Phase 0: Getting Started with SPEDE

Introduction

Throughout the course, we will be developing a simple operating system spread across multiple phases. This first phase (phase 0) is to set up our development environment and prepare for us to participate in subsequent project phases with your team members.

This environment is referred to as SPEDE (System Programmers Educational Development Environment). SPEDE was originally developed here at CSUS for this course with contributions over a number of years from professors, teachers, and students as well. Originally, this environment consisted of physical hardware to develop and run code but is now run using virtual machines and the environment has now been updated to use modern C compilers, debuggers, and tools.

Why SPEDE?

Developing an Operating System is not a simple task. It is a specialized discipline that requires deep understanding of computer hardware and system architectures and the ability to put computer science and engineering skills into practice.

In this course, the gap is shortened a bit for you and SPEDE helps facilitate some of the tedious steps to to get from system power-up to runninng (and more importantly: *debugging*) our code.

Within the SPEDE Virtual Machine, all of the development tools that you will need to write code, compile, debug, and run your operating system are available. It is a flexible environment that you can build upon and develop further.

Table of Contents

- Important Notice!
- Objectives & Outcomes
- Requirements
    - Part 1: Accessing GitHub Classroom
    - Part 2: Setting up the Development Environment
    - Part 3: Building and Running Code
    - Part 4: Debugging
- Project Submission
- Grading Metrics
- Requirements Checklist
- Additional Resources

Important Notice!

The SPEDE Virtual Machine is a 64-bit x86 linux operating system which has specific minimum system requirements to support.

Officially, we will be using Oracle VirtualBox to run the SPEDE Virtual Machine.

However, it *should* be possible to use other virtual machine hosts if you would prefer: VMware Workstation Player, VMware Fusion Player, or Parallels Desktop on Intel-based Apple computers. Technical support will only be provided for VirtualBox due to limited resources.

Minimum system requirements for your computer to run the SPEDE Virtual Machine include:

- Hardware Requirements
  - **x86-64 CPU (Intel or AMD)**
  - Minimum Dual Core CPU
  - Minimum 15GB Disk Space Available
  - Minimum 4GB RAM
- Supported/Tested Host Operating Systems
  - PC (Windows)
    - Windows 11
    - Windows 10
    - Windows 7
  - Apple (macOS)
    - macOS 13 Ventura
    - macOS 12 Monterey
    - macOS 11 Big Sur
    - macOS 10.15 Catalina
    - macOS 10.14 Mojave
    - macOS 10.13 High Sierra
    - macOS 10.12 Sierra

If you do not have access to a computer system meeting the above hardware and software requirements, please reach out immediately to your professor to request a laptop checkout from CSUS IRT.

Objectives & Outcomes

The objectives for this phase are to ensure that you:

1. Have access to the GitHub classroom and organization used for project repositories
2. Are able to navigate the development environment
3. Can compile code into an executable operating system image
4. Can run the operating system image in the emulation environment
5. Can debug code that you write and run
6. Have demonstrated some of the key pre-requisites for this course
    - Ability to program in the C Programming language
    - Demonstrate knowledge and implementation of simple data structures

At the end of this phase, you will be ready to embark on developing your first operating system!

Requirements

Part 1: Accessing GitHub Classroom

We will be using GitHub to submit all programming assignments and programming project phases. As a result, a GitHub account must be associated with you to access initial repositories and commit code to your individual and team repositories.

Requirements for this part are to:

1. Create a GitHub account if you do not already have one: https://github.com/join
    - Note: *If you already have a GitHub account that you would like to use, you do not need to create a new one just for this course*

2 .Link your github account to your student name in the Github Classroom.

Part 2: Setting up the Development Environment

All **development**, **debugging**, and **testing** will take place using the SPEDE Virtual Machine. The virtual machine needs host software to execute. The supported host software in this class is VirtualBox, which is free.

The Virtual Machine itself is a Linux-based desktop environment that contains a compiler, debuggers, libraries, tools, scripts, and the target hardware emulator that will run our compiled operating system.

