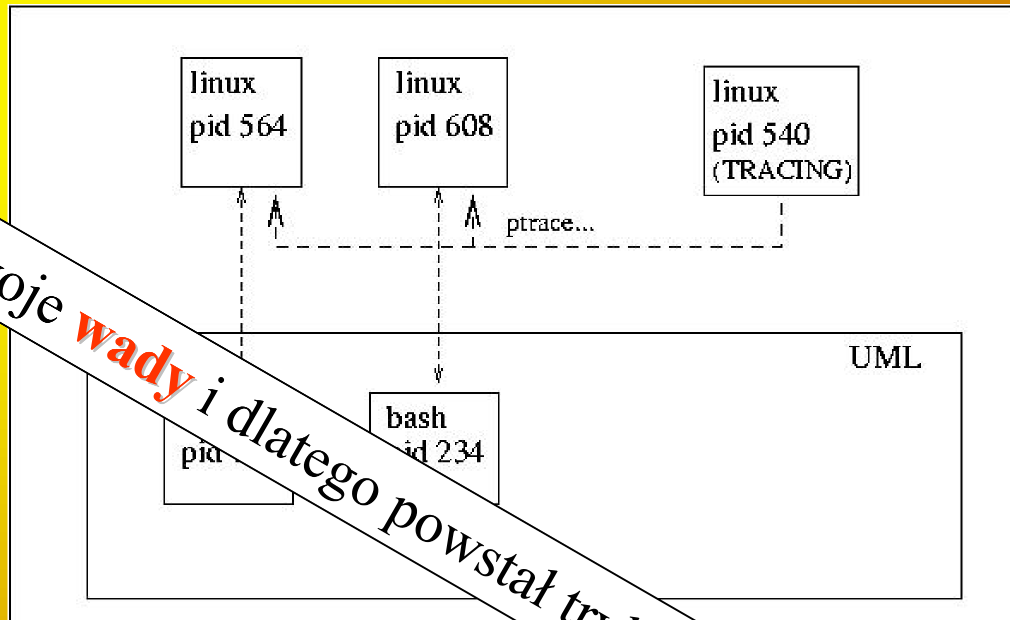
User Mode Linux (cd.)

- „zumlować” możemy dowolne jądro
- w wyniku instalacji powstaje plik wykonywalny linux i plik będący systemem plików
- szczegóły instalacji opisane są w naszych materiałach

Dygresja: jak to działa?

- UML może działać w dwóch trybach: trybie tt (od ang. tracing thread) i w trybie skas (od. ang. seperate kernel adress space).
- Tryb tt jest starszym trybem, ale to on, mimo pewnych wad będzie nas interesował, bo wtedy debuggowanie jest możliwe

User Mode Linux (tryb tt)



Tryb tt ma swoje **wady** i dlatego powstał tryb skas – więcej informacji w materiach

Każdemu procesowi, który "uruchamiamy" pod UML odpowiada rzeczywisty proces uruchomiony pod jądrem hosta, jednak kontrolowany całkowicie przez UML. Aby się o tym przekonać można np. wykonać polecenie ps. W wyniku otrzymamy mnóstwo procesów o nazwie linux. Ponadto przy starcie UML tworzy specjalny proces - wątek śledzący, którego zadaniem jest wirtualizacja wywołań systemowych innych procesów. W tym celu korzysta on oczywiście z funkcji ptrace.

User Mode Linux (debuggowanie)

- Widać, że aż się prosi, aby użyć UMLa do debuggowania jądra.

- Aby odpluswiać jądro UML w gdb należy przede wszystkim pamiętać o włączeniu opcji CONFIG_DEBUGSYM i CONFIG_PT_PROXY podczas konfiguracji. Spowoduje to skompilowanie jąda z opcją -g i włączenie ptrace proxy.

- Gdy mamy już takie jądro możdemy je uruchomić pod kontrolą gdb w następujący sposób:

```
Linux debug
```

- Jądro njpierw wyśle kilka komend i zatrzyna się na start_kernel, co wygląda mniej więcej tak:

```
GNU gdb 4.17.0.11 with Linux support
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or
distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.
Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
gdb) att 1
Attaching to program `/home/dike/linux/2.3.26/um/linux', Pid 1
0x1009f791 in __kill ()
(gdb) b start_kernel
Breakpoint 1 at 0x100ddf83: file init/main.c, line 515.
(gdb) c
Continuing.
Breakpoint 1, start_kernel () at init/main.c:515 515 printk(linux_banner);
(gdb)
```

I JUŻ!

