Chapter Overview

Bookmark this page

This chapter gives some examples of how to code user programs for a Linux environment. It shows you the use of interfaces from the assembly code. The steps provided are done on a Linux host computer with no RISC-V architecture, but the RISC-V cross toolchain and the executables run in the Qemu user mode. However, the programs can be built and run on a RISC-V architecture, as shown in the chapter discussing the development environment.

Learning Objectives

Bookmark this page

By the end of this chapter, you should be able to:

- Write assembly programs using system calls.
- Write assembly programs using C libraries.

Interfacing with the System

Bookmark this page

An operating system provides an interface to use its resources and services by system calls. Further, when linking an object file, the linker uses a linker script that takes care that the object file fits into the target environment.

The Linux operating system offers an interface for system calls. Note that the following steps are done on a Debian system, though they should also work on other distributions. You can use the man pages to get information about the ABI for different architectures. Type in a command line:

```
man syscall.2
```

And look for the tables and the rows with riscv:

```
Arch/ABI    Instruction           System  Ret  Ret   Error    Notes
                                  call #  val  val2
-------------------------------------------------------------------
riscv       ecall                 a7      a0   a1    -

…

Arch/ABI      arg1  arg2  arg3  arg4  arg5  arg6  arg7  Notes
--------------------------------------------------------------
riscv         a0    a1    a2    a3    a4    a5    -
```

You see that the register a7 is used for the number of the system call. The registers a0 and a1 are used for the return values. The parameters to a system call use the register a0 to a5.

```
man syscalls.2

NAME
       syscalls - Linux system calls

SYNOPSIS
       Linux system calls.

DESCRIPTION
       The system call is the fundamental interface between an application and the
Linux kernel.
…
System call                 Kernel        Notes
------------------------------------------------------------------------
write(2)                    1.0
…
```

As an example, we look at the system call for write. To get information about the parameters of write, type:

```
man write.2

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write()  writes  up to count bytes from the buffer starting at buf to the file
referred to by the file descriptor fd.
…
```

Using the system call write allows you to write to the standard output. The file descriptor number 1 is standard output. This is a convention, but also specified in the file `/usr/include/unistd.h`.

However, the information is for programming with the language C. For assembly, we need to know the number of the system call. The file `/usr/include/asm-generic/unistd.h` contains the identifier for the systems call:

```
grep write /usr/include/asm-generic/unistd.h
```

```
/* fs/read_write.c */
#define __NR_write 64
__SYSCALL(__NR_write, sys_write)
…
```

Furthermore, we need a clean exit from our program. This works with the system call exit. The system call exit has the number 93. You can find it with similar steps as with the system call write. With this information, we are able to write a simple “Hello World!” program. To this end, we need to load the registers with parameters and execute the system calls. The parameters of the system call write are the file descriptor in register a0, the address of the beginning of the message text in register a1, and the size of the message in bytes in register a2. The number of the system call is in register a7. The system call is executed by the instruction ecall:

```
# file hello.s
.equ write, 64
.equ exit, 93
.section .text
.globl  _start
_start:
        li      a0, 1
        la      a1, msgbegin
        lbu     a2, msgsize
        li      a7, write
        ecall
        li      a0, 0
        li      a7, exit
        ecall

.section .rodata
msgbegin:
.ascii  "Hello World!\n"
msgsize:
.byte   .-msgbegin
```

Built the program and run:

```
riscv64-linux-gnu-as hello.s -o hello.o
riscv64-linux-gnu-ld hello.o -o hello
./hello
Hello World!
```

Interfacing with Libraries

Bookmark this page

Besides system calls, an operating system comes with a whole bunch of libraries. The libraries are used for dynamic or static linking. Using linking with libraries, functionality provided by them can be used. The interface is usually a C function description. It can be accessed by linking within the C language environment.

Furthermore, the interface with C demands to stick to the RISC-V Calling Conventions. You can instruct the assembler/compiler/linker by parameters which ABI to use. For example, the default ABI for RV64G is LP64D. The ABI specifies the size of C types that the compiler uses. This is important, e.g., when passing arguments to a C function so that you have the correct size.

The linker adds some 'housekeeping' setup code to work within the C setting. This code is run before your program and finally calls your program by an entry point. In the previous examples, we have the entry point by the label _start as the setup code is not needed. To use it, we have to specify the label _main as entry point.

