CS302 OS Week4 Assignment - Report

Name: 刘仁杰
SID: 11911808

1. Read Chapter 2 of "Three Easy Pieces"

1. “three easy pieces” of operating systems

- Virtualization: transform a physical resource (cpu, memory, disk, ...) into a more general easy-to-use virtual form, which further allows many programs to run and share physical resources concurrently.
- Concurrency: working on many things concurrently in the same program or by multi-threaded programs without incurring incorrect results.
- Persistence: How the os take control of file system to store data persistently, access device, enhance performance with typical mechanisms and recover data from crash for failure.

2. How do these “three easy pieces” map to the chapters in the “dinosaur book”?

- Virtualization: Chapter 5,10
- Concurrency: Chapter 4,6,7,8
- Persistence: Chapter 12,13,14,15

2. What happens during context switch in detail?

- Process A is running
- Interrupted by timer
- Hardware saves A's registers onto its kernel stack and switch to the kernel mode
- Trap handler call `switch()` routine which saves current register values into the process structure of A and restores the registers of Process B from its process structure entry.
- Change the stack pointer to use B’s kernel stack
- OS returns-from-trap, which restores B’s registers from B's kernel stack and starts running Process B

3. Read slides “L03 Processes I” and “L04 Processes II” and answer questions:

(1) Explain what happens when the kernel handles the fork() system call

- When fork() system call is invoked, the program switch from user mode to kernel mode (save and restore context)
- In OS kernel, new address space is created for the child process, and copy the kernel space (PCB) of parent process to the newly created address space
- OS kernel does the kernel update of the child kernel space, which includes PID, running time, and a pointer to its parent
- OS kernel will also create a pointer in parent process's kernel space which points to the forked child
- Add the child process to the task list
- User space of the child process is also copied and updated
- fork() is completed, return value is set in both the parent process and newly forked child process respectively to be the child process PID and 0
- CPU scheduler decide which process to be executed next

- If parent process is to be executed next, the child PID is returned, program switch back to user mode and the process is continued
- If child process is to be executed next, there will be a context switch from parent process to child process which involves PCB reload, pc rebase and switch from kernel mode to user mode, starting running process B

(2) Explain what happens when the kernel handles the exit() system call

- When exit() system call is invoked, the program switch from user mode to kernel mode
- The kernel frees all allocated memory in kernel space, except the PID and the list of opened files are closed
- The kernel frees everything on the user-space memory about the concerned process, including program code and allocated memory
- The kernel sends a SIGCHLD signal to its parent process which notifies the child process has now terminated
- If its parent process invoked wait() system call
    - the kernel will register a signal handling routine for parent process and the process is blocked
    - When it receives SIGCHLD, the corresponding signal handling routine is invoked
    - The SIGCHLD handler accepts and removes the SIGCHLD signal and destroys the child process in the kernel-space (remove it from process table, task-list, etc.)
    - Now the child process is truly dead and recycled
- If the parent process hasn't invoked wait() system call
    - the parent process doesn't respond to the SIGCHLD signal
    - Thus, the child process becomes a zombie until the parent process invoked wait() and receive its SIGCHLD signal

4. What are the three methods of transferring the control of the CPU from a user process to OS kernel?

- system call: system calls in user program require the switch from user mode to kernel mode, which typically ask for the control of kernel resources
- interrupt: Typically invoked by interrupt signals, the signals received is passed to trap handler who further dispatches the task to the corresponding interrupt handler function
- exception: Typically invoked by exception signals, the signals received is passed to trap handler who further dispatches the task to the corresponding exception handler function