Step 1: Download and Install VirtualBox Download and install VirtualBox 7.0.6:

- Windows: https://download.virtualbox.org/virtualbox/7.0.6/VirtualBox-7.0.6-155176-Win.exe
- macOS: https://download.virtualbox.org/virtualbox/7.0.6/VirtualBox-7.0.6-155176-OSX.dmg
- Linux: https://www.virtualbox.org/wiki/Linux_Downloads

Step 2: Download and Import the SPEDE Virtual Machine

Download the SPEDE Virtual Machine from OneDrive: SpedeVM.ova

Import the Virtual Machine into VirtualBox:

1. Open VirtualBox
2. Select the `File` then `Import Appliance` menu option
    1. Note: *No settings should need to be modified during the import process*
3. Modify the Base Folder if you would like to choose a different location to install other than the default

Some option system configuration may be needed, depending on your environment.

- Enable virtualization in your PC BIOS/UEFI settings
- Adjust display resolution/scaling for the virtual machine on high-resolution displays (4k, Apple Retina, etc.)
- Adjust 3D rendering settings for the virtual machine depending on your video card/GPU configuration

Recommended configuration changes

- If you have sufficient memory on your host system, consider increasing the amount available to the virtual machine (default is 2GB, recommended is 4GB or more)
- If you have sufficient CPU cores on your host system, consider enabling an additional CPU core (default is 2 core, recommended is at least 4 cores)

Step 3: Launch the SPEDE Virtual Machine and Personalize

Within VirtualBox, start the SPEDE Virtual Machine. It may take a few moments to start up, but you should be logged in to the SPEDE desktop automatically.

Note: If for any reason you need to log in (such as to perform an action as root, log in via SSH if you enable remote access, etc.) the following credentials are set up by default:

- Username: `spede`

- Password: `spede`

When you launch the SPEDE Virtual Machine, you should arrive to a graphical user interface/desktop, such as the following:

Step 4: Connecting to GitHub

One you are running the SPEDE Virtual Machine, you will need to configure it for your personal use.

Git Configuration

Open a terminal so that you can configure your name and e-mail within git:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@csus.edu"
```

Substitute `Your Name` with your name, and `your.email@csus.edu` with your email address. Note: *You do not need to use your csus.edu email if you have another email address you prefer to use.*

To confirm that you have configured your name and e-mail address, you can view the configuration with the following command:

```
git config --list
```

SSH Keys

Once you have configured git, you will need to create an SSH key so you can authorize yourself with GitHub and be able to clone repositories and commit code.

Within your terminal, run the following command to generate a new SSH key:

```
ssh-keygen -t ed25519 -C $(git config --get user.email)
```

You may be prompted to add a passphrase. You are not required to do so, but it does add an extra layer of security (*always a good practice*).

Once you have created an SSH key, you will need to copy the public key, such as:

```
cat ~/.ssh/id_ed25519.pub
```

This public key will then need to be added to your GitHub account. For instructions on how to do this, please refer to Adding a new SSH key to your GitHub account from GitHub.

Once you have added your SSH public key to GitHub, you can test the connectivity by running the following command in a terminal within the SPEDE Virtual Machine:

```
ssh -T git@github.com
```

On your first connection, you may see a warning, such as the following:

```
> The authenticity of host 'github.com (IP ADDRESS)' can't be established.
> RSA key fingerprint is SHA256:p2QAMXNIC1TJYWeIOttrVc98/R1BUFWu3/LiyKgUfQM.
> Are you sure you want to continue connecting (yes/no)?
```

Enter `yes` to continue.

If you see an error message (such as the following), ensure that you have added the public key that you generated to GitHub. Repeat the key generation and and add the public key to GitHub again if necessary.