As an example, we use the printf and scanf of the libc in the following assembler program. The manual pages are printf.3 and scanf.3 for further information.

The following program computes the hash function djb2 proposed by D. Bernstein. A hash function computes a value of a fixed size for a given data, e.g., a string. The djb2 begins with hash(0) = 5381 and computes hash(i-1) = 33 hash(i) + character(i), whereby the character(i) stands for the character at the ith position of the input text string. The computation starts with the first character and ends with the last character. If the string has a size of n, hash(n) is finally calculated.

Note that the program is for RV64I, and the functions **addiw** and **slliw** would be **addi** and **slli** for RV32. The original hash function uses 32-bit, although 64-bit would work as well. Thus, we use 32-bit here.

```
# djb2.s
.section .text
.globl main                # run in C 'environment'
main:
    addi sp, sp, -8        # store ra (return address) on stack
    sd ra, 0(sp)

    la a0, prompt          # printf the prompt string
    call printf

    la a0, scanfmt         # scanf from stdin (console)
    la a1, input           # into buffer input
    call scanf             # with format scanfmt

    la a0, input           # process input with djb2
    call djb2
    mv a1, a0

    la a0, result          # print result
    call printf

    li a0, 0

    ld ra, 0(sp)           # restore ra
    addi sp, sp,8
    ret                    # return to caller
```

```
djb2:                   # compute djb2
    li t1, 5381         # init hash = 5381
djb2_loop:
    lb t0, 0(a0)        # process every char of input
    beqz t0, dbj2_end   # until zero appears

    mv t2, t1
    slliw t2, t2, 5     # t2 = hash << 5 = 32 * hash
    addw t1, t1, t2     # t1 = 32 * hash + hash = 33 * hash
    addw t1, t1, t0     # t1 = 33 * hash + char

    addi a0, a0, 1      # next iteration
    j djb2_loop
dbj2_end:
    mv a0, t1           # return hash value
    ret

.section .rodata
prompt:
.asciz "Enter text: "
scanfmt:
.asciz "%127[^\n]"          # scanf max 127 chars and end with return
result:
.asciz "Hash is %lu\n"      # write out the parameter als long unsign.
.section .bss
input:                      # storage for input
.zero 128
```

The file `djb2.s` needs to be linked with the C-compiler, and the built executable needs to know where dynamic libraries are located. This is done for qemu using the environment variable QEMU_LD_PREFIX:

```
riscv64-linux-gnu-as djb2.s -o djb2.o
riscv64-linux-gnu-gcc djb2.o -o djb2
export QEMU_LD_PREFIX=/usr/riscv64-linux-gnu/
qemu-riscv64-static djb2
Enter text: hallo
Hash is 261095189
```

You can inspect the executable using the tool objdump and see that the executable differs from the former examples and that an environment for C is set up.

Character Count

Bookmark this page

The following program counts the number of characters, words, and lines given by the stdin. It uses the getchar function of the C library and processes the input by its ASCII values. To count a word, it is assumed that a word is at least an alphabetic letter or a decimal digit. Each character has a number in ASCII code. The ASCII code of a line feed is 0xa. The ASCII codes of decimal digits are from 0x30 to 0x39 for 0 to 9, of the upper case letters from 0x41 to 0x5a for A to Z, and of the lower case letters from 0x61 to 0x7a for a to z. The following source code has line numbers to discuss some instructions:

```
00 # wordcount.s
01 .section .text
02 .globl main             # run in C 'environment'
03 main:
04     addi sp, sp, -40    # store ra (return address) and saved regs on stack
05     sd ra, 0(sp)
06     sd s0, 8(sp)
07     sd s1, 16(sp)
08     sd s2, 24(sp)
09     sd s3, 32(sp)
10
11     li s0, 0            # counter chars
12     li s1, 0            # counter line feeds
13     li s2, 0            # counter words
14     li s3, 0            # indicator if current input is in word
15
16 loop:
17     call getchar        # get input from stdin in a0
18     bltz a0, end       # if end of file (eof is -1) goto end
19
20     addi s0, s0, 1      # count characters
21
22     li t0, 0xa          # is linefeed (ascii 0xa)?
23     bne a0, t0, nolf   # no -> continue
24     addi s1, s1, 1      # yes -> count
25 nolf:
```

```
nolf:

                          # is this a word: char digit or alphabet?
    addi t0, a0, -0x30  # digits go from 0x30 to 0x39
    li   t1, 0x9        # if (char-0x30) >= 0 and <= 0x9 then digit
    bleu t0, t1, aldi  # trick: treat negative value as unsigned
                        # value (or neg. as unsign.) > 0x9, continue
    andi t0, a0, ~0x20  # 0x60 to range > 0x40, lower to upper cast
    addi t0, t0, -0x41  # letter go from 0x41 to 0x5a
    li   t1, 0x19       # (char-0x41) >= 0 and <= 0x19, then alphabet
    bleu t0, t1, aldi  # trick again
                        # reached here, then not in word (anymore)
    add s2, s2, s3      # count word, indicator is one if word else 0
    li s3, 0            # clear indicator
    j loop

aldi:
    li s3, 1            # char is part of word, indicate for word counter
    j loop
end:
    la a0, result       # print result
    mv a1, s1
    mv a2, s2
    mv a3, s0
    call printf

    li a0, 0

    ld s3, 32(sp)       # restore saved regs.
    ld s2, 24(sp)
    ld s1, 16(sp)
    ld s0, 8(sp)
    ld ra, 0(sp)        # restore ra
    addi sp, sp,40
    ret                 # return to caller

.section .rodata
result:
.asciz "Lines: %u  Words: %u  Chars: %u\n"      # write out result
```