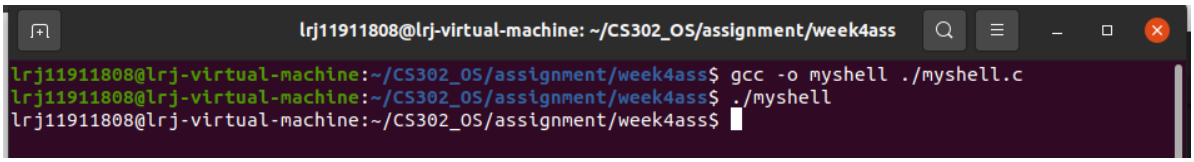
5. Describe the life cycle of a process

- Birth: Except the first process “init”, every process is created using fork()
- After one process is just forked or OS scheduler choose another process to run or a process is just returning from blocked states, the process will enter `ready state`
- When OS chooses this process to be running on the CPU, the process state is adjusted to `running state`
- While the process the running, it may wait for some resources, which will cause the process to be in `blocking state`
- In `blocking state`, when response arrives, the status of the process changes back to `ready state`
- When the process goes to an end or is forced to terminate, the process enters `Zombie(terminated) state`

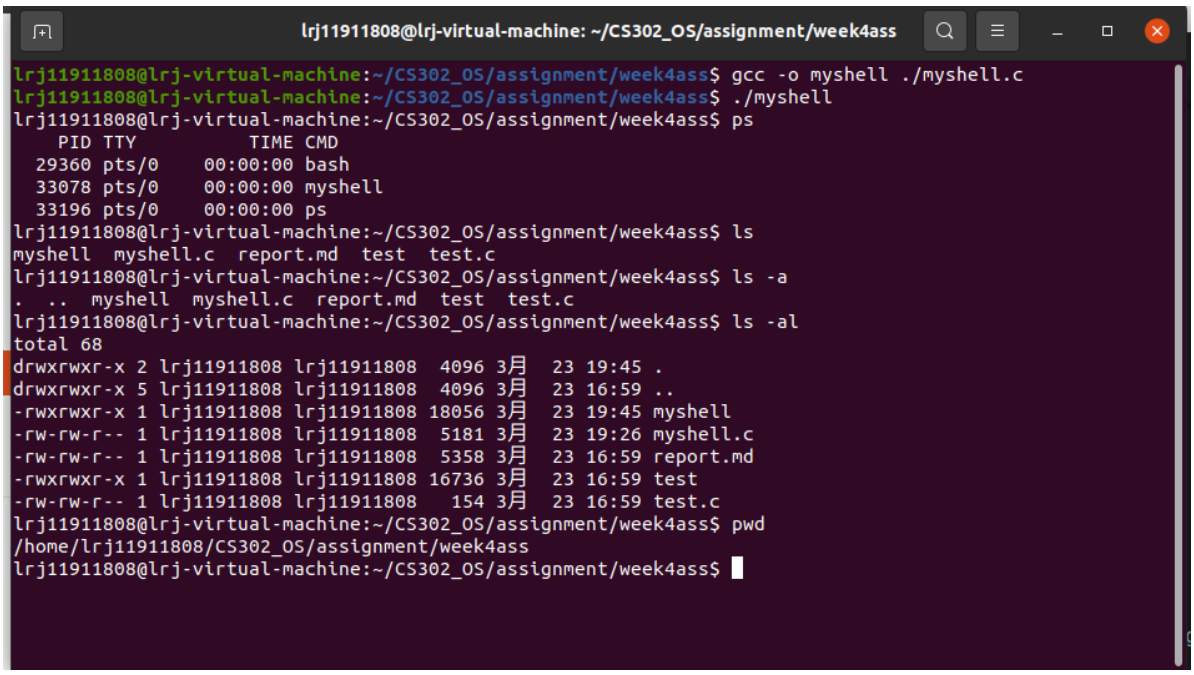
6. Realize a shell of your own in myshell.c through fork () + exec () + wait ()

