C and Its Relationship to Assembly

In this lab we will learn how to:

1. Use the core C language as an alternative to writing assembly
2. Write a simple C only program
3. Learn how to use gdb with a binary generated from C code

It is a good idea to partner up for this lab so that you can discuss and reflect upon what you are doing.

Setup

Follow the post on piazza to create your github classroom lab repository.

Log into your UNIX environment and clone the repository and change directories into the repository working copy.

Again we have provided pre-written versions of the files. These are in the `examples` sub-directory, if you find yourself short on time feel free to use these files. However, for this assignment since we are getting started on C we really recommend that you create all the files and type the code in yourself. This will also help you understand how the tools and files fit together.

Writing C instead of assembly

The C compiler knows how to write assembly code for our CPU, so let's see how we can use it instead of having to do all the work.

Let's revisit a simple example of summing an array of ten 8 byte integers.

Intel Assembly: sumit.s

```
        .intel_syntax noprefix
        .section .text
        .global sumit
        # code to sum data at XARRAY
        # hard coded length 10
sumit:
        # rax -> sum : sum = 0
        xor  rax, rax
        # rdi -> i : i = 0
        xor  rdi, rdi
loop_start:
         # i - 10
        cmp  rdi, 10
        # if equal done
        je   loop_done
        # sum += XARRAY[i]
        add  rax, QWORD PTR [XARRAY+rdi*8]
        # i=i+1
        inc  rdi
        jmp  loop_start
loop_done:
        ret

.section .note.GNU-stack,"",@progbits
```

And code to call sumit: usesumit.s

```
        .intel_syntax noprefix
        .section .text
        .global _start
_start:
        call sumit

        # save result to stack
        push rax
        # write result to standard
        # output
        mov rax, 1
        mov rdi, 1
        # rsp hold the memory
        # address to the value of
        # rax we pushed
        mov rsi, rsp
        mov rdx, 8
        syscall

        # exit with return 0 to OS
        # as exit status
        mov rax, 60
        mov rdi, 0
        syscall

        .section .data
        .global XARRAY
XARRAY: .quad 1, 2, 3, 4, 5, -15
        .quad 1, 1, 1, 1

.section .note.GNU-
stack,"",@progbits
```

At this point this should feel pretty familiar. To build a binary from this code we need to first assemble the source files into object files and link the object files together to produce a binary. Eg.

```
$ as -g usesumit.s -o usesumit.o
$ as -g sumit.s -o sumit.o
$ ld -g usesumit.o sumit.o -o usesumit
```

Given that our code is writing a binary value to standard output, not ascii, we will need to use another program to translate the result. So to run our app we would do the following:

```
$ ./usesumit  | od -t d8
0000000                    4
0000010
```

A C version of sumit

Ok let's now write a simple C version of sumit. From a syntax perspective this code should feel a little like JAVA, it is however C code. When given to the compiler with the right options, it will produce the assembly code on the right.

csumit.c

```c
extern long long
XARRAY[];

long long  sumit(void)
{
  long long i = 0;
  long long sum = 0;

  for (i=0; i<10; i++) {
    sum += XARRAY[i];
  }
  return sum;
}
```

csumit.s

```
        .file   "csumit.c"
        .intel_syntax noprefix
        .text
        .globl  sumit
        .type   sumit, @function
sumit:
        xor     r8d, r8d
        xor     eax, eax
.L2:
        add     r8, QWORD PTR XARRAY[0+rax*8]
        inc     rax
        cmp     rax, 10
        jne     .L2
        mov     rax, r8
        ret
        .size   sumit, .-sumit
        .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1)
9.4.0"
        .section        .note.GNU-stack,"",@progbits
```

While the compiler has written assembly code that is not quite the way we would have written the code, it does achieve the same thing.

Translating C into assembly

In lecture we discussed that the compiler **toolchain** is used to translate code into binaries. We noted that the toolchain is composed of several steps:

1. pre-processing
2. compilation
3. assembly
4. linking

There is a separate command for each step. To save programmers work and the trouble of remembering how to invoke all the commands, a single “compiler driver” command is usually used. The driver will invoke each of the commands of the toolchain for us. However, we can give the driver options to have it only run some of the steps. In our case the driver command is `gcc`

The following is the command line we used to create the `csumit.s` assembly source file from the `csumit.c` C source file.

```
$ gcc -fno-inline -fno-stack-protector -fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-unwind-tables -Os -S -masm=intel csumit.c
-o csumit.s
```

In the above command we use a lot of options to turn off various features of the compiler we do not need. We do this so that the assembly code it produces is easier to read. Behind the scenes, the `gcc` command given the `-S` option will do the following steps:

