PrairieLearn ARM32 Assembly Autograding with QEMU (Linux)

Senior Design May 2024 - 33

Mitch Hudson
Tyler Weberski
Chris Costa
Andrew Winters
Carter Murawski
Matt Graham

Table of Contents

Introduction

This document outlines the integration of QEMU into a new docker container for use with PrairieLearn as an auto-grader. It will go over installation of the code, as well as how to create new auto-grading containers in the future. Finally, it will show usage of the container in a test question, as well as cover some common issues with a resources page.

GitHub Repository and Installation

All of the code for the auto-grader is kept in the GitHub repo. To use with the production server, simply clone the code onto the system and run the included build-image.sh script. This script will create the image that can then be used in your question's info.json file.

Cloning and Building

On the production server, run the following command to clone the code:

```
$ git clone https://github.com/myriath/PrairieLearnARMGrader.git
```

This will create a folder called PrairieLearnARMGrader, then run the included script to build the container:

```
$ cd PrairieLearnARMGrader
$ ./build-image.sh
```

A successful build will output the following, along with the current image name and version to be used with PrairieLearn:

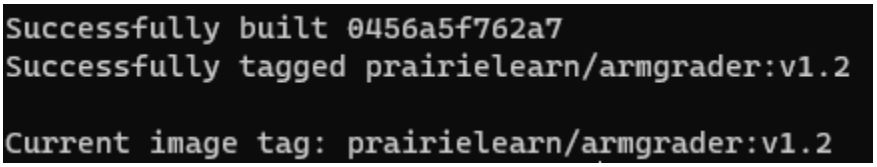

Fig 1: Successful output of build-image.sh

Repository Layout

The repository is fairly simple, consisting of a few files and folders. The files in the root directory are the Dockerfile, requirements.txt, build-image.sh, and the git files. The Dockerfile is used to tell docker how to construct the image, requirements.txt is used for making sure the Python installation on the docker container has everything it needs, and the build-image.sh is a script to tell docker to rebuild the image. The armgrader directory holds the Python code that will be used to run tests in your question. Finally, the questionTemplate directory holds two template assembly files that can be used to easily write new tests.

Dockerfile

The Dockerfile is fairly straightforward. It starts by using Ubuntu 22.04 as the base image, then tells debian it is non-interactive and has a maintainer label. The next couple of commands update the image using apt and install all dependencies for the auto-grader. These include Python 3.10, qemu-user, and the GNU Linux to ARM Linux 32-bit cross-compiler libraries. Next, the container moves the Python installation to a more regular place before copying requirements.txt and installing the Python requirements with PIP. Then, it sets some language environment variables before setting the Python path and encoding environment. Finally, it adds a new non-root user for sandboxing and copies the armgrader directory to the image.

```
1  FROM ubuntu:22.04
2
3  LABEL maintainer="wmhudson@iastate.edu"
4
5  ARG DEBIAN_FRONTEND=noninteractive
6
7  RUN apt-get update
8  RUN apt-get install -y python3.10 python3-pip make qemu-user gcc-arm-linux-gnueabihf binutils-arm-linux-gnueabihf binutils-arm-linux-gnueabihf-dbg
9
10 RUN ln -sf /usr/bin/python3.10 /usr/bin/python3
11
12 COPY requirements.txt /requirements.txt
13 RUN pip3 install --no-cache-dir -r /requirements.txt
14
15 ENV LANG=en_US.UTF-8
16 ENV LC_LANG=en_US.UTF-8
17
18 ENV PYTHONIOENCODING=UTF-8
19 ENV PYTHONPATH=/armgrader/:/grade/serverFilesCourse
20
21 RUN groupadd sbuser
22 RUN useradd -g sbuser sbuser
23
24 COPY armgrader /armgrader
```

Fig 2: Dockerfile contents

Requirements.txt

These requirements are standard for PrairieLearn, and are just the same requirements used by the C grader. Any additionally needed Python modules inside the grader container can be defined here.

Armgrader Directory

Inside this directory are the files that will be extended when creating auto-grader tests. Currently, the only file is armgrader.py, which defines a new ARMGrader class which can be extended to define the ‘tests()’ method. This method gets executed in the class’s ‘start()’ method, and will be where developers put their testing code. Aside from the name of the class, and the methods for compiling and testing, this grader works with the same options as the C auto-grader. Important changes are outlined in the tests.py section under Example Question.

Question Template Directory

This directory holds two files, main_template.s and syscalls.s. Syscalls.s is the same file talked about in Example Question. It contains basic implementations for various Linux system calls like dup, dup2, open, close, read, write, and an implementation for rand. All of these functions and their usage is detailed in the main_template.s comments above _start. *Note: if the kernel for the docker container is updated in the future, you may need to update the syscall constants in this file!*

Main_template.s is the meat of the grader, and it aims to make it easy to create new questions on the fly. There are three important sections to keep track of: TESTING VARS, CODE TESTING, and ANSWER FUNCTIONS; each enclosed by comments labeling them.