Results Display

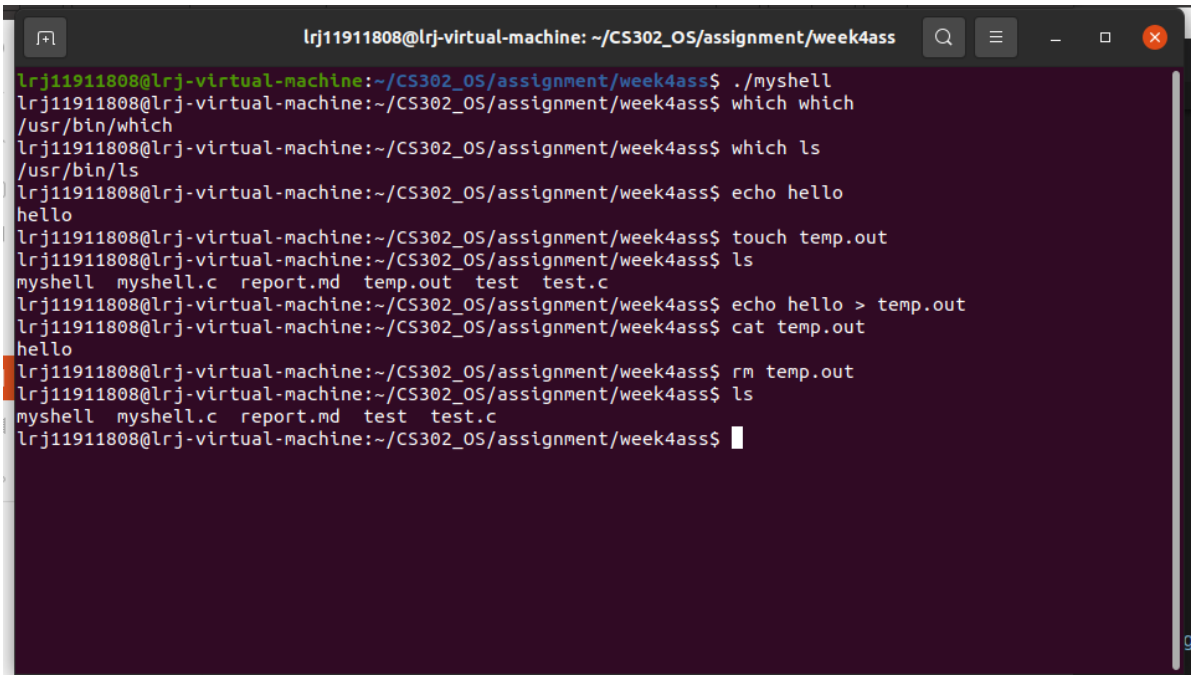
- start the shell



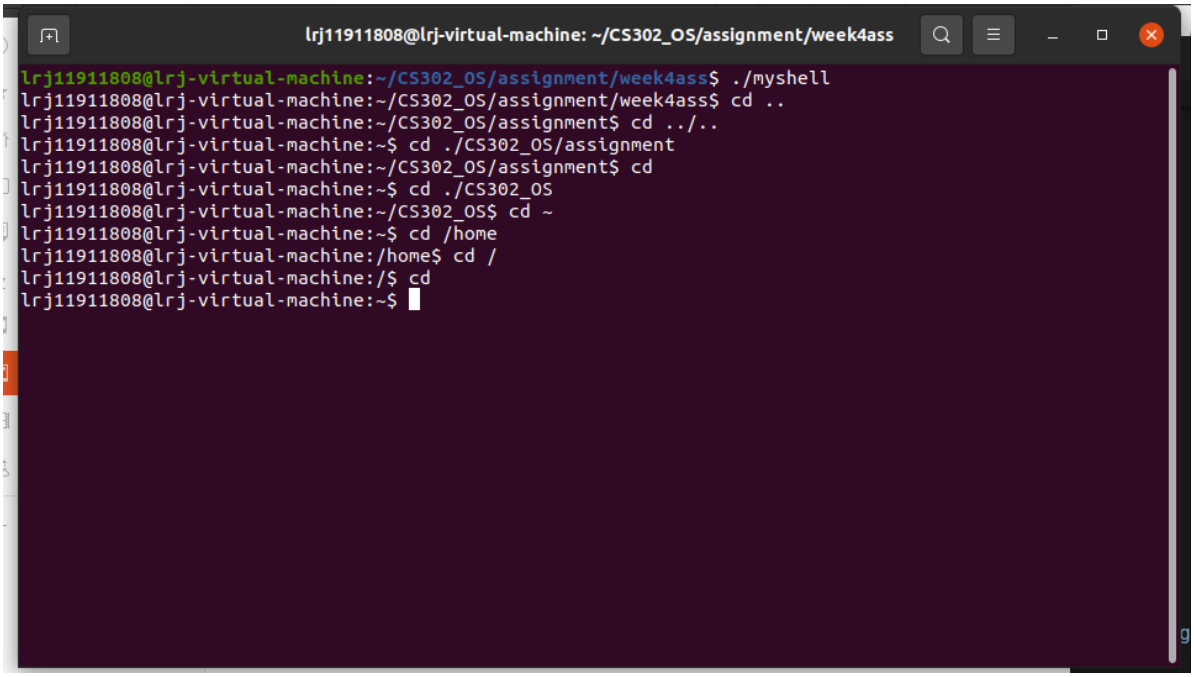
- Basic Instructions -- ps, ls, ls -al, pwd