```
> Permission denied (publickey).
```

Afterward, you should see a success message, such as:

```
> Hi username! You've successfully authenticated, but GitHub does not
> provide shell access.
```

Now that your SPEDE Virtual Machine is connected to GitHub you are able to clone the phase0 repository and begin building and running code!

Part 3: Building and Running Code

Note: For this part, you will be required to copy and save your terminal output as part of the submission requirements.

For Part 3, you should copy and save your terminal output into a text file within the phase0 repository as part of your submission. Name this file `part3.txt`

Step 1: Cloning Repository

With your SPEDE Virtual Machine configured and connected to GitHub, you will need to clone your phase 0 repository (created when you originally accepted the assignment in GitHub classroom) to your virtual machine.

Within your GitHub repository page, select the green `Code` button. Ensure that you have selected `SSH` (not `https`) and then copy the provided URL.

Open a terminal on your SPEDE Virtual Machine and the clone the repository using the copied URL, such as:

```
git clone git@github.com:csus-grove-csccpe159-f24/phase0-username.git
```

Included in the repository are several directories:

- `1-hello-world`
- `2-debugging`

Within the `01-hello-world` directory, you will find the following files:

```
├── Makefile
└── src
    └── hello.c
```

The `Makefile` contains the rules to compile all source and generate a binary that can be loaded into the SPEDE Target system. You should not need to modify it, but you can customize your operating system name by changing the line that says `OS_NAME ?= MyOS` and replacing `MyOS` with a name of your choice. However, it must be an alphanumeric string with no spaces or special characters.

The `src` folder contains all C source code that will be incorporated into your operating system image.

`hello.c` contains a sample "Hello World" program that we will use to demonstrate the ability run compiled code.

```

```

```
#include <spede/stdio.h>

void main(void) {

    printf("Hello, world!\n");
    printf("Welcome to %s!\n", OS_NAME);
}
```

Step 2: Compiling Source Code

To compile, you can simply run `make` and you will see output indicating when it is done:

```
$ make
Done linking "build/MyOS.dli"
```

The file (`MyOS.dli`) that is generated is the compiled code that will be run on the SPEDE Target system.

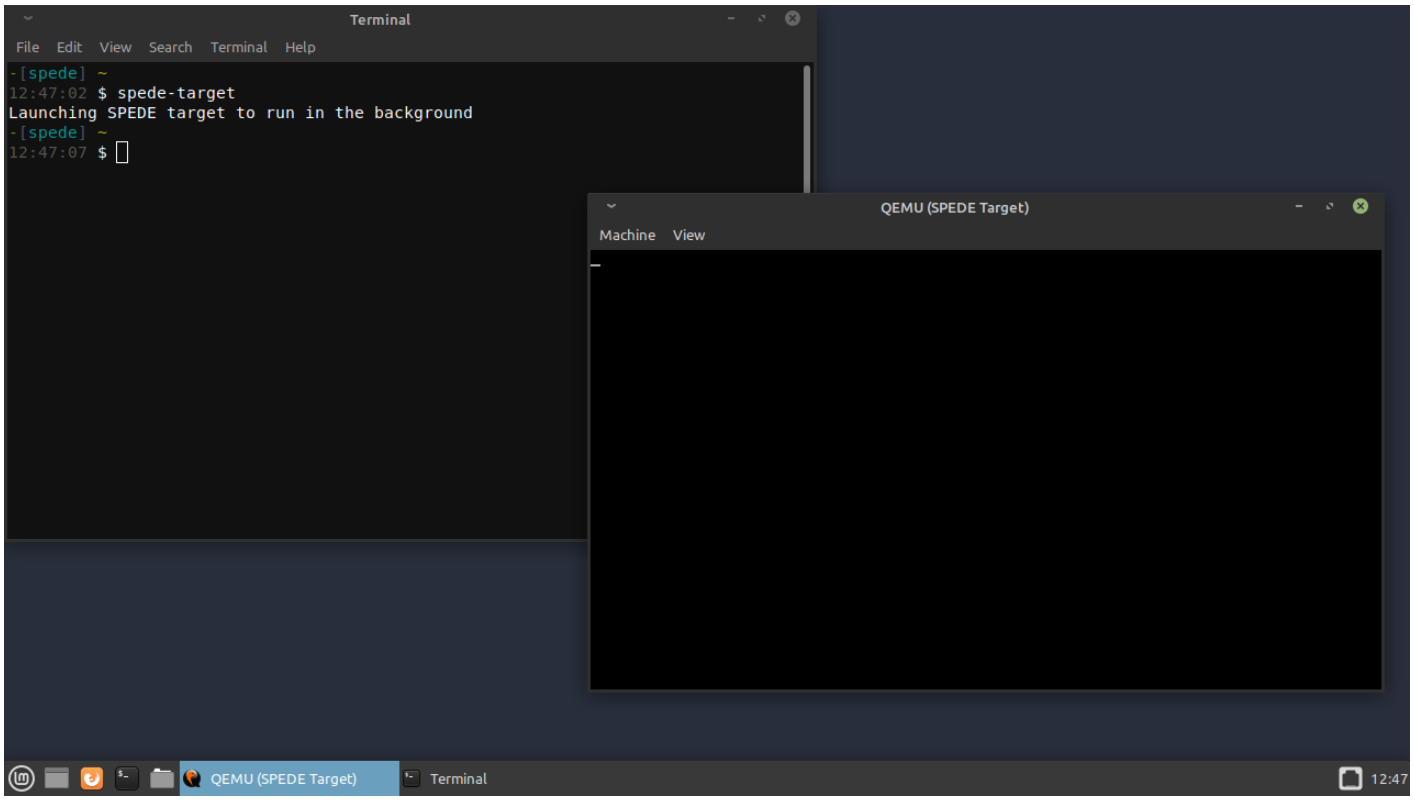
Step 3: Starting the SPEDE Target

Before you can run the code, you must start the target: `spede-target`

You will see a message that it is running in the background:

```
$ spede-target
Launching SPEDE target to run in the background
```

Additionally, you will see a new window open. This is the "video" or "console" output from the emulated computer system that will run our code.



If you click into the SPEDE Target window, it will take control of your keyboard and mouse. You regain control, press the following key sequence on your keyboard: `CTRL-ALT-G`.

If you close this window, the target will stop running.

If you would like to close or quit the SPEDE Target from the command line, you can run the `spede-target` command again with the `-q` parameter:

```
$ spede-target -q
The SPEDE Target has quit. Exiting...
```

Step 4: Running Code

With the target running, you can then run the code using the `spede-run` command in your terminal.

Note: Alternatively, you can run using the Makefile using `make run` which will build the image and launch `spede-run` for you.

```
$ spede-run
The SPEDE Target will be reset, are you sure? (y/n)
```

You will be prompted to confirm that you want to reset the SPEDE Target system. Simply enter `y` to continue.

The SPEDE Target will be reset and then your compiled code will be downloaded and executed immediately:

```
$ spede-run
The SPEDE Target will be reset, are you sure? (y/n) y
Resetting the SPEDE Target...
Downloading image 'build/MyOS.dli' to SPEDE Target via /dev/pts/3...
File type is 'ELF'
Total blocks to download:  0x8a  (128 bytes each)

Load Successful ; Code loaded at 0x0x101000
Executing the image 'build/MyOS.dli' on the SPEDE Target
$
```

Note that even though we were "printing" text, there was no output, on the console or within the SPEDE Target window.

This is because our operating system does not yet have a mechanism to provide input or output. The `printf` output in SPEDE will only be available when utilizing a debugging environment. In our case, this will be through GDB.

To run our code with GDB, we will use the same `spede-run` command and specify a command line argument to indicate we are debugging:

```
$ spede-run -d
```

Alternatively, you can run `make debug` which will perform the same action.

The SPEDE Target will be reset, code will be downloaded, but rather than being executed immediately, GDB will be launched and attached to the SPEDE Target system.

When GDB runs, it will automatically set a breakpoint at the `main` function so you have an opportunity to set breakpoints in GDB if needed for debugging or step through code as needed.