TESTING VARS

TESTING VARS is the first section, at the top of the code below STDOUT. This is where any heap variables used in testing should be placed. For example, a and b from HW12 Q2a should be placed here. You can do this with ‘VAR_NAME: .word 0’. Arrays are the same but with multiple values delimited by commas: ‘VAR_NAME: .word 0,0,0’ is equivalent to an ‘int VAR_NAME[3]’.

CODE TESTING

This section is in _start, after the boiler plate functions for preparing rand() and protecting stdout and stderr. Here is where you will write the code to execute both the students' functions and your own answer functions, compare the outputs, and either continue on to the 'b success_exit' if the students' code was correct, or branch to error_exit if it wasn't.

ANSWER FUNCTIONS

Finally, the answer section is where you might want to put the answer functions. Technically, since they are just labeled sections of code, the answer functions can go anywhere outside of _start, but this was put here to make it easier for people new to assembly.

Example Question

Since this grader is heavily based off of the code for the C auto-grader, a lot of the process is identical to creating C questions. First, configuration for the question is put into info.json, then the question layout is written into the question.html file as per PrairieLearn specifications, any randomization and question generation is done in the server.py file, and finally tests are placed into the tests subdirectory. You can see an example question in the repo in the Resources section.

```
question-folder/
├── tests/
│   ├── main.s
│   ├── ...
│   └── tests.py
├── info.json
├── question.html
└── server.py
```

Fig 3: Question folder layout

Info.json

```
{
    ...
    "gradingMethod": "External",
    "externalGradingOptions": {
        "enabled": true,
        // Image name from build-image.sh
        "image": "prairielearn/armgrader:v1.2",
        "timeout": 100,
        // Same entrypoint as the c-grader
        "entrypoint": "python3 /grade/tests/tests.py"
    }
}
```

The important difference for this grader is changing the image from the c-grader image to the one generated by build-image.sh. Everything else should be mostly the same as configuring the external C auto-grader.

Question.html

Question.html uses the same syntax and formatting as other questions in PrairieLearn. For help setting up external grading questions, see the Resources page.

Server.py

Server.py also uses the same syntax and formatting as other questions in PrairieLearn. For help with this, see the Resources page.

Tests Subdirectory

Inside the tests subdirectory, you will place all of the assembly source files needed to run tests, as well as the tests.py file that will tell the container what to do to test the students' code. This is NOT the directory for any student code; that will be moved to the container by PrairieLearn.

main.s

Main.s is a colloquial term used to describe any main program file that will execute student code. Student code should be set up in a way that it can be called from the main.s file once compiled, usually by calling the .global directive within the student's code on any of the functions there. Then, the functions can be called from main using bl <function name>. Calling conventions should follow standard ARM assembly conventions. For details on the main_template.s file that is packaged with the ARM auto-grader, see Question Template Directory in Repository Layout.

syscalls.s

NOTE: If you are getting weird issues with the code, check if you have updated the Linux kernel! Ubuntu 22.04 uses kernel 5.15, and that is what these syscalls are written for, if you update the distro / kernel in the future, you may need to change the syscall constants!

Syscalls.s stores code that implements basic Linux syscalls, specifically those for reading, writing, dup, dup2, and a few others.

There is also a function called 'protectoutputs' that does the same thing as the start of the C grader's main function; duplicating stdout to a separate file descriptor before

redirecting the default stdout and stderr file descriptors to /dev/null. This prevents students from easily writing to stdout in their code to bypass the testing.

The other function available here is called ‘rand’ which reads a random 32-bit word from /dev/urandom and returns it in R0. This can be used to generate random data for extensive testing of student code.

tests.py

Tests.py holds the code that actually compiles, runs, and tests the student’s code. All of the options here are the same as the C auto-grader’s with a few exceptions.

First, the ‘compile_file()’ function takes far fewer arguments, only needing a list of ARM source files to compile and the name of the compiled output file. *Note: the directory this runs from is where the student code is, any code from the tests directory is put into /grade/tests/*

Second, a new function was created for running the newly created ARM programs, ‘test_qemu()’. This command takes the same arguments as the regular ‘test_run()’ function but forces the execution through QEMU. Without this, the execution simply fails because of the incorrect architecture.

Third, another new function was added for creating a dump file of the registers at a desired breakpoint: ‘dump_registers()’. This command takes 1 required argument and has 3 optional arguments.

Required: *executable:* String containing the name of the program to run.

Optional: *port:* Port for the GDB server to use, only need to worry if you are running multiple of this command simultaneously. Defaults to 1000
Optional: *output_filename:* Filename of the dump. Defaults to ‘run.txt’
Optional: *breakpoint:* GDB understood breakpoint argument (the part that comes after the ‘b’ command). Defaults to the exit label.