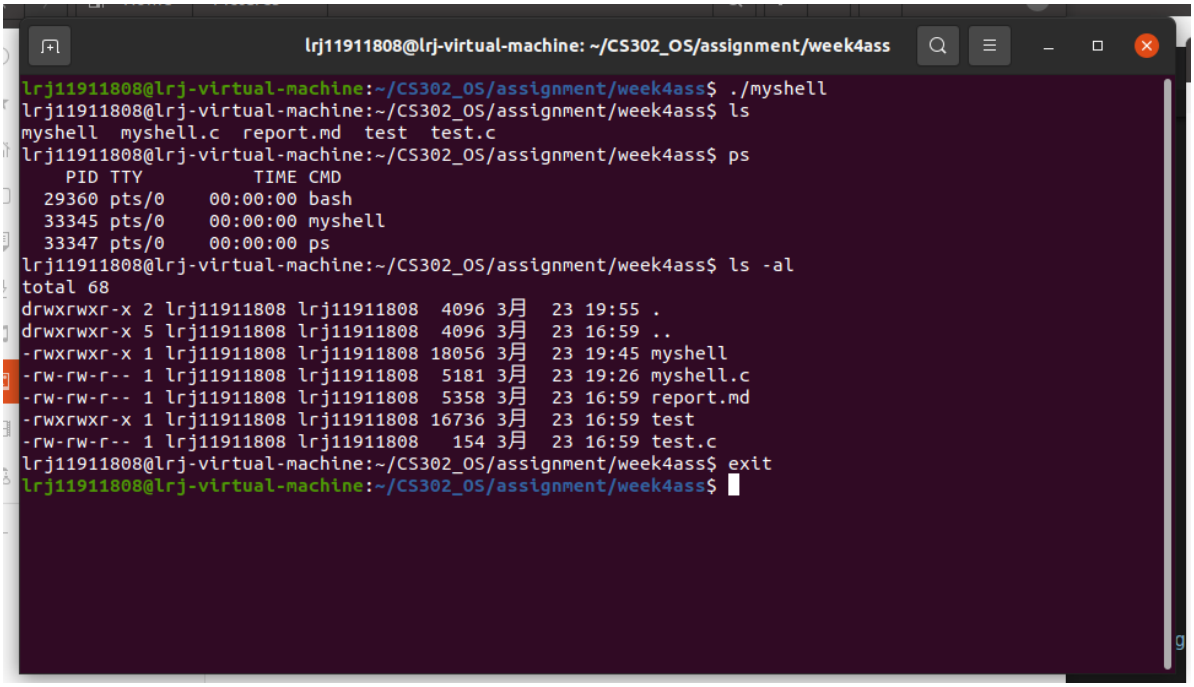
- Basic Instructions -- which, echo, touch, cat, rm



