CSIE 5374 Assignment 1 (Due on 03/02 23:59)

In assignment 1, you are asked to write a simple shell. A shell itself is just an user program. Operating systems like Linux relies on shells to run programs. For instance, the bash shell is an executable called ***bash***. When you log into a computer running Linux based systems, the shell's program, such as bash (located in /bin/bash) gets executed.

You should follow the requirements listed below to implement your shell. You are also asked to modify Linux kernel to support a new system call. Your shell should run on top of the modified Linux kernel to invoke the new system call. You can follow the instructions below to setup the environment for testing your shell and the modified kernel. Specifically, you will first boot the modified Linux in an Armv8 virtual machine using QEMU, then run your shell program on the Linux.

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

Assignment 1 is per person. Every student should work on their own assignment.

1. Add new system call to Linux (40%)

In this part of the assignment, you are asked add a new system call **sys_get_cpuid** to Linux, then asked to invoke **sys_get_cpuid** from your shell program.

The **sys_get_cpuid** system call takes no argument, but returns a long value. Specifically, your system call handler should do the following:

```
long sys_get_cpuid(void) {
  // return the current CPU ID when this line of code in the system call handler is
executed
}
```

You should first allocate **436** as the system call number for sys_get_cpuid, and add a handler in the guest Linux kernel for handling the system call. **You do not need to worry about preemption in this assigment.**

For getting the current CPU ID, take a look at the function **raw_smp_processor_id()** defined in here for Linux v5.4.

Testing Linux

The following provides the instructions for your to compile and run your modified Linux on an Arm based VM hosted on QEMU, an open source tool that supports virtualization.

Before you start

- You should prepare a working Ubuntu environment, which you are allowed to install any software packages as you wished. We recommend installing Ubuntu on a VM (on VMWare workstation/fusion or parallel), or on a bare-metal machine (laptop or lab server) that you have full control. The tutorial provided as follows is based on Ubuntu 20.04 LTS. Please make sure to have at least 50GB free storage in your Ubuntu environment.
- Download the attachment `files.zip` from NTU Cool into your Ubuntu host. There are 2 files used in this assignment:
    - defconfig
    - run-vm.sh

Compile QEMU

To setup the environment for testing, first clone QEMU from the repo and checkout to version v6.1.0:

```
# git clone https://gitlab.com/qemu-project/qemu.git
# cd qemu/
# git checkout tags/v6.1.0
```

Then configure and compile QEMU from the source. You may run into some errors when you do the "configure" command, this is because you have missing packages. Google will be your friend for fixing the 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
```

Compile Linux

Chances are that you are running Ubuntu on a x86 machine, so you will need a cross compiler to compile the KVM binaries for your Arm based machine. To install the cross compiler for Arm on your Ubuntu machine, you could apt install the "gcc-aarch64-linux-gnu" package.

Once you have QEMU installed, do the following to clone the mainline KVM source code.

```
# git clone git@github.com:torvalds/linux.git
# cd linux; git checkout tags/v5.4
```

Next, compile your Linux source. You can find the config file in the files/ directory.

```
# cp $YOUR_PATH/defconfig .config
# yes "" | make oldconfig ARCH=arm64
# make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -j4
```

Creating virtual disk image

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

First, download Ubuntu 18.04's file system binaries from here: https://cloud-images.ubuntu.com/releases/bionic/release/ubuntu-18.04-server-cloudimg-arm64-root.tar.xz

You can then follow the instructions below a ubuntu 18.04 virtual disk image:

```
# qemu-img create -f raw cloud.img 20g
# mkfs.ext4 cloud.img
# sudo mount cloud.img /mnt
# sudo tar xvf ubuntu-18.04-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 the following 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
```

Run Linux VM

After the compilation of the Linux kernel is complete, it is now time to test your newly compiled Linux binary. You can use the virtual disk image ***cloud.img*** that you just created above.

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 put your Linux source in /home/ubuntu/, your PATH_TO_IMAGE is equal to

