ECE471: Embedded Systems – Homework 3

Linux Assembly and Code Density

Due: Friday, 27 September 2024, 5:00pm EDT

1. **Use your Raspberry Pi for this assignment**
   - Download the code from:
     `https://web.eece.maine.edu/~vweaver/classes/ece471/ece471_hw3_code.tar.gz`
     and copy it to the Raspberry Pi.
   - Uncompress/unpack it with the command `tar -xzvf ece471_hw3_code.tar.gz`
   - Change into the ece471_hw3_code directory `cd ece471_hw3_code`
   - Put all answers to questions into the included text `README` file. This will automatically be bundled up with your submission.

2. **Modify the hello_world assembly program to return the value 42. (2 points total)**
   (a) Note! How you do this depends on if you have a 64-bit or 32-bit version of Linux installed! See the Appendix at the end here if you don't know which one you are running.
   (b) For 32-bit, cd into the "32bit" directory and edit the file `hello_world_arm32.s`
   (c) For 64-bit, cd into the "64bit" directory and edit the file `hello_world_arm64.s`
   (optionally, you can do the 32-bit version on 64-bit instead but you will need to install the cross-compiler, see the Appendix at the end of this homework for more info)
   (d) Be sure you comment the code you are modifying!
   (e) Run `make` to generate an updated version
   (f) To test, run `./hello_world_arm32` (or 64) followed by `echo $?` which will show you the previous program's exit status, which should be 42 if you did everything right.
   (g) Some reminders about Linux GNU assembler (`as`) syntax:
   `.equ IDENTIFIER,value` sets a macro replacement, like
   `#define IDENTIFIER value` would in C
   There are a few different ways to comment your code but I recommend using C++ style comments such as `// this is a comment`
   (h) Brief assembly reminder, to move the number 5 into a register you would do `mov r0,#5`
   (i) Reminder: for 32-bit the Linux the kernel ARM syscall ABI:
   Arguments go in `r0` to `r6`
   System Call Number goes in `r7`
   Use `swi 0x0` to trigger a system call.
   (j) Reminder: for 64-bit the Linux the kernel ARM syscall ABI:
   Arguments go in `x0` to `x6`
   System Call Number goes in `x8`
   Use `svc 0x0` to trigger a system call.

3. **Investigate the code density of** `integer_print` **(3 points total)**

   (a) To avoid confusion with 32-bit vs 64-bit, I've provided pre-compiled versions of the code in the "precompiled" directory. If you're excited for seeing how I generated these you can go into the "integer_print" directory and run `make` but it will only work on a 32-bit system.

   (b) To see the algorithm we use for printing an integer, look at the source code (the same file is used for both): `integer_print.c`. The algorithm used will be described in class.

   (c) Find the size of the `print_integer()` function in the ARM32 executable (see below) and record it in the README

      i. Find the `print_integer()` function. Look at the the `integer_print.disassem` file I generated for you using objdump. If you use the "less" text file viewer you can use the '/' slash character to start a search and then search for `print_integer`. The first location you find is the call from main() to the function, press '/' again to find the actual function.

      ii. (Note, to quit, just type `q`)

      iii. You should find something like the following:

```
0001043c <print_integer>:
   1043c:       e52de004        push    {lr}            @ (str lr, [sp, #-4]!)
   10440:       e30ccccd        movw    ip, #52429      @ 0xcccd
   ...
```

      The first column is the address in memory where this code lives, the next is the raw machine code for the instruction, the next is the decoded assembly language, and the last is the disassembler giving "helpful" hints about what's going on.

      iv. You can see in this first case all the instructions are 32-bit hex values.

      v. Calculate the length of this function and note it in the README. You can calculate length by scrolling down to where the function ends, (there will be a break in the disassembly, in the provided code it's after the return (pop pc) instruction and the start of the .fini section)

      vi. Please report the length in decimal bytes, you can do this by finding a hex calculator and subtracting the end address from the beginning address then convert to decimal.

   (d) Now go back and do the same for the THUMB2 `integer_print.thumb2` executable too. The disassembly is in `integer_print.thumb2.disassem` Record the size of the function in the README.

   (e) Answer in README: Does the machine language generated look different for THUMB2 than ARM32? How?

4. **C vs Assembly code density: (2 points total)**
   Put the answer to these in the README.

