

CSIS0230A Principles of Operating Systems (Class A)

Tutorial 6

Signals

Imagine that you are writing a large project, which is to be shipped to the user. You are afraid that the program contains problems. However, the regular core dump is not useful, as the core would be very large. So instead, you just want your program to trap segmentation faults and exits, telling what type of wrong access is done, and telling the memory location that is being accessed.

Write a handler to print out what has happened: some other process might send it a SIGSEGV signal, the program might access some non-existent memory, or the program might write to some read-only region. In your program, differentiate and print a message to indicate these cases. In the latter two cases, print out the memory address being accessed. Exit after the message is printed.

Write also the required code to setup the signal handler. Note that you need to setup an extended sigaction handler for that purpose. Test your code by generating a segmentation fault in your program using various methods (e.g., calling *raise()*), accessing invalid memory, and writing read-only memory.

Hints

- Remember that *si_code* stores information about why a signal is sent. Find the relevant information from the manpage of *sigaction*.
- Make sure to set the appropriate *sa_flags* in the signal action to select using an extended sigaction handler.
- To create an invalid pointer, just cast the corresponding integer to a pointer. E.g., a pointer to memory location 0x10 can be generated by **(char *)0x10**.
- If you finish early, you can try modifying it so that it traps all signal types, and try generating different types of signals without using *raise* or *kill*. Report the signal trapped within the signal handler. (E.g., to generate SIGINT, type Control-C. The program should print 2, the signal number of SIGINT.)

Note: it is possible to find more information about the processor states when the fault occurs, e.g., the address of the faulting instruction. This is done by using the last argument of the signal handler, say *cnt*. If you feel adventurous, write the following in the beginning of your signal handler:

```
ucontext *context = (ucontext *) (cnt)
```

Now run your program in *gdb*, set a break-point at the signal handler (by, e.g., `break my_segv_handler`), and asks *gdb* not to trap the segmentation fault signal (by saying to *gdb* `handle SIGSEGV nostop noprint`). Step the program so that the new line is executed, and say `print/x *context`. Inspect the *uc_mcontext* member. Note how *cr2* is set to the faulting address, and *eip* is exactly the address of the instruction causing the segmentation fault.

This is by definition OS and processor specific, though.