Utilities

CS3500 Operating Systems Jul-Nov 2025

Sanjeev Subrahmaniyan S B

EE23B102

1 Objective

This lab is aimed at setting up xv6 RISCV on a Docker container, booting into the operating system and using system calls to print a simple hello world text on the screen.

2 Assembly Code and Execution Output

2.1 Assembly Code

The contents of the file HelloWorld.S:

```
.global main    // makes this function available to the compiler and linker

main:   li a0, 1        // choosing stdout(display) as the file descriptor
        la a1, saddr    // pointer to the string to be printed
        li a2, 34       // length of the printed string
        li a7, 16       // index for the write sys call
        ecall           // changes the privilege of the cpu to handle the syscall

        li a0, 0        // return status for the exit syscall - 0 for success
        li a7, 2        // index for the exit sys call
        ecall

.data   // directive to place the string in data segment of the memory
saddr:      .ascii "Hello EE23B102! Welcome to CS3500\n"
// string to be printed, in ascii format
```

2.2 Execution

```
berserker@berserker-Pavilion-Pro:~$ docker run -it -v ~/sem5/os/xv6-riscv:/home/os-iitm/xv6-riscv svkv/riscv-tools:v1.0
root@6cb4bfed85bb:/home/os-iitm# cd xv6-riscv/
root@6cb4bfed85bb:/home/os-iitm/xv6-riscv# make qemu
8.0
riscv64-unknown-elf-gcc    -c -o user/HelloWorld.o user/HelloWorld.S
riscv64-unknown-elf-ld -z max-page-size=4096 -T user/user.ld -o user/_HelloWorld user/HelloWorld.o user/ulib.o user/usys.o user/printf.o user/umalloc.o
riscv64-unknown-elf-objdump -S user/_HelloWorld > user/HelloWorld.asm
riscv64-unknown-elf-objdump -t user/_HelloWorld | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > user/HelloWorld.sym
mkfs/mkfs fs.img README user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user
/_grind user/_wc user/_zombie user/_HelloWorld user/_PingPong
nmeta 46 (boot, super, log blocks 30, inode blocks 13, bitmap blocks 1) blocks 1954 total 2000
balloc: first 840 blocks have been allocated
balloc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,format=raw,id=x0
-device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ HelloWorld
Hello EE23B102! Welcome to CS3500
$
```

2.3 Instruction to Run

The given output can be reproduced by following these steps:

1. Place the riscv assembly code shown above into a file HelloWorld.S in xv6-riscv/user/
2. Add an entry in the xv6-riscv/Makefile under the UPROGS. The entry should be for the file "HelloWorld" and is thus of the form "$U/_HelloWorld".
3. Run "make clean && make qemu" to run the operating system and to compile the written code.
4. In the running shell, run the command "HelloWorld" to obtain the viewed output.

3 Issues Faced

I did not face any problems in setting up the Docker environment or pulling the xv6 repository while following the instructions provided in the problem document.

4 Explanation of Assembly Program

Each line in the assembly program has been briefly explained in the comments alongside the code. A more detailed explanation of the same is presented below.

1. **.global main**: This directive exposes the "main" function to the linker. When this is not done, the linker treats "main" as local to the HelloWorld.S file and would flag any other reference to this function as undefined. As the entrypoint is main, this directive ensures the linker can use this in calls to the function.
2. **li a0, 1**: From the source files syscall.c and syscall.h, one can observe that the for the write syscall, the destination file descriptor is placed in register a0. As we wish to write to the display, which is the STDOUT, the file descriptor 1 is used. Note that the li assembly instruction is used to load an immediate value into a target register.
3. **la a1, saddr**: Similarly, from the source code, it is seen the message to be printed is passed as a pointer to the first character of the string(the memory location of the string) into the register a1 for the write syscall. The la assembly instruction is used to load an address in the memory to a register.
4. **li a2, 34**: The length of the string to be printed("Hello EE23B102! Welcome to CS3500\n" is placed in the register a2.
5. **li a7, 16**: From the syscall.h file, it can be seen that the index for SYS_write is 16. This is expected to be placed in the register a7 using which the trap handler calls the appropriate system call.
6. **ecall**: The enviroment call instruction is used to trap into supervisor mode, where the trap handler processes the system call according to the values placed in the registers as done above.
7. **li a0, 0**: Moves the first argument for the exit syscall into a0. This is done to tell the process exits without any errors.
8. **li a7, 2**: Similar to the one for the write syscall, the index for the exit syscall is 2.

9. **ecall**: Again performs an enviroment call to process the exit.
10. **.data**: Directive to place the data in the data segment of the memory
11. **.ascii "Hello EE23B102! Welcome to CS3500\n"**: The string which is to be printed, along with a directive telling the string follows the ASCII encoding. Note that this is also prefixed with an identifier saddr, which is used to refer to this string while moving the pointer to the first character as an argument to write syscall.