```
$ spede-run -d
The SPEDE Target will be reset, are you sure? (y/n) y
Resetting the SPEDE Target...
Downloading image 'build/MyOS.dli' to SPEDE Target via /dev/pts/3...
File type is 'ELF'
Total blocks to download:  0x8a  (128 bytes each)

Load Successful ; Code loaded at 0x0x101000
Executing the image 'build/MyOS.dli' on the SPEDE Target with GDB Enabled
Launching GDB...
Reading symbols from build/MyOS.dli...
Expanding full symbols from build/MyOS.dli...
Remote debugging using /dev/pts/3
0x0010255d in breakpoint ()
Temporary breakpoint 1 at 0x101183: file src/hello.c, line 5.

Temporary breakpoint 1, main () at src/hello.c:5
5       printf("Hello, world!\n");
```

When you are ready to continue, you can enter the `run` or `continue` command and your program will begin execution.

Notice that the `printf` output shows up on the debug console within GDB at this point

```
SPEDE GDB$ run
Hello, world!
Welcome to MyOS!
[Inferior 1 (Remote target) exited normally]
SPEDE GDB$
```

In this case, we can observe that GDB runs and the program exits as expected.

To re-run your code, you must quit GDB and run again:

```
SPEDE GDB$ quit
$ spede-run -d
```

Step 5: Save Console/Terminal Output

Before proceeding to Part 4, be sure to copy your terminal output and save it to a text file (`part3.txt`) to show that you have walked through the exercise to compile and run code.

Part 4: Debugging

Note: For this part, you will be required to copy and save your terminal output as part of the submission requirements.

For Part 4, you should copy and save your terminal output into a text file within the phase0 repository as part of your submission. Name this file `part4.txt`

In the `02-debugging` directory, a simple program is provided to give you practice writing code, compiling, and using the debugger.

```
#include <spede/stdio.h>

/**
 * Returns the length of a NULL terminated string
 * @param str pointer to the string
 * @return value indicating the length of the string
 */
int strlen(char *str) {
    // Implement me!
}

/**
 * Prints a string to the screen starting at 0, 0
 * @param str pointer to the string to print
 */
void puts(char *str) {
    if (str) {
        printf("NULL pointer!\n");
        return;
    }

    int len = strlen(str);

    for (int i = 0; i < len; i++) {
        *((char *) (0xB8000 + (i * 2))) = str[i];
    }
}

/**
 * Main function
 */
void main(void) {
    char buf[128] = {0};
    int year = 1970;

    *((char *) 0xB8000) = 'A';
```

```
    printf("Hello, world!\n");
    printf("Welcome to %s!\n", OS_NAME);

    puts("Hello, World!\n");

    snprintf(buf, sizeof(buf), "welcome to %s!\n", OS_NAME);
    printf("The buffer size is %d bytes\n", strlen(buf));
    puts(buf);

    printf("CPE/CSC 159 for Spring %d will be fun!\n", year);
}
```

Step 1: Implement the `strlen` function

In this exercise, you should implement the `strlen` function to calculate the length of a given string.

Once the `strlen` function is implemented, use the debugger to test and debug the code.

Step 2: Debug the program

Build the image and then run SPEDE with debugging:

```
$ make
Done linking "build/MyOS.dli"
$ spede-run -d
The SPEDE Target will be reset, are you sure? (y/n) y
Resetting the SPEDE Target...
Downloading image 'build/MyOS.dli' to SPEDE Target via /dev/pts/3...
File type is 'ELF'
Total blocks to download:  0x83  (128 bytes each)

Load Successful ; Code loaded at 0x0x101000
Executing the image 'build/MyOS.dli' on the SPEDE Target with GDB Enabled
Launching GDB...
Reading symbols from build/MyOS.dli...
Expanding full symbols from build/MyOS.dli...
Remote debugging using /dev/pts/3
0x00102659 in breakpoint ()
Temporary breakpoint 1 at 0x101200: file src/main.c, line 42.

