CSIE 5374 Assignment 1 (Due on 03/06 23:59)

In assignment 1, you are asked to create a simple shell program. A shell functions as a user program. Operating systems like Linux rely on shells to execute commands. For instance, the bash shell is an executable named bash. When you log into a Linux-based system, the shell's program, such as bash (located in `/bin/bash`), is executed.

Your task is to adhere to the following requirements in implementing your shell. Additionally, instructions are provided below for setting up the testing environment to evaluate your shell. Initially, you will boot Linux on an Armv8 virtual machine using QEMU, followed by running your shell program on Linux.

This assignment is inspired by and adapted from Homework 1 in W4118 Operating Systems offered at Columbia University.

Assignment 1 is an individual task. Each student must complete their own assignment.

Testing Linux

Below are the instructions for compiling and running Linux on an Arm-based VM hosted on QEMU, an open-source tool supporting virtualization. This environment will be utilized for testing your shell and will also be applicable for future assignments. Although not mandatory, it is advisable to set up the QEMU Linux environment at this point.

Before You Begin

- Prepare a functional Ubuntu environment where you are permitted to install any required software packages. It's recommended to install Ubuntu on a VM (using VMWare Workstation/Fusion or Parallel) or on a machine (laptop or lab server) over which you have full control. The tutorial provided below is based on Ubuntu 20.04 LTS. Ensure at least 50GB of free storage in your Ubuntu environment.
- Download the attachment `hw1.zip` from NTU Cool into your Ubuntu host. The attachment includes `run-vm.sh` for running VM and tester for testing your implementation.

Compiling QEMU

To set up the testing environment, first clone QEMU from the repository and switch to version v7.0.0:

```
# git clone https://gitlab.com/qemu-project/qemu.git
# cd qemu/
# git checkout tags/v7.0.0
```

Next, configure and compile QEMU from the source. During the "configure" command, you may encounter errors due to missing packages. Refer to online resources for resolving these errors.

```
# cd qemu
# ./configure --target-list=aarch64-softmmu --disable-werror
# sudo apt update && sudo apt install git build-essential libglib2.0-dev libfdt-dev
libpixman-1-dev zlib1g-dev ninja-build
# make -j4 (-j is to compile in parallel)
# sudo make install
```

Compiling Linux

Most likely, you are running Ubuntu on an x86 machine, so you'll need a cross-compiler to compile Linux binaries for your Arm-based machine. To install the cross-compiler for Arm on your Ubuntu machine, you can `apt install` the `gcc-aarch64-linux-gnu` package.

Once QEMU is installed, follow these steps to clone the mainline Linux source code.

```
# git clone --depth 1 --branch v5.15 https://github.com/torvalds/linux.git
# cd linux
```

Then, compile your Linux source.

```
# make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig
# make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -j4
```

Creating Virtual Disk Image

To boot your Linux VM, create a virtual disk image to store its file system.

First, download Ubuntu 20.04's file system binaries from here: Ubuntu 20.04 Cloud Image

Follow these instructions to create a Ubuntu 20.04 virtual disk image:

```
# qemu-img create -f raw cloud.img 20g
# mkfs.ext4 cloud.img
# sudo mount cloud.img /mnt
# sudo tar xvf ubuntu-20.04-server-cloudimg-arm64-root.tar.xz -C /mnt
# sync
# sudo touch /mnt/etc/cloud/cloud-init.disabled
```

Next, open `/mnt/etc/passwd`, and update the first line to disable root login password:

```
root::0:0:root:/root:/bin/bash
```

Finally, `unmount` the file system image from your Ubuntu host.

```
# sudo umount /mnt
```

Running Linux VM

Once the compilation of the Linux kernel is complete, it's time to test your newly compiled Linux binary. Utilize the virtual disk image `cloud.img` created earlier.

You can run the VM using the script `run-vm.sh`.

```
./run-vm.sh -k $PATH_TO_IMAGE -i $PATH_TO_YOUR_cloud.img
```

Assuming you placed your Linux source in `/home/ubuntu/linux`, your `PATH_TO_IMAGE` would be `/home/ubuntu/linux/arch/arm64/boot/Image`.

Your newly compiled Linux binary should now be running. The output you observe is from the virtual serial port.

Writing a Simple Shell in C (100%)

The requirements are listed below.

- Your shell should receive user commands from standard input (`stdin`), parse the input, and execute the command. It should read one line at a time and continuously wait for user input. The shell should display a prompt `$`, with no additional spaces or characters. An example of a shell prompt and command is provided below.

```
$/bin/ls -al
```

- For delimiters, assume command line arguments are separated by whitespace characters. Do **NOT** handle other special characters supported by other shells, such as quotation marks, backslashes, ampersands, etc.
- The use of the `system` function in C, which invokes the system's shell, is **NOT** allowed.
- Set the maximum number of command line arguments to `_POSIX_ARG_MAX`. Your shell should handle input commands of any length.
- Your shell should support two types of commands: built-in and references to executables.
    - Built-in commands: `cd`, `exit`, and `history`.
        - `cd [dir]`: This command changes the current working directory of your shell. The cd command takes a single argument: the target directory to change to. Only absolute paths need to be handled.
        - `exit`: This command exits your shell.
        - `history [-c] [n]`: This command prints out the command history, including the current history command. Detailed specifications are provided below.
    - Executables: Execute the command by first invoking `fork()` then `exec()`. The executables will be specified with absolute paths.