- cd .., cd, cd ~, cd /home, cd /

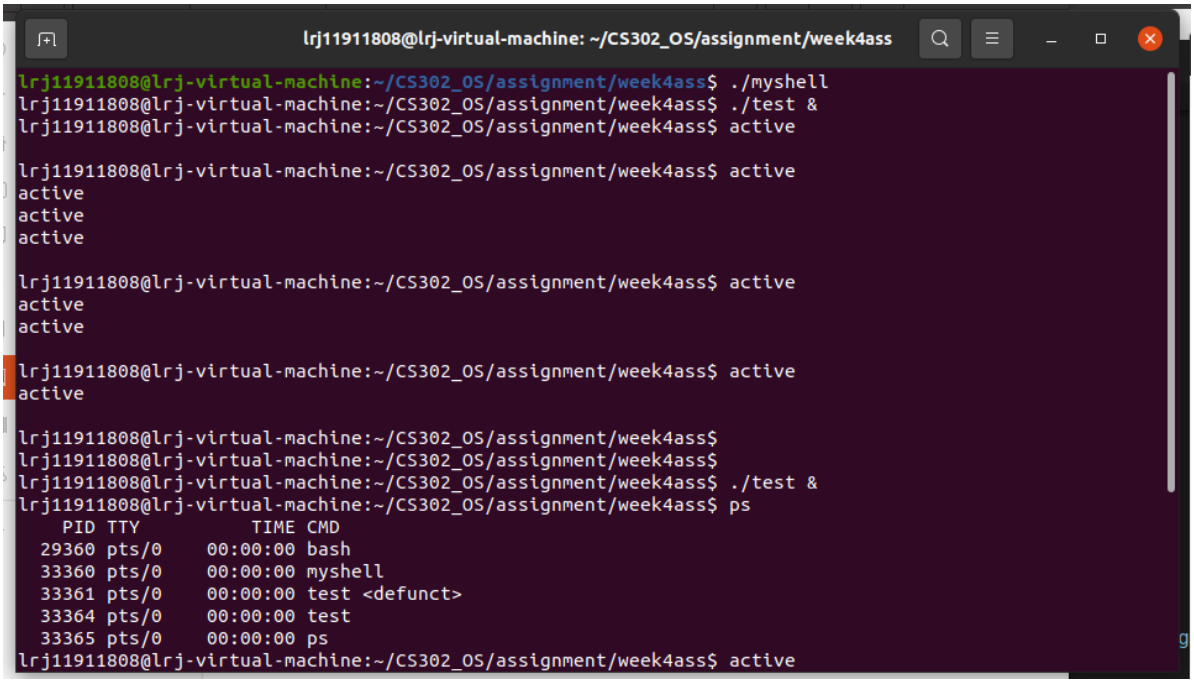


- exit

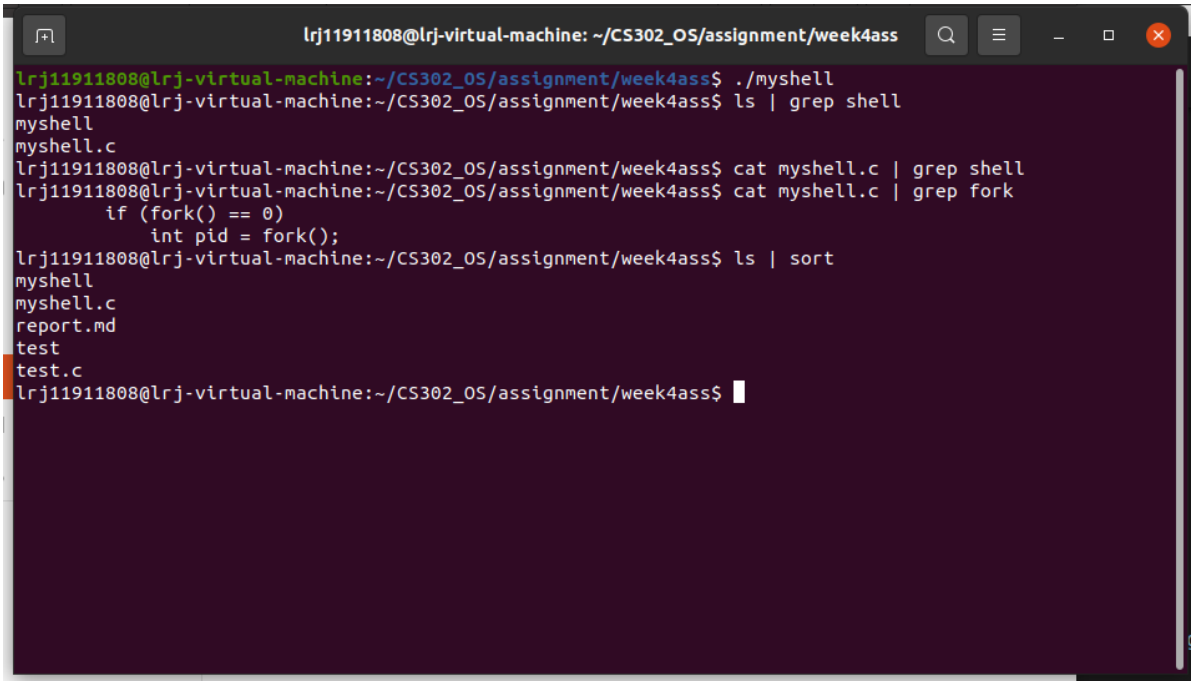


- background process

One can launch background process using `&` and the terminal goes without being blocked