User Mode Linux
(debuggowanie modułów)

- jest to dość skomplikowane, więc jakiś dobry człowiek napisał skrypt umlgdb do ściągnięcia ze strony umla.
- w tym skrypcie trzeba wstawić swój moduł do listy ścieżek
- należy go po prostu uruchomić, a on mówi, co zrobić:

```
******** GDB pid is 21903 ********
Start UML as: ./linux debug gdb-pid=21903

GNU gdb 5.0rh-5 Red Hat Linux 7.1 Copyright 2001
Free Software Foundation, Inc. GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) b sys_init_module
Breakpoint 1 at 0xa0011923: file module.c, line 349.
(gdb) att 1
Attaching to program: /home/jdike/linux/2.4/um/./linux, process 1 0xa00f4221 in __kill ()
(UML gdb) c
Continuing.
(UML gdb)
```

I JUŻ!

- gdy zrobimy insmoda, zostanie to dostrzeżone i zobaczymy coś w tym stylu:

```
*** Module hostfs loaded ***
Breakpoint 1, sys_init_module (name_user=0x805abb0 "hostfs", mod_user=0x8070e00) at module.c:349
349 char *name, *n_name, *name_tmp = NULL;
(UML gdb) finish
Run till exit from #0 sys_init_module (name_user=0x805abb0 "hostfs", mod_user=0x8070e00) at module.c:349
0xa00e2e23 in execute_syscall (r=0xa8140284) at syscall_kern.c:411
```

Symbole

- Wygodnie jest posługiwać się przy odpluskwianiu nazwami funkcji i zmiennych z kodu źródłowego
- Punkty przerwań również łatwiej jest ustawiać według numeru linii, a nie adresu w skompilowanym kodzie.

Symbole

- Trzeba więc dołączyć do programu pewne informacje o źródłach
- Sam kod nie wystarczy – trzeba jeszcze powiązać go z skompilowanymi instrukcjami
- Co okazuje się być nietrywialne

Symbole

- Debuger musi być przystosowany do uruchamiania programów pisanych w różnych językach
- Kompilatory generują kod na różnorakie sposoby
  - Czasem mogą np. wyrzucić nie używaną zmienną, rozwinąć pętlę etc.
  - W różny sposób przydzielają pamięć zmiennym

Symbole

- Potrzebny jest format informacji o symbolach umożliwiający
  - Kojarzenie instrukcji skompilowanych z liniami plików źródłowych
  - Wygodne badanie struktur danych języka źródłowego
    - Trzeba więc umieć kodować typy danych
  - Wyliczenie adresu zmiennej/funkcji o podanej nazwie
    - Trzeba pamiętać o regułach widzialności zmiennych w języku źródłowym
  - …

Formaty

- Stosowane w Linux'ie
  - Stabs
  - DWARF
- Stosowane w Windows
  - COFF
  - C7 (CodeView)
  - PDB
  - OMAP

Stabs

- Format stworzony przez Petera Kesslera na użytek pascalowego debugera *pdx*
- Informacje mogą być dołączone do róznych formatów plików wykonywalnych (a.out, COFF, ELF, ...)
- Dawniej domyślny format gcc

Stabs

- gcc umieszcza informacje w wynikowym pliku assemblerowym w postaci dyrektyw
- W dalszych fazach są one odpowiednio kodowane i umieszczane w wyjściowym programie

Stabs - dyrektywy

- **.stabs "*string*",*type*,*other*,*desc*,*value***
  - **W pozostałych przypadkach pole *string* uznajemy za puste**
- **.stabn *type*,*other*,*desc*,*value***
- **.stabd *type*,*other*,*desc***
  - **Pole *value* ma w tym wypadku wartość równą obecnej lokalizacji w pliku**

Stabs – struktura programu

- **Nazwa pliku jest zapisana w rekordzie typu N_SO, którego pole *string* jest rzeczoną nazwą, zaś *value* adresem kodu odpowiadającego plikowi**
  - **Czasem jest też osobny rekord N_SO z nazwą katalogu**

```
.stabs "D:/so/c/",100,0,0,Ltext0
.stabs "prog.c",100,0,0,Ltext0
# N_SO jest równe 100
```

