EEE3535-01 **Fall 2022**

Assignment 3: Shell

Due: Sunday, Nov. 6, 2022, 11:59PM

1 Introduction

- The objective of this assignment is to implement a simple shell in xv6-riscv, which we will call it "Yonsei shell" or simply **ysh**.
- A shell in Linux or Unix is a program that provides users with interfaces to the OS. It receives inputs from a user and executes the commands.
- When you log into a Linux (e.g., Ubuntu) or Unix (e.g., Mac) terminal, you should see something similar to the following trailed by a "`$`" sign waiting for your input. In Mac OS, it may display a "`%`" sign instead of `$`.

```
username@ubuntu $
```

- Try an `ls` command in the Ubuntu or Mac terminal as follows. This command displays a list of files in the `/usr/bin/` directory, which is similar to `C:\Program Files\` directory in Windows where program executables (i.e., `*.exe`) are located at.

```
username@ubuntu $ ls /usr/bin
alias       awk         bg          bison       bzip2           c++
cc          cpp         curl        env         find            g++
gcc         git         grep        ...
```

- When you type `ls /usr/bin/`, the shell should do something similar to the following. It creates a child process using `fork()` and makes the child execute the command via `exec()`.

```
int pid = fork();
if(pid > 0) {
    // The parent process (i.e., shell) waits for the child to finish.
    wait(0);
}
else if(pid == 0) {
    // The child process executes the command.
    char *argv[3] = {"ls", "/usr/bin/", 0};
    exec(argv[0], argv);
}
```

- The shell repeats such `fork()` and `exec()` executions in a while loop until an `exit` command is put to terminate the shell.

```
username@ubuntu $ exit
```

- There are many shell programs available in Linux. Common shell programs are bash, csh, ksh, tsh, zsh, etc. Typing the following command will show you what kind of shell you are in.

```
username@ubuntu $ echo $SHELL
/bin/bash
```

- Since the shell itself is a program, you can make it execute another shell as a child process. The following example makes the current shell run a bash shell.

```
username@ubuntu $ bash
bash $
```

- To start this assignment, go to the `xv6-riscv/` directory. Download `shell.sh`, and execute it to update xv6-riscv. The script will make a few changes to xv6-riscv for the assignment.

```
username@ubuntu $ cd xv6-riscv/
username@ubuntu $ wget https://icsl.yonsei.ac.kr/wp-content/uploads/shell.sh
username@ubuntu $ chmod +x shell.sh
username@ubuntu $ ./shell.sh
```

- To check if the update is successful, build xv6-riscv and try executing `ysh`. If you can reproduce the following example, the update is done. Terminate xv6-riscv by pressing `ctrl+a` and then `x`.

```
username@ubuntu $ make clean

...

username@ubuntu $ make qemu

...

qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1
-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

EEE3535 Operating Systems: booting xv6-riscv kernel
EEE3535 Operating Systems: starting sh
$ ysh
EEE3535 Operating Systems: starting ysh
$ exit
EEE3535 Operating Systems: closing ysh
$ QEMU: Terminated
```

2 Implementation

- The only working commands in the `ysh` skeleton code are `cd` and `exit`. These do not invoke program executions since they are reserved keywords in the shell, which are handled by the main process (i.e., `ysh` itself instead of forking a child).

- The following shows a part of the `ysh` skeleton code in `user/ysh.c`.

```
/* user/ysh.c */

...

int main(int argc, char **argv) {

    ...

    // Read and run input commands.
    while((cmd = readcmd(buf)) && runcmd(cmd)) ;

    ...
}

// Run a command.
int runcmd(char *cmd) {
    if(!*cmd) { return 1; }                    // Empty command

    // Skip leading white space(s).
    while(*cmd == ' ') { cmd++; }
```

```
    // Remove trailing white space(s).
    for(char *c = cmd+strlen(cmd)-1; *c == ' '; c--) { *c = 0; }

    if(!strcmp(cmd, "exit")) { return 0; }      // exit command
    else if(!strncmp(cmd, "cd ", 3)) {          // cd command
        if(chdir(cmd+3) < 0) { fprintf(2, "Cannot cd %s\n", cmd+3); }
    }
    else {
        // EEE3535-01 Operating Systems
        // Assignment 3: Shell
    }
    return 1;
}
```