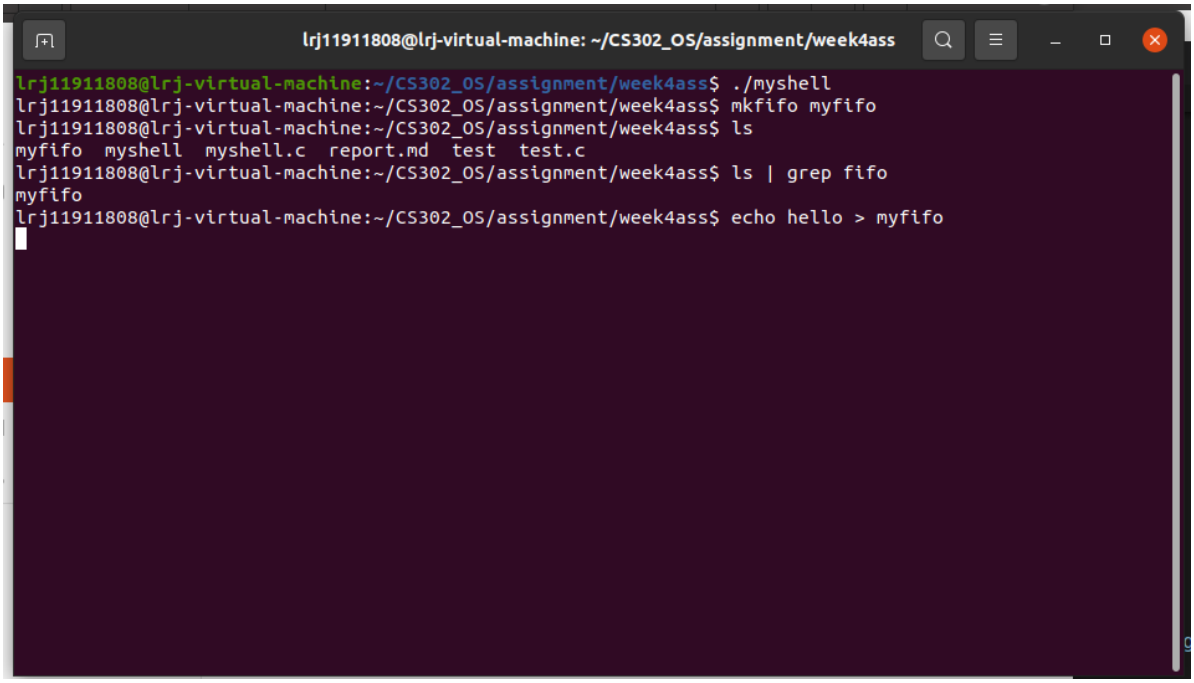


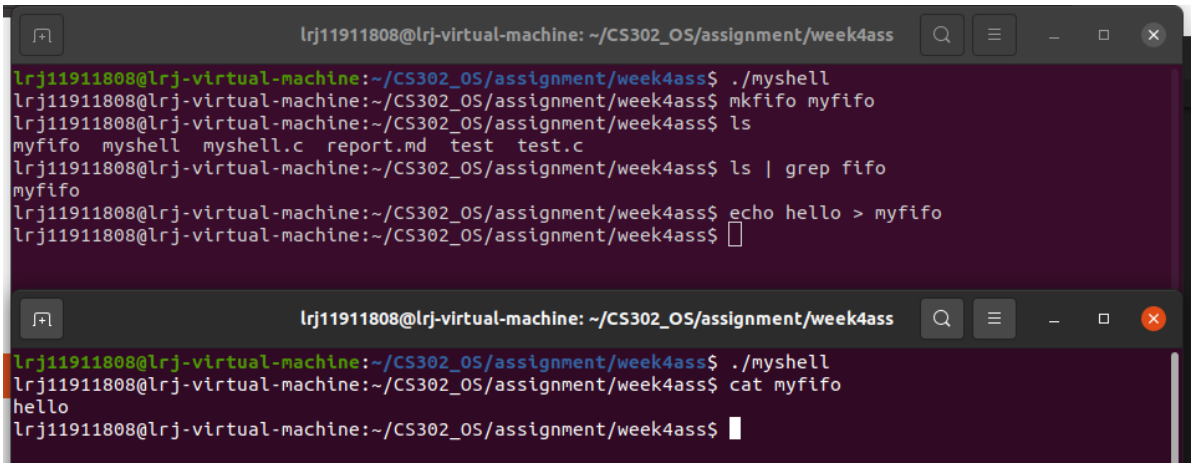
- pipe



- FIFO