The instructions of lines 04-09 and 53-58 (re)store the values of the registers ra and s0-s3 according to the ABI. The registers s0-s2 are used for the counters. The register s3 becomes one if the input character is recognized as a letter of a word. These registers are initialized with zero (lines 11-14). In a loop, the function **getchar** is called, and the loop is exited if **getchar** returns -1, end of file (lines 16-18). If **getchar** returns a char, the counter for chars is incremented (line 20). If the char is a line feed, the counter for line feeds is incremented; otherwise, not (lines 21-25). A char is tested if it is a decimal (lines 27-30) or a letter (32-36). In case of a decimal or letter, the register s3 is set to 1, and the loop begins (lines 41-43). The register s3 indicates that a word is detected w.r.t. the assumptions. If the character is not a decimal or a letter, the indicator value (register s3) is added to the word counter. The indicator is zero as long as there is no word. It changes to one for a word character and back for a nonword character.

Lines 28-30 and 32-35 determine if a char is in a specific range. ASCII code is from 0 to 127. The first check is if a value (for decimals) is between 0x30 and 0x39. Thus, the value 0x30 is subtracted from the current character. The result is below zero for values less than 0x30. However, if the result is treated as an unsigned number, it is a number greater than 0x9. The result is between 0x0 and 0x9 only if the character is a decimal. Hence, the comparison used is if-less-or-equal-unsigned (bleu). Lines 33-35 work similarly for the range 0x41 to 0x5a, except that line 32 clears the 5th bit in the current character by andi with the inverse of 0x20 (note the tilde ~). By clearing the bit, all values greater or equal 0x60 are mapped in the range starting with 0x40, but values below 0x60 are unaffected. This allows us to use the test (lines 33-35) for the (mapped) range 0x61 to 0x7a, too.

The program is stopped with return followed by **CTRL-D**. Using the **<**, you can also redirect a file as input to the program.

```
riscv64-linux-gnu-as -o wordcount.o wordcount.s
riscv64-linux-gnu-gcc -o wordcount wordcount.o
export QEMU_LD_PREFIX=/usr/riscv64-linux-gnu/
qemu-riscv64-static wordcount
word Word     .... hello
next line     should be two lines
Lines: 2  Words: 9  Chars: 60
```