```
/home/ubuntu/arch/arm64/boot/Image
```

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

2. Write a simple shell in C (60%)

The requirements are listed below.

- Your shell should get user commands from standard input (***stdin***), parse the input, then run the command. Your shell should read one line at a time. The shell should wait continuously for user input. The shell should display a prompt "$", with no spaces and characters. Example for a shell prompt and command is shown below.

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

- For delimiters, your should assume command line arguments are separated by the whitespace characters. ***DO NOT*** do anything special for the other special characters supported by other shells, such astquotation marks, backslashes, ampersands, etc.
- You are ***NOT*** allowed to use the ***system*** function in C, which it will invoke the system's shell.
- You should 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 an executable.
    - built-in commands: **cd**, **exit**, and **getcpu**.
        - **exit**: the command exits your shell.
        - **cd [dir]**: the command changes the current working directory of your shell. The cd command takes a single argument: the target directory to change to.
        - **getcpu**: the command should invoke the ***get_cpuid*** system you added in part 1, and output the return value to stdout with a new line.
    - executables: you should execute the command by first invoking fork() then exec().
- The user can terminate the shell by ***ctrl+c***. You should wirte a proper signal handler in your shell to handle the input event.
- Check the return values of all functions utilizing system resources. Do not simply assume all requests for memory will succeed and all writes to a file will occur correctly. Your code should handle errors properly. Your program should handle errors gracefully.

  In Linux, a system call will return -1 in the case of an error. If a function call sets the ***errno*** variable (see the function's man page to find out if it does), you should use the first string as format shown below. The second format below is for when errno is not set, in which case you may provide any error text message you like on a single line.

  The error messages should be printed using ***fprintf*** to ***stderr*** using either one of the formats below.

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

"error: %s\n", "your error message"
```

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

Additional Requirements

- All of your code should follow Linux's coding style rules:
  - Tab size: 8 spaces.
  - Do not have more that 3 levels of indentations (unless the function is extremely simple).
  - Do not have lines that goes after the 80th column (with rare exceptions).
  - To comment your code, use /* ... */ and not // ...
  - Follow the Linux Kernel Coding Style. Use checkpatch, a script which checks coding style rules. You can use the following command against your .C, .h. and Makefile:

```
checkpatch.pl --terse --file --no-tree --ignore SPDX_LICENSE_TAG $YOURFILE
```

  We will use checkpatch to check your code when we grade your assignment, so you will lose points if your code does not satisfy the rules.
- Use a makefile to control the compilation of your code. The makefile should have at least a default target that builds all programs. Use **-Wall** to ensure compile warnings are displayed and properly handled. You should ensure no warnings exist, or you will lose points.

Extra Tips

Configure ssh

You can then login to the machine as a root user without a password. When you log into the Linux VM for the first time via the serial console, do the following to configure ssh for your machine:

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

You do need to reconfigure ssh to enable root login, you can set "PermitRootLogin yes". You will also need to setup the ssh key authentication for your root user by doing the following:

```
ssh-keygen
```

Then you should create a new file ***/root/.ssh/authorized_keys***, then copy your Ubuntu host's public key (from .ssh/id_rsa.pub) and paste to authorized_keys.

After this, you should be able to login using ssh from the Ubuntu host.

Connect to Linux VM via ssh

Furthermore, run-vm.sh supports port forwarding from the host (running UBUNTU), so you can ssh to the Linux VM from your host Ubuntu environment. To enable ssh, do the following in your VM's shell:

```
# dhclient
```

Then open another terminal session on your **Ubuntu host**, and do the following to ssh to the Linux VM:

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

TIPS: to copy files to your Linux VM, you can either do so via ***scp*** from your Ubuntu, or first mount the virtual disk image on your Ubuntu host, then do the copying. For the latter, you can do:

```
# 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**. For the shell, you will be required to submit source code, a Makefile, a README file documenting your files and code. You are also required to submit your kernel patch for Linux v5.4 to support the new system call. Compress all the files in hw1.zip, and upload the zip file to NTU Cool.

Grading criteria

We will check if your Makefile works and generate a proper cs5374_sh binary. We will also apply your submitted kernel patch to Linux v5.4. If your shell or Linux fails to compile, you will get **Zero** points.

A testing tool is provided for testing your shell. The tool allows to write additional test cases yourself to test your code. Please make sure your program works with the provided testing tool. We will use the tool for grading. Other than the basic tests included in the testing tool, we will test your program more extensively. You will get **Zero** points if the testing tool does not work for your shell program.