

IMPLEMENTATION OF A BASIC SHELL IN XV6

A MINI PROJECT REPORT

Submitted towards the professional course

15Z404 Operating Systems

Abishek S (17z302)

Anusha SM (17z306)

Pitchappan P RM (17z336)

Suraj Subramanian A (17z352)

BACHELOR OF ENGINEERING

Branch: Computer Science and Engineering



April 2019

Department of Computer Science and Engineering

PSG College of Technology,

Coimbatore - 641 004.

TABLE OF CONTENTS

CHAPTER NO.	TITLE
1	What is XV6?
2	Life Cycle Model Used and Reasons
3	SRS
4	Data Flow Diagrams
5	Modules Description
6	Codes with output
7	Conclusion

WHAT IS XV6?

For many years, MIT had no operating systems course. In the fall of 2002, one was created to teach operating systems engineering. In the course lectures, the class worked through Sixth Edition Unix (aka V6) using John Lions's famous commentary. In the lab assignments, students wrote most of an exokernel operating system, eventually named Jos, for the Intel x86. Exposing students to multiple systems—V6 and Jos—helped develop a sense of the spectrum of operating system designs.

V6 presented pedagogic challenges from the start. Students doubted the relevance of an obsolete 30-year-old operating system written in an obsolete programming language (pre-K&R C) running on obsolete hardware (the PDP-11). Students also struggled to learn the low-level details of two different architectures (the PDP-11 and the Intel x86) at the same time. By the summer of 2006, we had decided to replace V6 with a new operating system, xv6, modeled on V6 but written in ANSI C and running on multiprocessor Intel x86 machines. Xv6's use of the x86 makes it more relevant to students' experience than V6 was and unifies the course around a single architecture. Adding multiprocessor support requires handling concurrency head on with locks and threads (instead of using special-case solutions for uniprocessors such as enabling/disabling interrupts) and helps relevance. Finally, writing a new system allowed us to write cleaner versions of the rougher parts of V6, like the scheduler and file system. 6.828 substituted xv6 for V6 in the fall of 2006.

This was a very basic OS completely coded in ANSI C with only the bare minimum functionalities required. This Operating System has always been a very valuable resource for students wanting to delve deeper into the internal workings of an OS. It also allows people to implement their own designs into the XV6 and make it work. People can try to implement everything from their own shell to an editor or some CLI commands and test it out in the OS.

LIFE CYCLE MODEL USED AND THE REASON BEHIND USAGE OF SUCH MODEL:

The model that we used in our project is waterfall model. The reason why we used such model is because it is a systematic approach that gets initiated with the collection of well-defined requirements followed by planning, modeling, construction and deployment. The major reasons are as follows:

- 1- It's a small scale project with no constraint but time which is just a few weeks
- 2- The requirements are well defined and should remain very stable
- 3- Any future requirements can wait till the basic core product is finished

PHASES INVOLVED:

- Requirement analysis and specification
- Design-Focuses on data structure, software architecture etc.
- Implementation and Unit Testing.

System Requirements Specification

1 Introduction

1.1 Purpose

- Run an improvised version of the MIT XV6 basic OS
- Implement most common Command Line Interface functionalities in XV6
- Enhance smooth operation of the XV6
- Ensure security for the all the documents which will be saved in XV6

1.2 Scope

With the decrease in the number of people actually learning to work with the base OS like XV6 due to its lack of functionality even for educational purposes. We took it upon ourselves to create a Shell in XV6 with all the functionalities which we think is absolutely necessary for us someone to use it properly without any problem.

1.3 Overview of the system

The system focuses on improving the already existing open source XV6-public OS distribution by MIT on GitHub and use create the basic shell functionalities like Copying, Moving and Editing files and also to display all running process. This means we create a basic working Editor and add extra functionalities into it while at the same time implementing all missing common Linux commands.

2 General Requirements

- Basic XV6 – use the MIT XV6 as a base code and make it run
- Copy – Implement a copy function to copy files from one location to another
- Move – enable moving a file from one location to another using the function
- Head – display first 10 lines of any file
- Tail – Display last 10 lines of any file
- Editor – Create a basic editor to create and modify files
- Process Display – display all running process

3 Functional Requirements

3.1 Necessary requirements

- The user should have general computer knowledge
- The users should have a popular Linux Distribution
- User should have a virtualization command like Qemu or Qemu-KVD
- User should be comfortable with working on a sole Command-Line-Interface without any mouse usage