Prime Number Check

The next program asks for a number. The number is checked if it is a prime number. The result is printed out. The program implements the Sieve of Eratosthenes. The C function **scanf** takes the parameter string "%u" which reads an unsigned integer. Labels beginning with .L are local labels and are not exported to the symbol table by the assembler.

```
# prime.s
.equ maxnb, 0x100000
.section .text
.globl main             # run in C 'environment'
main:
    addi sp, sp, -8    # store ra (return address) on stack
    sd ra, 0(sp)

    la a0, prompt       # printf the prompt string
    call printf

    la a0, scanfmt      # scanf from stdin (console)
    la a1, input        # into buffer input
    call scanf          # with format scanfmt

    blez a0, .Lerr      # input error

    la  a1, input       # check if input number n fits
    lw  a1, 0(a1)
    li  t0, maxnb
    bge a1, t0, .Lerr

    la a0, input        # process input with sieve
    call sieve

    bnez a0, .Lp1
.Lp0:
    la a0, outno
    j  .Lpp
.Lp1:
    la a0, outyes       # print result
    j  .Lpp
.Lerr:
    la a0, error
.Lpp:
    call printf
```

```
    li a0, 0

    ld ra, 0(sp)        # restore ra
    addi sp, sp,8
    ret                 # return to caller

sieve:
    # input: register a0 points to number n
    # that is checked if it is a prime.
    # output: if n is prime a0 is one else zero
    # sieve of Erastosthenes
    # init array with numbers
    lw t1, 0(a0)        # nb to check
    li t2, 2            # counter start with 2
    la t3, array        # pointer to array
.Ls0:
    sw   t2, 8(t3)      # set item to index, 8() is begin with index 2
    addi t3, t3, 4      # increment by four for word size
    addi t2, t2, 1      # counter
    ble  t2, t1, .Ls0   # until counter == nb to check

    # array has now the values: 0 0 2 3 4 5 6 7 8 9 10...

    # non-primes are cancelled out
    # by setting their array items to zero
    li t2, 2            # start with 2, t2 is index i
    la t3, array        # t3 is pointer to array
.Ls1:
    lw   t4, 8(t3)      # t4 is current array item (offset by 2)
    beqz t4, .Ls3       # no prime, continue at .Ls3

    mul t4, t2, t2      # t4 = t2 * t2; t4 is index j
.Ls2:
    slli t5, t4, 2      # t5 = t4 * 4 for offset (words) in array
    add  t5, t3, t5     # t5 = t3 + t5; t5 is address in array for j
    sw   x0, 0(t5)      # set entry to 0, no prime nb, array[j] = 0
    add  t4, t4, t2     # t4 = t4 + t2; j += i
    ble  t4, t1, .Ls2   # cancel out all multiples of i for i < n

```

```
.Ls3:
    addi t2, t2, 1      # continue with next number
    mul  t0, t2, t2     # as long as n*n > index
    ble  t0, t1, .Ls1

    slli t0, t1, 2      # use n as index
    add  t0, t3, t0     # compute address in array
    lw   t0, 0(t0)      # load its item
    snez a0, t0         # set a0 to 1 if array[n] != 0

    ret


.section .rodata
prompt:
.asciz "Enter number (<1048576): "
scanfmt:
.asciz "%u"                # scanf an unsigned int number
outyes:
.asciz "is a prime number.\n"
outno:
.asciz "is not a prime number.\n"
error:
.asciz "wrong input.\n"
.section .bss
input:                      # storage for numbers
     .word 0
array:
     .zero 4*maxnb          # max number storage
```

Lines 01-41 deal with the input of the number and the output of the result. The sieve of Eratosthenes is done from lines 43-86. Lines 89-104 are for data.

The first part of the prime number check initializes an array (lines 49-56). The main part is canceling out multiples of all numbers starting with the number two (lines 62-79). To this end, each item of the array is processed. If it is already canceled out (== 0) the next item is taken (lines 64-66). Otherwise, all multiples of the number are canceled out, beginning with the number squared (lines 68-74, inner loop) and ending with the input number. The next item of the array is then taken and processed (lines 64-79, outer loop). This continues as long as the number is less or equal to the input number (lines 77-79). The item of the array corresponding to the input number is finally checked whether it is canceled out or not (lines 81-84).

```
riscv64-linux-gnu-as -o prime.o prime.s
riscv64-linux-gnu-gcc -o prime prime.o
export QEMU_LD_PREFIX=/usr/riscv64-linux-gnu/
qemu-riscv64-static prime
Enter number (<1048576): 1034123
is a prime number.
```

Chapter 5 Lab Exercise

Write a program that analyzes the input given by stdin. The program should count the number of decimals, number of letters, and number of non-letter characters:

- Write a loop that reads in the text by the C function **`getchar`** and breaks the loop by EOF.
- Check each input character if it is a decimal, letter, or none of both.
- Increment counters for the result of the checks.
- Use the C function **`printf`** for printing the results.