1. runs the `pre-processor` command on the c source given to produce a new version
2. runs the `compiler` command on the new c source to create the assembly file called `csumit.s`

Now we can use the assembler to produce an object from this new assembly source file

```
$ as -g csumit.s -o csumit.o
```

Folks usually don’t explicitly do this, rather they let the compiler driver command (`gcc`) do these steps automatically in a hidden way. This can make it feel like the c source is being directly translated into an object file, but it is important to remember that they are being done! C source code is always translated into assembly and then the resultant assembly file is assembled into a corresponding object file. The following is a version of the `gcc` command we would normally use to compile the c source into an object file.

```
$ gcc csumit.c -c
```

This will have the compiler driver command, `gcc`, use all its defaults and it will automatically do the necessary steps of the tool chain:

1. runs the pre-processor command on the c source given to produce a new version
2. runs the `compiler` command on the new c source to create the assembly file called `csumit.s`
3. assemble the assembly file into an object file

Create a `usecsumit` binary

So this part is the same as before, except we are using the new `csumit.o` object file that was created from the `csumit.c` source file. Eg.

```
$ ld -g usesumit.o csumit.o -o usecsumit
```

Now we have a binary that we did not have to write all the assembly for!!!!

```
$ ./usecsumit | od -t d8
0000000                    4
0000010
```

C Calling Conventions

You might be wondering why the code the compiler produced was “compatible” with the assembly code we wrote in `usesumit.s`. Eg. Why did the compiler place the result in `rax`? How could it have known that our “calling” code expected the result to be left in `rax`. Also, why was it ok that the code the compiler generated used the `r8` register? How could it know we did not have an important value in that register? Or imagine if we wanted to pass arguments to our sumit function, how would the compiler have known what registers to use for its arguments?

These are very real problems. To solve them, humans define standards for how assembly code for a particular cpu must be written to be compatible with code produced by a compiler for that CPU. Specifically, these rules are called the calling conventions and they provide rules about what the registers can be used for and how caller code and callee code must cooperate over the registers. Following these conventions allows code to correctly work together.

As it turns out, the calling conventions for Intel Linux C code state that a return value for a function goes into `rax`. So that is why the compiler used `rax` to return the sum. Fortunately, our code was written to expect that! Additionally, the calling conventions state that the `r8` register can be used within a function without having to worry about what it might have been holding. So if code wants to be sure of the value of `r8` it must save it prior to calling any function and restore it afterwards. To write assembly code that can “safely” work with code generated by the compiler we must pay attention to the calling conventions.

Up until now we have been writing all our code by hand and we did not have to care about working with code that other people or the compiler wrote! But now that we are using the compiler it is worth realizing it will pick how to use the registers based on the calling conventions. Our assembly reference sheet has a summary of the INTEL 64 bit Linux calling conventions on it.

INTEL 64 Bit Linux Calling Conventions Summary

INTEL C Linux Calling Conventions :

Defines how registers should be used by caller and callee code. It also defines how arguments and the return value for a C function should be assigned to registers and the stack. The First 6 integer arguments are passed in registers as follows

- Argument 0 : rdi
- Argument 1 : rsi
- Argument 2 : rdx
- Argument 3 : rcx
- Argument 4 : r8
- Argument 5 : r9
- Return value : rax If more than 6 arguments are required, the remainder are pushed on the stack in reverse order (last pushed first). A function's return value must be placed in rax.

The function code (callee) is free to overwrite any of the 7 above registers along with r10 and r11. Calling code (caller) needs to save and restore these registers if it wants to rely on their values. Thus, they are called volatile and caller saved. The values of the remaining general purpose registers (rbx, rsp, rbp, r12-r15) must not be affected by a function, and as such they are called non-volatile and callee saved. Eg. if a function writes them it must restore their value before returning to the caller.

I like to use the following saying to remember what the six argument registers and their order:

- "DIane SIps Delicious Coffee 8 times out of 9"
- means: rdi, rsi, rdx, rcx, r8 and r9

Not perfect but it works for me.

A complete program in C – `main`

Now let's write a complete version of our summing program in C. To make our lives easier, the C toolchain can provide the assembly code for `_start`. This code will take care of all the initial assembly code necessary to get the C standard library ready, figure out what the command line arguments are, and invoke a function we provide that will act as the starting point for our program. This function is called `main`. So now when we write a program using the C compilers, we don't need to worry about writing `_start`. Rather, we write a C function called `main` that the code provided by the toolchain will call, passing in the number of command line arguments, and an array of pointers to the arguments themselves.

main

Let's remove the need for any assembly and write our sum program completely in C.