Stabs – struktura programu

- **Numery linii są przechowywane w rekordach N_SLINE,**
  - **Pole *desc* zawiera numer linii**
  - **Pole *value* zawiera adres w kodzie odpowiadający początkowi tej linii**
  - **Niektóre linie generują nieciągły kod – wtedy jest wiele rekordów N_SLINE o tej samej linii**
- **Do oznaczania kodu z dołączonych plików używa się rekordów N_SOL**

Stabs – struktura programu

- Procedurom odpowiadają rekordy N_FUN
- Część informacji można zdobyć ze zwykłej tablicy symboli pliku wykonywalnego
  - Co jednak bywa nietrywialne i zwykle jest nieefektywne
- Pole *string* zawiera nazwę i typ funkcji oraz nazwy procedur, w których jest ona zagnieżdżona
- Numer linii można otrzymać z następnego rekordu N_SO

Przykład (F oznacza funkcję globalną (extern), (0,1) – typ, 36 = N_FUN

stabs "main:F(0,1)",36,0,3,_main

_main to etykieta odpowiadająca początkowi funkcji

Stabs – struktura programu

- Strukturę blokową programu odzwierciedlają rekordy N_LBRAC i N_RBRAC
- Blok z treścią funkcji jest opisany po odpowiednim rekordzie N_FUN
- Albo i nie

Stabs - zmienne

- Zmienne globalne – rekordy N_GSYM
- Zmienne lokalne
  - N_LSYM dla zmiennych na stosie
  - N_RSYM dla zmiennych w rejestrach

  Przykłady
  - .stabs "x:1",128,0,0,-12
  - .stabs "x:r(0,1)",64,0,4,0

  N_LSYM = 128, x = nazwa, 1 i (0,1) = typ, -12 = offset w ramce, 4 = numer rejestru, 64 = N_RSYM

Stabs - typy

- Typom odpowiadają numery
- Niekiedy także numery plików, wtedy typ wygląda tak: (nr_pliku, nr_typu)
- Definicja typu ma postać

  .stabs „nazwa:t<nr>=[deskryptor] inne; dane” 128, 0, 0, 0

  Przykład

  .stabs "long int:t(0,3)=r(0,3);-2147483648;2147483647;",128,0,0,0

  Deskryptor ‘r’ oznacza typ przedziałowy, 128 to N_LSYM

Stabs – C++

- Format stabs został rozszerzony tak, by obsługiwać C++ i podobne języki
- Umożliwia więc debugowanie kodu używającego klas, szablonów i innych takich
- Dla ciekawskich: g++ –g –S prog.cpp
- Miłej lektury

DWARF

- **D**ebugging **W**ith **A**ttributed **R**ecord **F**ormats
- Format powstał na użytek sdb
- Przystosowany jest do obsługi różnych języków i procesorów
- Obecną (od 4 stycznia 2006) wersją jest DWARF3
- Zmiany w stosunku do wersji 2 to m.in.
  - Obsługa informacji większej niż 4GB
  - Wsparcie dla dodatkowych języków (Java)
  - Poprawiona obsługa optymalizacji oraz eliminacja zduplikowanych informacji

DWARF

- Informacja w formacie DWARF ma postać wpisów, składających się z typu oraz zestawu atrybutów
- Wpisy umieszczone są (od DWARF2) w sekcji *.debug_info* pliku obiektowego
- W danym wpisie jest nie więcej niż jeden atrybut o danej nazwie. Wartość atrybutu należy do jednej z predefiniowanych klas

DWARF

- Wpisy reprezentowane są w formie drzewa, co pozwala m.in. w naturalny sposób reprezentować blokową strukturę programu.
- Oczywiście istnieją także inne związki między wpisami.

