Troubleshooting

From OSDev Wiki

Contents

Providing a basic debugging environment

Exception handlers

The first thing ever to do is to implement a reliable 'exception handler' which will tell you what went wrong. Under an emulator like Bochs, the absence of such a handler will lead to a '3rd Exception without resolution' panic message (a.k.a Triple Fault), if the emulator is configured to do so... On bare hardware, it will simply reset your computer with a laconic 'bip'.

Every time the CPU is unable to call some exception handler, it tries to execute the Double Fault exception handler. If it fails to call it either, a Triple Fault occurs. Also keep in mind that exceptions cannot be masked, so either your code is perfect or you need exception handlers. Also keep in mind that when you run applications ***their*** code must be perfect without exception handlers, so it's a good idea to get them quite quickly.

It's quite convenient to have the exception handler showing what kind of exception occurred before anything 'hazardous' is attempted. Displaying, for instance, the (hexadecimal) exception number in a corner of the screen, can save you hours of debugging. :)

```
exc_0d_handler:
    push gs
    mov gs,ZEROBASED_DATA_SELECTOR
    mov word [gs:0xb8000],'D '
    ;; D in the top-left corner means we're handling
    ;;  a GPF exception right ATM.
    ;; your 'normal' handler comes here
    pushad
    push ds
    push es
    mov ax,KERNEL_DATA_SELECTOR
    mov ds,ax
    mov es,ax

    call gpfExcHandler

    pop es
    pop ds
    popad

    mov dword [gs:0xb8000],'  D-'
    ;; the 'D' moved one character to the right, letting
    ;; us know that the exception has been handled properly
    ;; and that normal operations continues.
    pop gs
    iret
```

Once you have implemented such a technique, it may be wise to test it, deliberately issuing 'faulty' instructions to see if the correct code is displayed. Having the 'double fault' exception (08) displayed somewhere else on the screen may also be a smart move.

What to do if characters cannot be displayed

Such things occurs for instance when your GDT or paging tables has been badly configured (e.g. 0xb8000 no longer refers to the video memory). Fortunately enough, the video memory is not your sole communication technique with your kernel:

- you may use the keyboard LEDs to report some events (for instance enabling the 'scroll lock' LED when you're a handler and disabling it when you're out).
- you may use the internal PC Speaker to make Morse-code-like signals reporting early errors. (May be disgraceful if coding late, though.)
- you may use VGA registers to change the background color or the overscan (screen border) color to report the current 'state' of your kernel. (E.g. black = normal operations, yellow = processing interrupt, red = crash condition occurred (just before cli:hlt), blue = processing an exception, etc...)

Refer to VGA Resources to see how you can modify colors. The resources page should have all the documentation for LED flashing and speaker beeping.

- you may output bytes via the serial port. Most emulators allow you to redirect these characters into a file and unlike the screen the number of characters is not limited. A driver for the serial port is also very easy to implement.

Avoiding exception loops

So we know when exceptions occur and which exception occurred. That's better but still not especially useful. Your exception handler is likely to become something complex as your kernel will evolve, and you'll discover that exceptions mainly occur ... in exception handlers.

In order to avoid recursive exceptions to occur endlessly, you can easily maintain a 'nested exceptions counter' that will be incremented every time you enter an exception handler and decremented just before you leave that handler. If the counter is above a certain threshold of a few units (3 should give interesting enough results), the kernel will abort trying to solve the exception and enter a 'panic' mode (red background, flashing LED, whatever).

```
int nestexc = 0;

// called by the stub
void gpfExcHandler(void) {
   if (nestexc > MAX_NESTED_EXCEPTIONS) panic();
   nestexc++;

   if (!fix_the_error()) {
     write_an_error_message();
   }
   nestexc--;
   return;
}
```

You need to know, of course, that some exceptions are not 'resumable'. If your kernel issued a division by zero, trying to return to the 'div' instruction will only trigger the exception one more time (yeah! altogether, now :). Such loops cannot be solved by the 'nestexc' counter

Showing the stack content

Much of your program's state (function arguments, return address, local variables) is stored on the stack, especially when using C/C++ code. A complete debugger (like GDB) will inspect the debugging info to give names to the stack content, provide a list of calls, etc. This is a bit complex to do ourselves, but if your kernel can simply *show* the content of the stack and if you know *where* in the code the process halted, you can already fix quite a lot of bugs by doing the job of the debugger yourself, guessing which stack location holds which variable, where the return addresses are, etc.