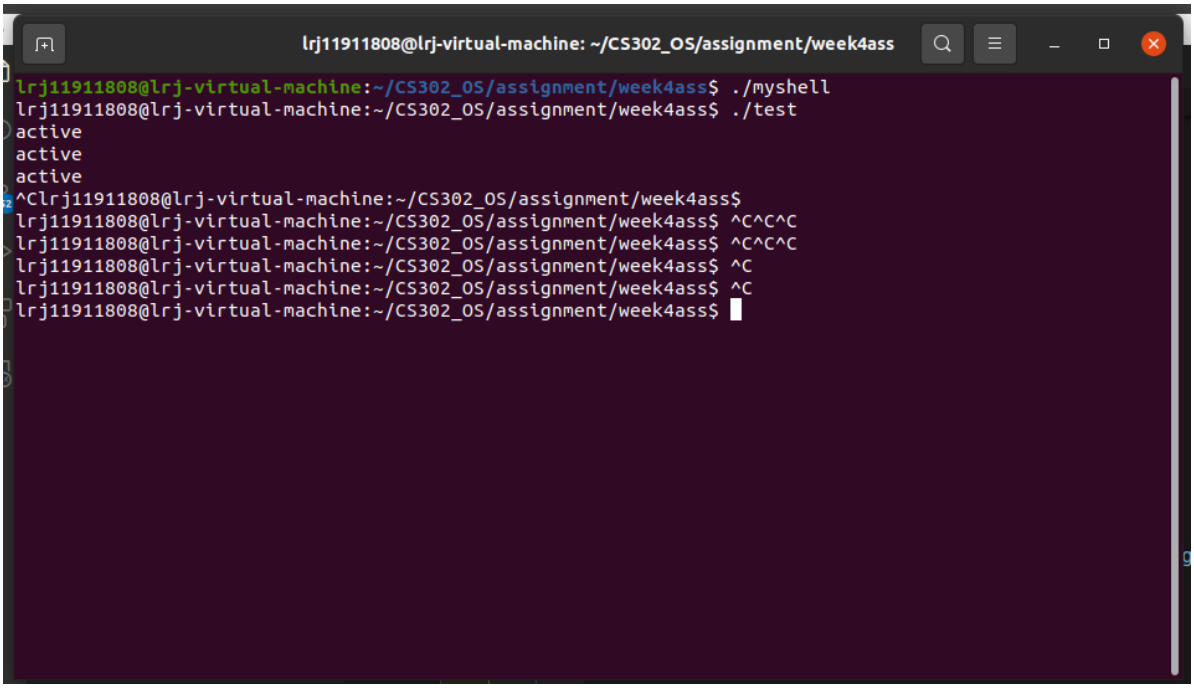
one can launch a FIFO as usual and it will be blocked until someone reads