DWARF - Wyrażenia

- Często konieczne jest wyrażenie jakiejś wartości w formie wyrażenia, odnoszącego się np. do obecnej wartości pewnych rejestrów
- DWARF zapisuje takie wyrażenia jako program prostej maszyny stosowej.
- Wyrażenia są przydatne do opisywania położenia obiektów w czasie wykonania programu. W DWARF2 był to jedyny rodzaj wyrażeń

DWARF - Wyrażenia

- **Poniższe przykłady są wyrażeniami opisującymi lokacje. Jest to jedyny rodzaj wryażeń, w którym można używać operacji nazywających rejestry**
  - `DW_OP_REGX 42`
    - **Obiekt jest w rejestrze 42**
  - `DW_OP_BREG11 42`
    - **Obiekt jest 42 bajty od miejsca wskazywanego przez rejestr 11**
  - `DW_OP_bregx 54 32 DW_OP_deref`
    - **Obiekt jest wskazywany przez słowo położone 32 bajty od adresu z rejestru 54**

DWARF – listy lokacji

- Służą do opisu położenia obiektów, które przemieszczają się w trakcie działania programu (np. ze względu na optymalizację).
- Umieszczone są w sekcji `.debug_loc`
- Jak sama nazwa wskazuje, są to listy zawierające lokacje (wyrażenia) oraz adresy opisujące częsci programu, w których dana lokacja jest ważna

DWARF – Numery linii

- Informacja o numerach linii pliku źródłowego nie występuje w zwykłych wpisach, lecz w sekcji .debug_line
- Ma postać macierzy przyporządkowującej instrukcjom kodu maszynowego odpowiednie pliki, numery linii, kolumn, …
- Macierz ta zakodowana jest w formie programu prostej maszyny o 10 rejestrach

DWARF – rozwijanie wywołań

- Debugery często muszą badać i modyfikować aktywacje procedur położone w głębi stosu wywołań
- Niestety, kompilatory generują kod obsługujący ramki w rozmaitych miejscach, formach i stopniach optymalizacji
- Dlatego DWARF zawiera dość złożony schemat opisu zawartości i położenia ramek. Jego elementem jest zestaw instrukcji służących do wyliczania położenia różnych elementów ramki.
- Informacje te umieszczane są w sekcji `.debug_frame`
- Niektóre elementy opisu ramek są specyficzne dla konkretnych procesorów/kompilatorow.

DWARF – inne cechy

- Jedną z głównych cech formatu jest rozszerzalność
- Przystosowany do
  - obsługi konstrukcji z języków typu C++ (klasy, szablony, …)
  - Operowania na optymalizowanym kodzie (przemieszczalne obiekty, nieciągła widzialność zmiennych)
  - Korzystania z makrodefinicji w plikach źródłowych

Opcje kompilacji

- Cała omawiana wcześniej informacja o symbolach musi być oczywiście wygenerowana przez kompilator
- Który trzeba o to ładnie poprosić

gcc

- Do dołączania informacji o symbolach służy opcja –g
- Po niej może występować nazwa formatu
  - stabs, dwarf-2, coff, xcoff, vms
  - Znak ‘+’ po nazwie formatu oznacza dołączenie informacji specyficznej dla gdb
  - -ggdb oznacza użycie formatu o największej sile wyrazu, z informacjami dla gdb

gcc

- Po nazwie formatu może występować poziom
  1) Minimalna informacja, brak danych o numerach linii i zmiennych lokalnych
  2) Poziom domyślny
  3) Dodane informacje o makrach
- W przypadku dwarf-2 poziomu nie można dokleić po nazwie formatu, trzeba dodać opcję ***-gpoziom***

gcc

- Opcja –pg powoduje powstanie programu generującego informacje dla gprof
- -p pozwala tworzyć informacje dla prof.
- -ftest-coverage powoduje powstanie podcas działania programu informacji dla gcov (umożliwiającej wykrywanie martwego kodu)
- -fprofile-arcs sprawia, że wynikowy program gromadzi informacje o liczbie przejść każdym łukiem.
  - Użyteczne do testowania pokrycia z –ftest-coverage
  - Oraz optymalizacji (z –fbranch-probabilities)

MSVC

- Za informacje dla odpluskwiaczy odpowiada głównie opcja /Z
  - /Zi powoduje wygenerowanie tejże informacji
  - /Z7 oznacza wykorzystanie starszego formatu C7 (CodeView)
  - /Zd dołączy tylko informacje o numerach linii

MSVC

- Aby przygotować program do profilowania należy skompilować go z opcją /link profile oraz odpowiednimi informacjami dla debugerów
- MSVC obsługuje profilowanie tylko w wersjach Professional i Enterprise


KONIEC