The stack content is still in memory. The EBP value of the erroring process is still in memory, and points to the start of the stack frame for the current function. Everything from this address and up was the current stack. Now, you can use the value in ebp as the source. Just use the following call:

```
stack_dump:
  push ebp
  mov ebp, esp
  call dump_hex
  pop ebp
  ret ; note that this is not going to work, but it should be here for co
```

and use `void dump_hex(char *stack)`.

Locating the Faulty instruction

In most cases, when your exception handler is called, the address of the faulty instruction is somewhere on the stack. The first step here is to print out the address of this instruction.

Once this is done, you (as a human) can inspect the *linker map* and find out in which object file the problem was. You can request a map with `ld` *<usual options>* `-Map` *<filename.map>*.

```
.text           0x0000000000005330      0x556 bin/init.o
                0x0000000000005370              kinit_dsp_buffer
                0x0000000000005380              kinit_glocal_tag
                0x0000000000005400              kinit
                0x0000000000005830              kreset
 *fill*         0x0000000000005886        0xa 00
 .text          0x0000000000005890      0x66d bin/kalloc.o
                0x0000000000005ac0              kfree
                0x0000000000005b70              kmRegister
                0x00000000000059b0              kealloc
                0x0000000000005e60              kmFindP
                0x0000000000005dd0              kmFindA
                0x0000000000005d10              kmSetFull
                0x0000000000005890              kalloc
```

Is an example of what a map can look like. If the error address was 0x554f, we can tell from this that the error is somewhere in init.o, and most likely in `kinit()` function (it may still be in some other function if there are some *static* functions in the source file). All we know is that the error occurred at offset +21f in the file.

Now, we can use `objdump -drS bin/init.o` to get a look at the disassembled output. Note that this step will work properly only if you had enabled debug information in those separated `.o` files...

```
#ifdef __DEBUG__
  kprint("kernel in debug-mode(%x) press [SHIFT+SPACE] to bypass anykey()
 216:   83 c4 f8                add    $0xfffffff8,%esp
 219:   a1 00 00 00 00          mov    0x0,%eax
                        21a: R_386_32   DbMsk
 21e:   50                      push   %eax
 21f:   68 a0 01 00 00          push   $0x1a0
                        220: R_386_32   .rodata
 224:   e8 fc ff ff ff          call   225 <kinit+0x155>
                        225: R_386_PC32 kprint
         DbMsk);
#endif
```

Of course, as I picked up a random address, there's nothing wrong to see at +21f, but I guess you got my point. :)

Locating the offending line of source code

Once you have found the address of the faulty instruction in the previous step, you can identify the corresponding line of source code by running

```
addr2line -e <your_kernel.elf> <address of faulty instruction>
```

Enhanced debugging techniques

Stack tracing

By analyzing the default way to create a stack frame, you can rip off a stack frame at a time, resulting in the call sequence that leads to the fault. For a single bonus point, also extract the arguments and dump them as well. For multiple bonus points, use C++ name mangling, and export the arguments in readable form in the correct type.

Each time a function is called it gets the following head/tail: (GCC 3.3.2)

```
push ebp
mov ebp, esp
...
leave
ret
```

On the place of the ... the rest of the code is filled in. Now, if you analyze the stack output, it looks something like:

```
0000FFC0 0000FFD0 -> this is the result of a push EBP (which pushes the esp at the start of the previous function)
0000FFC4 001023A5 -> this was the old EIP, which can be looked up in the function table (map file)
0000FFC8 01234567 -> this is an argument
0000FFCC 89ABCDEF -> this is another argument
0000FFD0 0000FFF0 -> this is again another EBP
0000FFD4 00105BC3 -> this is again an EIP
0000FFD8 001023A5 -> this is an argument (could be a function pointer)
0000FFDC 01234567 -> this is another argument
0000FFE0 89ABCDEF -> this is again an argument
0000FFE4 000B8000 -> this is an argument, but not to this function
0000FFE8 FFC00000 -> this is again an argument to a different function
0000FFEC 00010000 -> this is again another argument, but again not to this function.
0000FFF0 0000FFFC -> this is the previous EBP again (note this is in this case the top)
0000FFF4 0010002C -> this is an old EIP
0000FFF8 00000001 -> this is an argument
0000FFFC 00000000 -> this is the EBP at the start of the first function, not necessarily valid!
```

Now, you can traverse along the path of execution. The content of EBP is the old value of EBP, that is, the one of the last stack frame. The value above that is the old instruction pointer (which points inside the current function), and the values above that, up to but not including the value pointed to by the old EBP, are the arguments. Note that the arguments don't have to belong to this function, GCC occasionally saves an add to esp by not popping the values. By then pretending the old EBP is the current EBP, you can unwind another call. Do this until you are fed up by it, you have enough output or the stack ends. If the last one, watch out for not generating a double fault.

