

Debugging your OS

By Daniel Stødle
daniels@stud.cs.uit.no

© 2003 all rights reserved.

One of the most common things you will experience during the development of your OS, is having your OS crash horribly. This text will give you information on some handy tools, as well as provide some general debugging techniques.

First, when testing your OS, you should primarily be using vmware, as its speed is far superior to that of bochs. You should switch to bochs whenever you run into showstopper bugs (ie, bugs that end up killing vmware). Note that both vmware and bochs support running your OS directly from the image-file generated when the OS is built. For bochs, the name of the image-file is specified in the .bochsrc file, located in your home directory. In the FLOPPYA section, add this:

```
floppya: 1_44=image, status=inserted
```

You should comment out any other lines starting with “floppya”. For vmware, the configuration is similar; select Settings->Configuration Editor and then expand the floppy-section. Select the first floppy-drive, and set its type to “file.” Finally, enter the path to the image-file in the text field, and ensure that the “Start connected” button is checked. (This will also work for the home exam, where your OS writes to the disk. Yay!)

Now, you have run into a showstopper bug, and you have no clue what made your OS to crash. The first step towards tracking down the bug, is to figure out where it crashed. This is quite simple, as all you need to do is run the OS under the bochs emulator. Bochs will exit with an error message when your OS crashes, writing a brief log file (bochsrc.txt) to the working directory, which in addition to saving the contents of the registers at the time of the crash, also displays the current value of EIP when the crash occurred. A short textual description (triple fault, pop beyond stack limits, etc) of why things crashed is also given.

So, with the EIP at hand, it's time to figure out where the crash occurred in our code. Using the utility objdump, figuring that out is easy. Execute the following command (your working directory needs to be the one where your kernel-file is located): `objdump -d kernel | less` You should see a disassembled version of your kernel. Now, scroll until you find the value of EIP in your kernel (it's along the left column), and look at the surrounding disassembly. At the very least, you should be able to see what function you are in; additionally, you may be able to see if the instructions “look right” compared to what you would expect the function to

do. (Note that if the EIP given by bochs is less than 0x1000, it's probably invalid. This is due to the fact that the kernel's entry point is 0x1000. Know your memory layout!)

As an example, let's say we want to save the context of a process, and we want to do this inline in a C function. The function might look something like this:

```
void    save_context(void) {
    asm volatile ("
        # Save all registers
        pusha
        # Save stack pointer to PCB
        movl    current_running, %eax
        movl    %esp, KERNEL_STACK(%eax)
    ");

    print_str(0,0, "Context saved, scheduling next..");
    scheduler();
}
```

(Note that gcc will issue a warning for the above code, stating that "multi-line string literals are deprecated." It's safe to ignore this warning.)

Let's assume that upon dispatching a process, the kernel expects to just be able to restore the stack, then restore the registers (by executing the popa instruction), and finally return to the function that called save_context by issuing a ret-instruction. Unfortunately, the computer crashes when it attempts to return.

For our example, let's say EIP points to the ret-instruction in our dispatch() routine when the computer crashes. We immediately see that there's nothing wrong with our dispatch, so consequently something must have happened to the return address (which is pushed automatically when we 'call' a procedure). Thus, we take a closer look at our save_context function. Now, our code above seems perfectly correct – we only push some registers, then save the value of the stack pointer. Nothing else happens, so the return address should be just before the saved registers on the stack:

```
0: return address
-4: eax
-8: ebx
etc.
```

In this case, the code *is* in fact correct as well, our problem is the compiler messing up our assumption that the stack is left untouched immediately after entering the save_context function. Objdump allows us to discover this. Looking at the output produced it produces, we see the following disassembly of our save_context function:

```

01000814 <save_context>:
1000814:      83 ec 10          sub     $0x10,%esp
1000817:      60              pusha
1000818:      a1 00 00 00 00    mov     0x0,%eax
100081d:      89 60 0c          mov     %esp,0xc(%eax)
1000820:      68 80 08 00 01    push   $0x1000880
1000825:      6a 00              push   $0x0
1000827:      6a 00              push   $0x0
1000829:      e8 8e fa ff ff    call   10002bc <print_str>
100082e:      83 c4 1c          add     $0x1c,%esp
1000831:      e9 9e fe ff ff    jmp     10006d4 <scheduler>

```

The column at the right indicates the value EIP would have when executing the instruction on the corresponding line. Notice how the compiler sneakily has introduced a sub-instruction before our own assembly code? Using objdump and by knowing where the OS crashed, we have been able to at least determine one potential error source.

(The solution to the above problem, as an aside, is to put the stack-sensitive code in its own assembly file, as is done in entry.S in our OS projects. This gets rid of any compiler-peculiarities, and allows us to write more readable code, as we no longer need to worry about asm volatile (“..”) all the time. A bad solution to the above problem would be to compensate for the stack modifications done by the compiler by adding the offset created by the compiler back to the stack when resuming a process, as this is prone to create even harder-to-track bugs down the line. The technique also introduces a nasty compiler dependency. The best solution might be forcing the code down the compiler’s throat, beating it until it doesn’t generate the leading “sub” instruction.)

All errors, unfortunately, are not as easily discovered as the one we caught above. Sometimes the value for EIP that we get from bochs is worthless (EIP points to something that isn’t in our code, for instance), or the value is valid, but the reason for the crash is bad data. At any rate, at times like these, all one could ever wish for would be a source-level debugger with some rudimentary data display tools. Unfortunately, no such thing is available for our OSes, so the second-best solution is to code defensively and check all inputs and outputs before proceeding. (Actually, bochs **does** have some debugging tools, but as I haven’t used them myself, I won’t go into more detail about them. Not yet, anyway ;))

The basic technique here is really to return to the stone age, and add calls to delay/ms_delay¹ at potential trouble spots in your code, combined with printing out the values of important variables. In addition, using asserts wherever there are critical variables being accessed, will help you track down errors.

Crashes in your code will often be caused by broken assembly code, and there are a number of gotchas to watch out for when doing assembly programming. For

¹delay is replaced by ms_delay starting with P3.

instance, comparing a memory location with a register, when what you intended to do was compare the register with a constant value, or adding the contents of a memory location rather than a constant:

```
# The following is dead wrong, unless you really want to
# compare %eax with whatever lies at location 0 in memory,
# or add whatever is at mem[4] to %esp:
cmpl    0, %eax
addl    4, %esp
# The right way to do it
cmpl    $0, %eax
addl    $4, %esp
```

The same gotcha applies when you are using named constants:

```
.equ    MY_CONSTANT, 0
# Wrong (unless MY_CONSTANT actually is a memory address)
cmpl    MY_CONSTANT, %eax
# Correct
cmpl    $MY_CONSTANT, %eax
```

Forgetting parentheses when you need to store something at a memory location:

```
my_temp_var:
    .long    0
my_function:
    # Wrong! (Unless you intend to store the address of
    # my_temp_var in %eax)
    movl    %eax, my_temp_var
    # Correct!
    movl    %eax, (my_temp_var)
```

Finally, always take care to know your memory layout. If you know where your processes are located in memory and know where their stacks are, you will be able to use the value of esp stored in the bochsout.txt-file in your debugging efforts as well. Good luck!