- Your shell should be capable of supporting Command Pipelines. When two commands are separated by a `|` character, the `stdout` of the preceding command should be redirected to the `stdin` of the subsequent command. Your shell should also have the capability to handle multiple pipes.
    - Your shell does not need to support other duplication operators.
- The user can terminate the shell by pressing ***ctrl+c***. Implement a proper signal handler in your shell to handle this event.
- Check the return values of all functions utilizing system resources. Do not assume all requests for memory will succeed and all writes to a file will occur correctly. Handle errors properly. Print error messages using `fprintf` to `stderr` following one of the formats provided below.

```
"error: %s\n", strerror(errno)
"error: %s\n", "your error message"
```

- Free up resources and memory upon exiting or termination of your shell. Ensure there are no memory leaks. Although many modern OSes free up resources such as memory of a process upon its termination, do not assume the OS always does that for you in your program.

To support the `history` command, your shell should display all command history with their respective command numbers if the total number of commands is not greater than 10. If the total exceeds 10 commands, your shell should display the last 10 command history with their command numbers. The command numbers should be right-aligned to the 5th character, and there should be two spaces after the command number. In our test cases, the total will not exceed 99999 commands. For instance, if the total number of commands (including the current `history` command) is 8, the `history` command output should appear as follows:

```
$history
    1  /bin/ls
    2  /bin/pwd
       ... (3-6)
    7  cd /
    8  history
```

If the total command number (including the current `history` command) is 1500, the `history` command should print:

```
$history
 1491  /bin/ls
 1492  /bin/pwd
       ... (1493-1498)
 1499  cd /
 1500  history
```

To support the `history -c` command, your shell should simply clear all the command history:

```
$history -c
$history
    1  history
```

To support the `history n` command, where `n` is a positive integer, your shell should print out the last *n* command history, or all command history if the total command number is less than *n*. For `n` larger than 10, treat it as 10:

```
$history 2
 1499  cd /
 1500  history 2
$history 200000
 1492  /bin/ls
 1493  /bin/pwd
       ... (1494-1499)
 1500  history 2
 1501  history 200000
$ history 3
 1500  history 2
 1501  history 200000
 1502  history 3
$ history 3
 1500  history 2
 1501  history 200000
 1502  history 3
```

Note that if a command is the same as the previous command, it should not be included as a new entry in the command history. In our test cases, commands will not have leading or trailing spaces. Additionally, there will be exactly one space character between command arguments. Therefore, you are not required to differentiate between commands such as `history__3` and `_history_3_` (where the `_` characters represent space characters). Furthermore, the test cases will not contain any tab characters.

Additional Requirements

- Utilize a makefile to control the compilation of your code. The Makefile should include at least a default target that compiles all programs. Ensure to use the `-Wall` flag to display and properly handle compile warnings. Make sure there are no warnings present, failure to address warnings will result in point deductions.
- Your shell will be tested in the Arm-based VM hosted on QEMU, as built using the instructions provided above. Ensure that your shell operates effectively in this environment.

Extra Tips

Configuring SSH

Configure SSH to enable logging into the machine as a root user without a password. Upon initially logging into the Linux VM via the serial console, configure SSH for your machine by executing the following command:

```
dpkg-reconfigure openssh-server
```

If root login needs to be enabled, set `PermitRootLogin yes`. Additionally, set up SSH key authentication for your root user by executing:

```
ssh-keygen
```

Create a new file `/root/.ssh/authorized_keys`, then copy your Ubuntu host's public key (from `.ssh/id_rsa.pub`) and paste it into `authorized_keys`.

After completing these steps, you should be able to log in using SSH from the Ubuntu host.

Connect to Linux VM via SSH

The `run-vm.sh` script supports port forwarding from the host (running Ubuntu), enabling SSH access to the Linux VM from your host Ubuntu environment. To enable SSH, follow these steps in your VM's shell:

```
# dhclient
```

Then, open another terminal session on your Ubuntu host and SSH to the Linux VM:

```
# ssh root@localhost -p 2222
```

Tip: To copy files to your Linux VM, either use `scp` from your Ubuntu host or `mount` the virtual disk image on your Ubuntu host first, then proceed with the copying. For the latter method, follow these steps:

```
# mount cloud.img $YOUR_MNT_DEST_DIR
# cp $FILES to $YOUR_MNT_DEST_DIR
# umount $YOUR_MNT_DEST_DIR
```

Homework submission

You should submit the assignment via NTU Cool.

Submission Format

For homework submission, name the shell program as `cs5374_sh`. You are required to submit the source code, a `Makefile`, and a `README` file documenting your files and code. Compress all the files into `hw1.zip` and upload the zip file to NTU Cool.

Grading criteria

We will verify if your Makefile functions properly and generates a valid `cs5374_sh` binary. If your shell fails to compile, you will receive zero points.

A testing tool is provided for testing your shell, allowing you to create additional test cases to evaluate your code. Ensure that your program works with the provided testing tool, as it will be used for grading. Note that the testing tool is not compatible with Python3. Please use **Python2** to execute it. Apart from the basic tests included in the testing tool, we will conduct more extensive testing of your program. Failure of the testing tool to work with your shell program will result in zero points.