- The `main()` of `ysh` gets a user input from `readcmd()` and executes the command through `runcmd()`. The `readcmd()` function is complete in the skeleton code. Do not modify `readcmd()` and `main()` functions.
- The basic form of `runcmd()` is provided at the bottom of `user/ysh.c`. This function takes a `char` array, `cmd`, as an input argument.
- `runcmd()` first checks if `cmd` is not an empty string. An empty string immediately returns 1, which indicates that `runcmd()` is successfully done.
- For a non-empty string, `runcmd()` first removes leading and trailing white spaces. Then, it checks if the command is `"exit"` or starts with `"cd "`. An `exit` command returns 0, and a `cd` command calls a change-directory syscall, `chdir()`.
- If the command is not `"exit"` nor starts with `"cd "`, it must be a regular command, which executes program(s).
- A successful run of `runcmd()` returns 1 so that the `while()` loop in `main()` repeats indefinitely until an `exit` command returns 0 and breaks the loop.
- The body of `else { ... }` near the end of `runcmd()` is what you have to implement for this assignment. This is the only part that you will have to work on. This assignment does not require changes in the xv6-riscv kernel or other user files.
- You are allowed to add functions or structs to `ysh.c` or introduce additional variables in `runcmd()`. But, do not modify `readcmd()` and `main()` functions.
- **Hint:** Every shell program uses a different parsing scheme for user inputs. Regardless, simple commands are executed basically in the same way because there cannot be multiple interpretations of them. However, when complex commands are put together using pipes (`|`), in series (`;`), and/or in the background (`&`), different shells may react differently depending on how they prioritize the options. In short, there is not only one solution to implement a shell. The following describes one of the possible ways to implement the `else` body of `runcmd()`.
  - Suppose a user input is given as the following example. It first executes `forktest`. Commands on the right side of the `;` sign are executed after the left-side command (i.e., `forktest`) is done. After `forktest`, three piped commands (i.e., `grep xv6 README | grep Unix | wc`) are executed, followed by two piped commands (i.e., `whoami | grep ID`). An `&` sign at the end indicates that the preceding commands are executed in the background. An expected output is shown below. Since the `&` sign makes the main process of `ysh` do not wait for children processes to finish, a `$` sign requesting the next user input is prematurely printed while the children are progressing. Pressing the enter key will display a `$` sign.

    ```
    $ forktest; grep xv6 README | grep Unix | wc; whoami | grep ID &
    $ fork test
    fork test OK
    1 11 71
    Student ID:   2022143535

    $
    ```

- What you need to figure out is how the shell should parse the character string (i.e., `cmd` of `runcmd()`) and execute right commands in appropriate orders.
- The following describes the sequence of a working procedure. Note that this is not the only way of implementing the shell algorithm.
  1. In the `else` body of `runcmd()`, check if the last character of `cmd` is an `&` sign. If so, replace this character with 0 so that `cmd` is null-terminated without the `&` sign at the end, labeled as ① below. Fork a child process via `fork()`, and let the child recursively process `runcmd(cmd)`. The parent process does not wait for the child.
     `forktest; grep xv6 README | grep Unix | wc; whoami | grep ID &`
     ①
  2. If the last character of `cmd` is not `&`, then check if `cmd` contains a `;` sign. `char* s = strchr(cmd, ';')` should return a `char` pointer to the first `;` sign in `cmd`. If `cmd` does not have one, `s` gets a null pointer. Replacing the first `;` character with 0 can separate the original `cmd` into two parts labeled as ① and ② below. Fork a child, and let the child recursively process `runcmd(cmd)` for part ①. The parent process also recursively calls `runcmd(s+1)` for part ②, which will again be divided into two parts since it again contains a `;` sign.
     `forktest; grep xv6 README | grep Unix | wc; whoami | grep ID`
     ① ②
  3. If `cmd` is not `&`-ended nor has a `;` sign, then check if it contains a `|` sign. The first `|` sign in `cmd` can be similarly found as `char* p = strchr(cmd, '|')`. Replacing the first `|` character with 0 can separate `cmd` into two parts labeled as ① and ② below. Fork a child, and make the parent process wait for the child. The child creates a pipe and forks again to create a grandchild. Let the grandchild execute the write end of the pipe (i.e., part ①) by calling `runcmd(cmd)`, and the child process runs the read end of the pipe (i.e., part ②) by calling `runcmd(p+1)`. Since the second part again contains a pipe, the same procedure will be conducted once again in the child process.
     `grep xv6 README | grep Unix | wc`
     ① ②
  4. If `cmd` does not fall into any cases above, this must be a single command. Parse `cmd` into an `argv[]` array, fork a child, and let the child execute the command by calling `exec(argv[0], argv)`.
     `grep xv6 README`
     ① ② ③
- Although there are many other functionalities commonly used in shells, the simple `ysh` will support only the four cases explained above, i) background (`&`), ii) series (`;`), iii) pipe (`|`), and iv) single commands.

3 Validation

- Your assignment will be graded based on the following seven cases to validate the implementation. The shown cases are not the exact and only ones that will be on test. Your code should be able to handle other similar common-sense situations but will not be tortured by insanely irrational commands to make it crash on purpose.
  1. **Single commands**: The shell should be able to execute a command with an arbitrary number of inputs. The maximum number of args (i.e., length of `argv[]`) that a command can have is 16 (i.e., `#define max_args 16`). This should be large enough to cover most user inputs.

```
$ ysh
EEE3535 Operating Systems: starting ysh

$ grep xv6 README
xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
Version 6 (v6).  xv6 loosely follows the structure and style of v6,
xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
The code in the files that constitute xv6 is
(kaashoek,rtm@mit.edu).  The main purpose of xv6 is as a teaching
```

```
$ ls init
init            2 7 33000

$ forktest
fork test
fork test OK
```