Temporary breakpoint 1, main () at src/main.c:42
42      char buf[128] = {0};
SPEDE GDB$
```

Once you are in GDB, continue execution so the output can be observed:

```

```

```
SPEDE GDB$ continue
Continuing.
Hello, world!
Welcome to MyOS!
NULL pointer!
The buffer size is 17 bytes
NULL pointer!
CPE/CSC 159 for Spring 1970 will be fun!
[Inferior 1 (Remote target) exited normally]
```

If we look at the display of the SPEDE Target we should simply see the letter 'A'

Additionally, observe the output that seems to indicate something isn't correct:

```
NULL pointer!
```

Before we inspect the code, use the debugger.

```
$ spede-run -d
```

At the `SPEDE GDB$` prompt, set a breakpoint on the `puts` function where the "NULL pointer!" output is generated:

```
SPEDE GDB$ break puts
```

Confirm which breakpoints are set:

```
SPEDE GDB$ info break
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x001011a7 in puts at src/main.c:27
```

As we have confirmed this breakpoint, we can continue with execution:

```
SPEDE GDB$ continue
```

When tbe breakpoint is hit, the console will return to the GDB prompt and we can start debugging.

For context, list the source code we are debugging:

```
SPEDE GDB$ list
22  /**
23   * Prints a string to the screen starting at 0, 0
24   * @param str pointer to the string to print
25   */
26  void puts(char *str) {
27      if (str) {
28          printf("NULL pointer!\n");
29          return;
30      }
31
```

Since the output is indicating that the pointer is NULL, the pointer should be inspected. This can be done by printing the variable:

```
SPEDE GDB$ print str
$2 = 0x10423b "Hello, World!\n"
```

As we can see, the pointer is not NULL, so something is wrong with the code.

Lets step through the code and confirm which code path is taken:

```
```

```
SPEDE GDB$ step
28          printf("NULL pointer!\n");
SPEDE GDB$ step
NULL pointer!
29          return;
```

At this point, the code path is being taken even though the pointer is not NULL.

If you inspect the condition more closely, you'll see that there is a bug:

```
    if (str) {
```

Since the pointer value is not NULL, this condition will be true. Before the code is corrected, continue with execution until the next breakpoint at `puts` is hit again:

```
SPEDE GDB$ continue
Continuing.
The buffer size is 17 bytes

Breakpoint 2, puts (str=0x10dfc "welcome to MyOS!\n") at src/main.c:27
27      if (str) {
```

Instead of taking this code path, it can be jumped over:

```
SPEDE GDB$ jump +3
```

In this case, the next three lines are jumped over and execution continues and the program completes:

```
Continuing at 0x1011bf.
CPE/CSC 159 for Spring 1970 will be fun!
[Inferior 1 (Remote target) exited normally]
```

Observing the output of the SPEDE Target, we should now see a string printed:

Correct the source code and then run again to validate the output.

If the string is not output, then consider debugging the `strlen` function that you implemented.

For practice, perform some basic operations when debugging:

1. Set a breakpoint
2. List breakpoints
3. List a backtrace
4. List function arguments

Set a breakpoint for the `strlen` function:

```
SPEDE GDB$ break strlen
Breakpoint 2 at 0x101178: file src/main.c, line 9.
```

Confirm that a breakpoint is set for the `strlen` function:

```
SPEDE GDB$ info break
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x00101178 in strlen at src/main.c:9
SPEDE GDB$ continue
Continuing.
Hello, world!
Welcome to MyOS!
```

```
Breakpoint 2, strlen (str=0x10423b "Hello, World!\n") at src/main.c:9
```

List the backtrace when the breakpoint is hit:

```
SPEDE GDB$ backtrace
#0  strlen (str=0x10423b "Hello, World!\n") at src/main.c:9
#1  0x001011ca in puts (str=0x10423b "Hello, World!\n") at src/main.c:32
#2  0x00101271 in main () at src/main.c:51
```

This is especially useful when a function may be called from multiple places, such as in `main` and in `puts`) in this program.

List the function arguments:

```
SPEDE GDB$ info args
str = 0x10423b "Hello, World!\n"
```

The breakpoint for strlen can be cleared by deleting the breakpoint number associated with it:

```
SPEDE GDB$ info break
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x00101178 in strlen at src/main.c:9
    breakpoint already hit 1 time
```

Delete the breakpoint:

```
SPEDE GDB$ delete 2
```

Confirm that the breakpoint has been cleared:

```
SPEDE GDB$ info break
No breakpoints or watchpoints.
```

Lastly, looking more closely at the output, there are some items that could be fixed as well:

1. Display the current year instead of 1970
2. Print an uppercase 'W' to the SPEDE Target output

Rather than exiting, editing the code, recompiling, and running, this is a great opportunity to modify memory or variables within GDB directly.

First, set some breakpoints associated with the following lines in the main function:

```
    puts(buf);
```

followed by:

```
    printf("CPE/CSC 159 for Spring %d will be fun!\n", year);
```

Note: The line numbers in your implementation may be different based upon the edits you have made.

```
SPEDE GDB$ break main.c:55
Breakpoint 3 at 0x1012b5: file src/main.c, line 55.
SPEDE GDB$ break main.c:57
Breakpoint 4 at 0x1012c7: file src/main.c, line 57.
SPEDE GDB$ info break
Num     Type           Disp Enb Address    What
3       breakpoint     keep y   0x001012b5 in main at src/main.c:55
4       breakpoint     keep y   0x001012c7 in main at src/main.c:57
```

Continue execution until the first breakpoint is hit:

```
SPEDE GDB$ continue
Continuing.
The buffer size is 17 bytes

Breakpoint 3, main () at src/main.c:55
55      puts(buf);
```

Print the buffer:

```
SPEDE GDB$ print buf
$1 = "welcome to MyOS!\n", '\000' <repeats 110 times>
```

Modify the first byte:

```
SPEDE GDB$ set buf[0] = 'W'
SPEDE GDB$ print buf
$2 = "Welcome to MyOS!\n", '\000' <repeats 110 times>
```

Continue with execution:

```
SPEDE GDB$ continue
Continuing.

Breakpoint 4, main () at src/main.c:57
57      printf("CPE/CSC 159 for Spring %d will be fun!\n", year);
SPEDE GDB$ print year
$3 = 0x7b2
```

Observe the year value is in hexadecimal output. In this case it is not very easy to interpret, so print it as a decimal value:

```
SPEDE GDB$ print/d year
$4 = 1970
```

Try setting the current year and then confirm that it has been set:

```
SPEDE GDB$ set year=2023
SPEDE GDB$ print/d year
$5 = 8227
```

Notice that the value is incorrect. Print normally (hex) to observe what the value is:

```
SPEDE GDB$ print year
$6 = 0x2023
```

Set the value in decimal format by appending a `.` character to specify that it is a decimal number:

```
SPEDE GDB$ set year=2023.
SPEDE GDB$ print/d year
$7 = 2023
```

Continue to observe the value has been changed and printed:

```
SPEDE GDB$ continue
Continuing.
CPE/CSC 159 for Spring 2023 will be fun!
[Inferior 1 (Remote target) exited normally]
```

Step 3: Save Console/Terminal Output

Before proceeding to Part 5, be sure to copy your terminal output and save it to a text file (`part4.txt`) to show that you have walked through the debugging procedures.

Project Submission

Repositories

For phase 0, you will submit code individually using an individual repository. For every subsequent phase, teams will submit code using a team repository.

Branches

All source code must be submitted via GitHub using the `main` branch. You are welcome to utilize separate branches for your own development workflows but only source code on the `main` branch will be considered for grading.

Adding Files

Add all source files, which can be done using the `git add` command:

```
git add *
```

Commit your changes and include a commit message to describe the changes in the commit:

```
git commit
```

Push your changes to your repository:

```
git push
```

Tags

As git allows us to iteratively develop source code over a series of different commits, you will need to tag a specific commit to indicate the source code you are submitting to be graded.

All project phases must be submitted with a tag in the format of `phaseX.Y`, where `X` refers to the phase number (`0`, in this case) and `Y` refers to a submission number (usually `0`, but may increment if needing to submit code after a due date or for re-grading when permitted).

To tag a commit for Phase 0 submission, use the following commands to add a tag and push the tag to your repository.

Tag the commit:

```
```

```
git tag phase0.0
```

Push the newly added tag(s):

```
git push origin --tags
```

Grading Metrics

Grading encompasses meeting functional requirements as well as submission requirements.

While this project phase is intended to prepare you for the semester-long project, completing each part is necessary to ensure that you are adequately prepared and the grading will reflect the importance of completing each part accurately.

No credit will be given for the following:

- Source code is submitted past the due date
- Source code cannot be compiled as-is with the original/included Makefile
- Source code has not been materially modified from the base source code or provide template
- Source code has been plagiarized, duplicated, or otherwise demonstrated to not be originally developed

The following deductions may be made:

- 10 points for lack of an appropriate tag for the commit
- 25 points for each missing or incomplete requirement in the project phase
- up to 10 points per occurrence of programming errors

Examples of programming errors include (but are not limited to):

- Invalid use of pointers (such as dereferencing a NULL pointer or potentially NULL pointer)
- Invalid memory accesses (reading from or writing to memory that should not be accessed)
    - Specifically: Out-of-bounds indexing
- Failures to initialize variables or memory to known/default values
- Buffer overflows

Requirements Checklist

To help ensure that you have met each of the requirements for this phase, refer to this checklist below:

Part 1: Accessing GitHub Classroom

- [ ] Created a GitHub Account
- [ ] Accepted the Assignment in GitHub Classroom
- [ ] Linked GitHub Account to Student Name

Part 2: Setting up the Development Environment

- [ ] Downloaded and installed VirtualBox
- [ ] Downloaded and imported the SPEDE Virtual Machine
- [ ] Configured git with your name and email address
- [ ] Generated an SSH key for use with GitHub
- [ ] Added the SSH key to GitHub

Part 3: Building and Running Code

- [ ] Clone the Phase 0 repository
- [ ] Compiling Code
- [ ] Starting/Stopping the SPEDE Target
- [ ] Running compiled code on the SPEDE Target
- [ ] Running compiled code on the SPEDE Target with GDB
- [ ] Console output saved (`part3.txt`)

Part 4: Debugging Code

- [ ] Working with breakpoints
- [ ] Stepping through code
- [ ] Printing variables
- [ ] Modifying variables
- [ ] Implementation of the `strlen` function
- [ ] NULL pointer check fix in `puts` function
- [ ] Console output saved (`part4.txt`)

Project Phase 0 Submission

- [ ] Console output (`part3.txt` and `part4.txt`) committed to repository
- [ ] Source code (`02-debugging/src/main.c`) committed to repository
- [ ] Commit/submission tagged as `phase0.0`

Additional Resources

Working with Git

1. Pro Git Book
2. GitHub Git Cheat Sheet
3. Git Cheat Sheet

Working with GDB

1. GDB Quick Reference
2. Debugging with GDB

Support Resources

- CSUS IRT Laptop Checkout
- VirtualBox Forums

Virtual Machine Hosts

- VirtualBox
  - Windows Hosts
  - MacOS Hosts (*Intel Only*)
  - Linux Hosts
- VMware (*not required / not officially supported*)
  - VMware Workstation Player
  - VMware Fusion Player (*Intel Only*)
- Parallels Desktop (*not required / not officially supported*)
  - Parallels Desktop (*Intel Only*)