Writing Tests

Here is our _start code from HW12 Q2a as an example for this section:

```
.global start
_start:
    bl open_rand
    bl protectoutputs
    ldr r1, =STDOUT
    str r0, [r1]

    mov r4, #100
    mov r5, #0
    b _start_check

_start_loop:
    // randomize var_a, var_a_ans, var_b, var_b_ans
    bl rand
    ldr r1, =var_a
    str r0, [r1]
    ldr r1, =var_a_ans
    str r0, [r1]
    bl rand
    ldr r1, =var_b
    str r0, [r1]
    ldr r1, =var_b_ans
    str r0, [r1]

    // call functions to test
    ldr r0, =var_a
    ldr r1, =var_b
    bl _arithmetic     // call student code

    ldr r0, =var_a_ans
    ldr r1, =var_b_ans
    bl ans_arithmetic  // call answer method

    // test function outputs
    ldr r0, =var_a
    ldr r0, [r0]
    ldr r1, =var_a_ans
    ldr r1, [r1]
    cmp r0, r1
```

```
    bne error_exit

    ldr r0, =var_b
    ldr r0, [r0]
    ldr r1, =var_b_ans
    ldr r1, [r1]
    cmp r0, r1
    bne error_exit

_start_check:
    add r5, #1
    cmp r5, r4
    ble _start_loop

    // ------------- END CODE TESTING ------------- //

    // testing was successful! print message and exit
    b success_exit
```

The first block of the _start function is from the template and only serves to prevent student code from writing their own messages to stdout. After that, we set up a loop to loop through the test 100 times.

Inside the loop, the general idea is to perform unit tests against the functions written by students. Here, we call rand a couple of times and place the values into the a and b variables. One set is for the student's code to use and the other is for the answer code to use. Then, we simply call both the student function and the answer function before checking that they result in the same values, and continuing the loop. If there is any input that results in differences between the student's and answer's code, we break from the loop and move to the error_exit function that prints a failure message and exits the program.

As you can see, the process for writing question tests is fairly straightforward: set up global variables, write an answer function, create a loop to test many random values, run both the student and answer code, and check that they create the same output. This is extremely flexible in how you want to set up the functions, but the important thing is incorrect output should call error_exit and correct output should call success_exit.

Debugging Tests

Debugging assembly code running through QEMU can be kind of annoying. First, the program needs to be run using QEMU with a GDB server, then the server needs to be connected to.

To do this, start a docker container using the armgrader image and connect a volume to the tests folder in the question. Then, individual steps are as follows:

```
$ arm-linux-gnueabihf-as -g <s files list> -o <o file> // compile program
$ arm-linux-gnueabihf-ld -g <o file> -o <executable>   // link program
$ qemu-arm -g <port> <executable> &   // Starts QEMU in background
[1] <pid>
$ gdb-multiarch <executable>          // Loads the symbol table
> target remote localhost:<port>      // Connects to QEMU GDB server
> layout asm                          // Sets the layout for GDB
```

To make this process simpler, a one liner can be used instead (after compiling and linking):

```
$ qemu-arm -g 1000 <exe> & gdb-multiarch <exe> -ex 'target remote localhost:1000' -ex 'layout asm'
```

This starts QEMU in the background, starts GDB, and automatically connects to the remote server and sets the layout.

Once in GDB, using ‘info registers’ or ‘i r’ will dump every register of the system, and you can get specific registers using ‘p $r0’ (replace r0 with the register you want). You can also print the value in memory of an address stored in a register using a dereference: ‘p *(int*)($r0)’. Breakpoints can be set on function names using ‘b <function>’ or specific line numbers with ‘b <filename>:<line #>’. To control execution use ‘c’ for continue, ‘s’ for step-in, or ‘n’ for step-over.

GDB is extremely complex and there are a lot of tricks you can find, but this should cover the basics for using GDB inside the docker container with QEMU.

Resources

Git Repositories

ARMGrader: https://github.com/myriath/PrairieLearnARMGrader
Test Question:
https://git.ece.iastate.edu/sd/sdmay24-33/-/tree/mitch-asm-test/questions/ASM_TEST?ref_type=heads

PrairieLearn Resources

C-grader options: https://prairielearn.readthedocs.io/en/latest/c-grader/
Sample question.html:
https://github.com/PrairieLearn/PrairieLearn/blob/master/exampleCourse/questions/demo/autograder/c/square/question.html
Server.py docs: https://prairielearn.readthedocs.io/en/latest/question/#question-serverpy

QEMU and Assembly Help

Tutorial for QEMU and cross-compilation:
https://azeria-labs.com/arm-on-x86-qemu-user/
How to assemble .s files from .c files:
https://stackoverflow.com/questions/44275519/converting-c-source-to-arm-assembly
Docker image tagging:
https://stackoverflow.com/questions/38986057/how-to-set-image-name-in-dockerfile
Linking C with ARM:
https://stackoverflow.com/questions/63492410/nasm-ld-cant-find-printf

ARM Linux Help

Syscall table for Linux Kernel v5.15 (Ubuntu 22.04) (for syscall numbers):
https://github.com/torvalds/linux/blob/v5.15/arch/arm/tools/syscall.tbl
Ubuntu 22.04 syscall manpages (return and argument information for syscalls):
https://manpages.ubuntu.com/manpages/jammy/en/man2/syscalls.2.html