2. **Two piped commands**: Connect two commands using a pipe such that the command on the left side of the pipe passes its output to the input of the right-side command.

```
$ grep xv6 README | grep Unix
xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix

$ ls | grep c
cat             2 3 33656
echo            2 4 32536
wc              2 17 34576
console         3 21 0
```

3. **Two serial commands**: If two commands are put together using a `;` sign in the middle, the shell executes the first command on the left side of the semicolon and then executes the second command on the right after the first one is done.

```
$ stressfs; ls
stressfs starting
write 0
write 1
write 2
write 3
write 4
read
read
read
read
read
.               1 1 1024
..              1 1 1024
README          2 2 2305
cat             2 3 33656
echo            2 4 32536
forktest        2 5 16632
grep            2 6 37096
init            2 7 33000
kill            2 8 32448
ln              2 9 32272
ls              2 10 35592
mkdir           2 11 32512
rm              2 12 32504
sh              2 13 55080
stressfs        2 14 33392
usertests       2 15 181136
grind           2 16 48256
wc              2 17 34576
whoami          2 18 32064
ysh             2 19 38192
zombie          2 20 31872
console         3 21 0
stressfs0       2 22 10240
stressfs1       2 23 10240
stressfs2       2 24 10240
stressfs3       2 25 10240
```

```
stressfs4      2 26 10240

$ rm stressfs0; ls stressfs0
ls: cannot open stressfs0
```

4. **A command in the background:** Placing an `&` sign at the end of a command makes the shell execute the command in the background. The `grind` program slowly and repeatedly prints `BABABABA` ···, which never ends. While `grind` is running in the background, `ysh` should be able to process other user inputs.

```
$ grind &
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2305
cat            2 3 33656
echo           2 4 32536
forktest       2 5 16632
grep           2 6 37096
init           2 7 33000
kill           2 8 32448
ln             2 9 32272
ls             2 10 35592
mkdir          2 11 32512
rm             2 12 32504
sh             2 13 55080
stressfs       2 14 33392
usertests      2 15 181136
grind          2 16 48256
wc             2 17 34576
whoami         2 18 32064
ysh            2 19 38192
zombie         2 20 31872
console        3 21 0
grindir        1 22 32
b              1 25 48
a              1 24 48
$ BABABABABABABABABABABABABABABABABABABABA ... // grind never ends.
```

5. **A series of piped commands:** If the shell can handle two piped commands, it can also process a series of piped commands as well.

```
$ ls | grep c | grep e
echo           2 4 32536
console        3 21 0

$ stressfs | grep read | cat | wc
$ 5 5 25
```

6. **A series of serial commands:** Similarly, the shell should be able to handle multiple serial commands.

```
$ echo EEE3535; forktest; grep Unix README; ls mkdir; stressfs; whoami
EEE3535
fork test
fork test OK
xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
mkdir          2 11 32512
stressfs starting
write 0
write 1
write 2
write 3
write 4
```

```
read
read
read
read
read
Student ID:   2022143535
Student name: Operating Systems
```

7. **Complex commands with pipes and series:** The shell can also mix up pipes and serial commands in a single command line.

```
$ forktest; grep xv6 README | grep Unix | wc; whoami | grep ID
fork test
fork test OK
1 11 71
Student ID:   2022143535
```

4 Submission

- When the assignment is done, execute the `tar.sh` script inside the `xv6-riscv/` directory. This will compress the `xv6-riscv/` directory and create a tar file named after your student ID (e.g., `2022143535`).

```
$ ./tar.sh
$ ls
2022143535.tar  Makefile    fs.img      mkfs        user
LICENSE         README      kernel      tar.sh
```

- Upload the tar file (e.g., `2022143535.tar`) on LearnUs. Do not rename the file.

5 Grading Rules

- The following is the general guideline for grading. A 30-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change if needed, and a grader may add a few extra rules for fair evaluation of students' efforts if necessary.

  -5 points: A submitted tar file is renamed to include some redundant tags such as a student name, `hw3`, etc.

  -5 points: A submitted code does not have a sufficient amount of comments. Comments in the skeleton code do not count. You must make an effort to clearly explain what each part of your code intends to do.

  -4 points each: `ysh` fails to produce a correct result for each test case. The last case of **Complex commands with pipes and series** accounts for 6 points.

  -30 points: No or late submissions.

  Final grade = F: A submitted tar file is not from student's own implementation but copied from someone else. All students involved in the incidents will be given "F" for final grades.

  Final grade = F: A code is intentionally tweaked and faked to deceive a grader as if some efforts are made. Such an attempt is regarded as an unethical behavior and thus will be seriously penalized.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect for any reasons, feel free to discuss your concerns with the TA. Always be courteous when contacting the TA. In case no agreements are made between you and the TA, elevate the case to the instructor to review your assignment. Refer to the course website for the contact information of the TA and instructor: `https://icsl.yonsei.ac.kr/eee3535`.

- Arguing for partial credits with no valid reasons will be regarded as a cheating attempt, and such a student will lose all scores of the assignment.