   (a) Compare the size of the ARM32 `integer_print` executable and the
   THUMB2 `integer_print.thumb2` executables. (list both sizes)
   You can get filesize with `ls -l` (that's a lowercase L) You will want to run the `strip` command on the executables first (i.e. `strip integer_print`) which strips off debugging info and will make the files smaller.

   (b) The `integer_print_static` file is also generated. This has the C library "statically linked" (included in the executable instead of dynamically loading the system copy of the C library at runtime). How does the size of the static version compare to the others?

(c) I also provide a 64-bit version of the integer print code. What is the size of that file?

(d) Finally, I provide a pure assembly language version of the integer print code: `integer_print_asm`. What is the size of that file?

(e) Given these results, Which language (C vs assembly) might you use in a space constrained embedded system? Why?

(f) Which code to you think is easier to write, the C or assembly one?

5. **Use `gdb` to track down the source of a segfault. (2 points total)**

   (a) Change into the "crash" directory and run `make`

   (b) This will build a program called `crash`

   (c) Run that program. It should crash with a `Segmentation fault` error.

   (d) Use the `gdb` debugger to find the source of the error.

   (e) Run `gdb ./crash`

   (f) When it comes up to a prompt, type `run` and press enter to run it until it crashes.

   (g) It should tell you it crashed, then tell you what line of code caused the crash. Put the line that caused the crash in the `README`

   (h) You can do various other things here, such as run `bt` to get a backtrace, which shows you which functions were called to get you to this error. You can run `info regis` to see the current register values.

   (i) Run `disassem` to get a disassembly of the function causing problems. There should be an arrow pointing to the problem code. Cut and paste this line into the `README`

   (j) In the end, what was the cause of the error in this program? (again, put this in the `README`)

6. **Linux Command Line Exploration (1 point total)**

   You can use your Linux machine to find the time/date, just type `date` at the command prompt.

   For years and years there was a program called `cal` bundled with Linux that would make a helpful ASCII art calendar. Sadly they decided to remove this, I guess the 70k of disk space it takes up was deemed too much. You can install it back yourself by doing `sudo apt-get install ncal` if you want to try it out.

   By the default it prints the current month:

```
   September 2024
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30
```

   You can also `cal 2016` or `cal 12 2016`. Beware not to do `cal 16` as that will give you year 16, not 2016.

(a) Run `cal 9 1752` and you will get this odd output.

```
   September 1752
Su Mo Tu We Th Fr Sa
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Is there a bug here? Can you explain what is happening?

7. **Submitting your work**

- Run `make submit` which will create `hw3_submit.tar.gz` containing the various files. You can verify the contents with `tar -tzvf hw3_submit.tar.gz`
- e-mail the `hw3_submit.tar.gz` file to me by the homework deadline. Be sure to send the proper file!

Appendix

1. How to tell if you are on a 32-bit or 64-bit system

    (a) The most obvious way you can tell is if you try to build assembly code and it gives an error like "Error: unknown mnemonic ".

    (b) To tell the operating system kernel type run `uname -m` and see if it says "aarch64" for 64-bit, or "armv7l" or "armv8l" for 32-bit.

    (c) In theory uname tells you what the bitness of the kernel, but on Raspberry Pi you can actually have a 64-bit kernel with 32-bit userspace. To tell in this case you can do something like `file /bin/ls` to see what type of executable `ls` is. It should say 64-bit or 32-bit.

2. Installing a cross-compiler on a 64-bit install.

    (a) You can set up ARM64 systems to compile 32-bit code, but unlike x86 on ARM you need to install a cross-compiler.

    (b) If you're on the network, you can do something like `apt-get install gcc-arm-linux-gnueabihf`

    (c) To run the 32-bit version of the compiler you'll need to prefix it with `arm-linux-gnueabihf-gcc` instead of just gcc.

    (d) If you use this to do your homework, you can modify the Makefile and find the line that says `CROSS=` and change it to be instead `CROSS=arm-linux-gnueabihf-gcc` and in theory it should work.

3. Running 32-but code on a 64-bit install

    (a) In an ideal world this would just work, but in practice you probably have to set up 32-bit system libraries too. I haven't had time to test this all, but usually the process is something like this to set up a "multi-arch system":

    ```
    sudo dpkg --add-architecture armhf
    sudo apt-get update
    sudo apt-get install libc6:armhf
    ```

4. Disassembling code at the command line

    (a) To disassemble the code, run the command
    `objdump --disassemble-all ./integer_print | less`
    the | less at the end says to feed the output of the call to the program less, which lets you scroll backwards and see all the output.