- ctrl + c

ctrl + c can terminate child process while it loses efficacy on the shell itself



Codes

- start the shell, get `username`, `hostname`, `pwd`. Also check if the `pwd` can be simplified by replacing `/home/username` with `~`. Then print it on the screen.

```
int main()
{
    signal(SIGINT, SIG_IGN);
    char hostName[128];
    char *pwd = (char *)malloc(128 * sizeof(char));
    char *userName;
    char line[1024];
    struct Command commands[128];
    while (1)
    {
        userName = getenv("USER");
        gethostname(hostName, 128);
        getcwd(pwd, 128);
        maySimplifyPwd(pwd, userName);
        printf("%s@%s:%s$ ", userName, hostName, pwd);

        fgets(line, 1024, stdin);
        if (strcmp(line, "\n") == 0)
        {
            continue;
        }
        line[strlen(line) - 1] = '\0';
        // start split the string
        struct Redirection redirection;
        redirection.flag = 0;
        int size = parseLine(commands, line, &redirection);
```

- handle `exit` command

```
if (strcmp(commands[0].args[0], "exit") == 0)
{
    break;
}
```

- handle `cd` command

```
else if (strcmp(commands[0].args[0], "cd") == 0)
{
    if (commands[0].length == 1 || strcmp(commands[0].args[1], "~") == 0)
    {
        commands[0].args[1] = getenv("HOME");
    }
    chdir(commands[0].args[1]);
}
```

- handle `background process`

```
int bgFlag = 0;
if (strcmp(commands[size - 1].args[commands[size - 1].length - 1], "&") == 0)
{
    bgFlag = 1;
}
int pid = fork();
if (pid)
{
    if (!bgFlag)
    {
        waitpid(pid, NULL, 0);
    }
}
else
{
    signal(SIGINT, SIG_DFL);
    if (bgFlag)
    {
        commands[size - 1].args[commands[size - 1].length - 1] = NULL;
        setpgid(0, 0);
        dup(STDIN_FILENO);
        dup(STDOUT_FILENO);
        dup(STDERR_FILENO);
    }
```

- handle `single` command with output redirection (`FIFO`)

```
void singleProcess(struct Command command, struct Redirection redirection)
{
    int error = 0;
    if (redirection.flag)
    {
        int file = open(redirection.file, O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU | S_IRGRP | S_IROTH);
        dup2(file, STDOUT_FILENO);
    }
    error = execvp(command.args[0], command.args);
    if (error == -1)
    {
        perror("execvp");
    }
}
```

- handle `pipe`

```c
void pipeProcess(struct Command commands[], int size, struct Redirection redirection)
{
    int myPipe[2];
    if (pipe(myPipe) < 0)
    {
        perror("pipe");
    }
    for (int i = 0; i < size - 1; i++)
    {
        if (fork() == 0)
        {
            if (i == 0)
            {
                dup2(myPipe[1], STDOUT_FILENO);
            }
            else
            {
                dup2(myPipe[1], STDOUT_FILENO);
                dup2(myPipe[0], STDIN_FILENO);
            }
            int error = 0;
            error = execvp(commands[i].args[0], commands[i].args);
            if (error == -1)
            {
                perror("execvp");
            }
            return;
        }
        wait(NULL);
    }
    close(myPipe[1]);
    dup2(myPipe[0], STDIN_FILENO);
    int error = 0;
    if (redirection.flag)
    {
        int file = open(redirection.file, O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU | S_IRGRP | S_IROTH);
        dup2(file, STDOUT_FILENO);
    }
    error = execvp(commands[size - 1].args[0], commands[size - 1].args);
    if (error == -1)
    {
        perror("execvp");
    }
    close(myPipe[0]);
}
```