Step 0: `csumit.c`

Be sure to have created the `csumit.c` file from above

Step 1: `main1.c`

Create a `main1.c` with the following code in it:

```
// preprocessor directive to include the contents of the unistd.h  "header file"
here (needed for declaration of write)
#include <unistd.h>

/*forward declaration of function called in main*/
long long sumit();
/*this line introduces an array with 10 elements, which are of the type long long,
meaning 8 byte signed integers. In the same line, we assign the array elements to be
the values inbetween the curly braces*/
long long XARRAY[10] = { 1, 2, 3, 4, 5, -15, 1, 1, 1, 1 };

/* argc is an integer that holds the number of arguments passed to the program, and
argv is an array of pointers to those arguments */
int main(int argc, char *argv[])
{
    /* Declares the variable sum, which is type long long */
    long long sum;

    /* assign the value returned by the sumit function to the sum variable */
    sum = sumit();

    /* call the write system call, passing 1 as the file descriptor to write to
(standard out), the address of the data we want to write (&sum), and the length to
write in bytes (8) */
    write(1, &sum, 8);

    return 0;
}
```

In the above code we use this libc version of write to call the Linux write system call for us.

Step 2: compile and link

Use the following commands to compile and link our programs

```
$ gcc --static -g -c csumit.c
$ gcc --static -g -c main1.c
$ gcc --static -g -o csum1 csumit.o main1.o
```

Notice we no longer even call `ld`, the linker command, directly. Rather, we let the `gcc` compiler driver command call `ld` with all the necessary options so that all the necessary extra files and options are correctly added.

Step 3: Makefile

Create a Makefile that automates the above, so that when you run `make csum1` it will do all the right things.

> **Static linking: –static**
>
> Notice we pass the `--static` to all the commands. By default, gcc will pass options to the compiler, assembler and linker to create what are called dynamically linked binaries. We override this behavior by passing the `--static` option. This will create binaries that are easier to trace and follow the assembly code. If you are interested in more details regarding dynamic linking, see the virtual memory management lecture.

> **debug info: -g**
>
> Notice we pass the `-g` to all the commands. This tells the commands to add extra information to the binary so that the debugger can better understand how our code relates to the memory contents.

printf

One of the big advantages of going to a C version of the program, and letting the compiler add the C startup code, is we can now use all the standard library functions. So finally we can write programs that convert their binary outputs into ascii before sending them to standard output. While we could have done this before, we would have had to write a lot of painful translation code ourselves. Now we can simply use existing functions like printf.

Modify your program to call printf to translate the binary value of the sum into an ascii representation and send it to standard output.

printf formats

As discussed in the lecture, printf is a very powerful function with a cryptic syntax. It will take time to learn about all that it can do. You can find details with `man 3 printf`, but like other manual pages it can be pretty intimidating at first. The manual page has some examples of its use, but even those can be hard to understand at first. Here is an example to help you with the above task. To better understand this example, have a look at the "length modifiers and conversion specifiers" section of the manual page.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   long long myValue=100;
   printf("%lld %llx\n", myValue, myValue);
   return 0;
}
```

GDB and code written in C

As we have seen, GDB is a very powerful program. It turns out that it knows how to read information in the binary that lets it present a "source" level view of a process. Specifically, it can make it look like we are executing lines of source code – in reality it knows which lines of our source correspond to which assembly addresses, and it steps the right number of instructions to make it appear like we have executed a single line of source code. Remember, one line of C source code can correspond to many assembly instructions. It also reads information about the variables we declare and then lets us work with them regardless of if they are in memory or registers.

1. Start gdb with the provided `csetup.gdb`: `gdb -x csetup.gdb examples/csum1`
2. Set a break point at the main function: `b main`
3. Start a process: `run`
4. Single source line step the program using the `step` command. What is the difference is between `step` and `stepi`?
5. What happens when you : `print XARRAY`? How about `p XARRAY[5]`?
6. What happens when you type `p XARRAY[15]`? What do you think is going on?
7. What happens when you : `print & XARRAY`? What do you think this is telling you about XARRAY? How about `p & XARRAY[5]` and `p & XARRAY[15]` ?
8. Try `whatis XARRAY`
9. What does this do? `x/10gx &XARRAY`
10. What does this do? `set XARRAY[5]=0` Be sure to try `p XARRAY` and `x/10gx &XARRAY` before and after you do the set.
11. When you are in `sumit` what does `where` tell you?
12. When you are in the sumit function what happens when you; 1) `print i` and 2) `print sum` ?
13. Put a break point on the `return sum` line of sumit and then issue the `continue` command and then print the value of sum with `p sum`. Did your changes to XARRAY have an effect?