If you use C++ name mangling, the arguments are encoded in the function name. If you can read that, you can decode what the value on the stack must be, so you can actually present it to the user in the form of a normal function call with legible arguments and everything. This is the 'crème de la crème' of stack dumping methods, so most aren't expected to do this.

While I program my kernel in C, I actually thought of writing a script that would parse the header files for function declarations, extract the debugging symbols from the compiled kernel image using `objdump`, and write a system map which would provide the types. Forgot it after falling in love with Bochs' debugger though. Similarly, typemaps for structured types could be created, which would allow the same kind of browsing that GDB or Visual Studio give you. THIS would be the crème de la crème.

Debugging techniques

If your function `x()` wreaks havoc only after 1000 calls it may not suffice to put a `panic()` statement inside the functions to see where the functions breaks. You may want to know which call is malignant. To do this, one might use a global or static var to count calls and panic() after an amount to see if it managed to crash. If not, you try twice that amount; if it does crash, you try bisection to find the amount.

```
void scheduler_choose_task() {
    static uint32_t Z=0;
    Z++;
    uint32_t N = 1000;
    if (Z > N) panic();  //find the largest integer N for which it crashe
    if (in_critical_section()) return;
    ...
}
```

...and then check how far does it go:

```
Z++;
uint32_t N = 1000;
                                       //we get here,
if (in_critical_section()) return;
if (Z > N) panic();                    //do we get here to panic() before
```

However as complexity rises or multithreading is involved, it is less probable that a crash would be consistently occurring at the same point, after the same amount of calls every time. Then it would not be possible to find the number of the call to `scheduler_choose_talk()` that crashes it (because that number changes). Debugging needs some imagination; what if you knew, by tracing the program flow with `print(__LINE__)` that `scheduler_choose_task()` crashes only when a call to `fun1()` is in progress? You might use a global var `uint32_t dbg` or an array (`uint32_t dbg[20]`) of various `dbg` vars (which are used only in debugging code which is cleaned after the programmer ceases to debug) in a manner such as:

```
void fun1() {
    dbg[3] = 1;
    ...
    if (x()) { dbg[3]=0; return; }
    ...
    dbg[3] = 0;
}
```

...and:

```
void scheduler_choose_task() {
    //  if (dbg[3]==1) panic();         //check here.. a panic saves the
    if (in_critical_section()) return;
    if (dbg[3]==1) panic();             //check here.. it crashes
}
```

(Or mix it with a call count, `Z++; if (Z>5 && dbg[3] == 1) panic().`)

Using the `__LINE__` aids tracing the program flow:

```
print(__LINE__);
```

See Uses for debugging for more info.

External assistance

So far, we assumed that the kernel was containing all the information required for debugging (like symbols names, etc). In production-stage, however, this information, as well as most of the 'debugging prints' have usually been stripped out of the final binary object.

Still, one could imagine a kernel that would be equipped with serial-line communication code and connected to another computer that would have all the 'removed' information (like the symbols map, or the debugging-info featured intermediate binaries).

In case of a panic, the kernel could for instance send the value of `eip` over the serial line and expect the helper PC to reply with the function name and line number corresponding to that address (or dump it on the helper PC's screen, that's a matter of choice :)).

Debugging interface

Now we have plenty of information about what was wrong... can we ask for more ? what do Mobius' debugging shell and Clicker's information panels tell us ...

Does anybody else know of an OS that allows the hacker to interactively probe the system state when crashes occur ?

> *Yes. Guess. Correct: AmigaOS. ;-) It offered a mode that allowed debugging over serial line even after the system went into Guru Meditation. That was possible because AmigaOS enjoyed a 256 / 512 kByte ROM image that could not get corrupted.* - MartinBaute

Unix systems traditionally write their state to /dev/core for offline guru meditation

See Also

Articles

- How Do I Use A Debugger With My OS