3.2 Technical requirements

- Linux Distro with QEMU or any other Virtualization support must be installed

4 Interface requirements

4.1 Software Requirements

Visual Studio Code – A basic editor for modifying the code

4.2 Hardware Requirements

- Intel core i3 processor at 2.0 GHz or higher
- 256 MB RAM or higher
- 256 GB Hard disk

6 Performance Requirements

- Response time of the system should be as quick as possible.
- In case of technical issues, The system should try to handle it without entering Panic State

7 References

- XV6 MIT PDOS
- COL331/COL633 Operating Systems Course Lecture Videos
- XV6 Survival Guide

Data Flow Models

Level 0 DFD

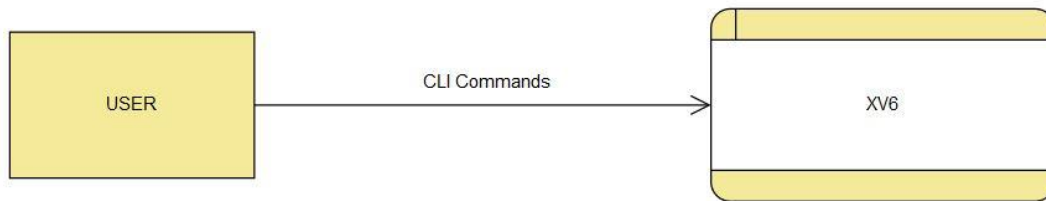


TABLE DESCRIPTIONS

Main Memory

The RAM and HDD/SSD parts of an OS where all data is finally stored. It does not lose any data even when the OS enters a panic state or is shut down. It has a logical memory address or physical memory address. The RAM houses all files which are for immediate access while the HDD/SSD houses the rest.

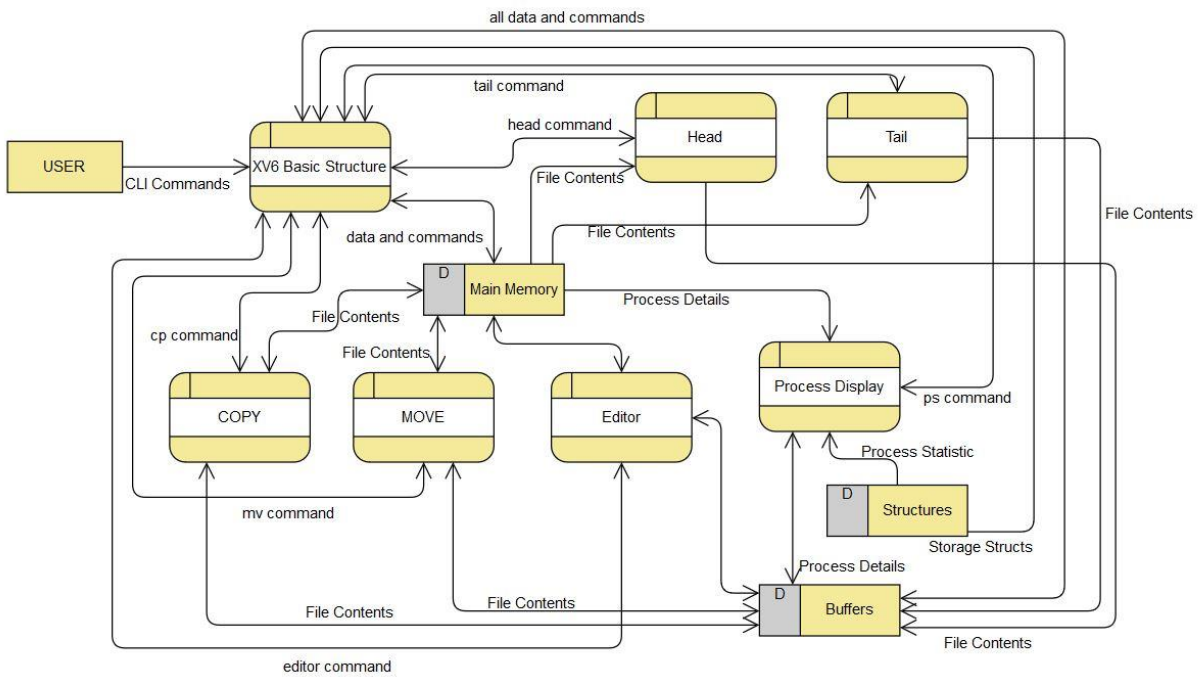
Buffers

The buffers are streams or intermediate storages that house all data for display or modification. The stream 2 is connected straight to the output terminal and is used for displaying in the Terminal. The other streams are used to carry around information and commands from all devices and the CPU.

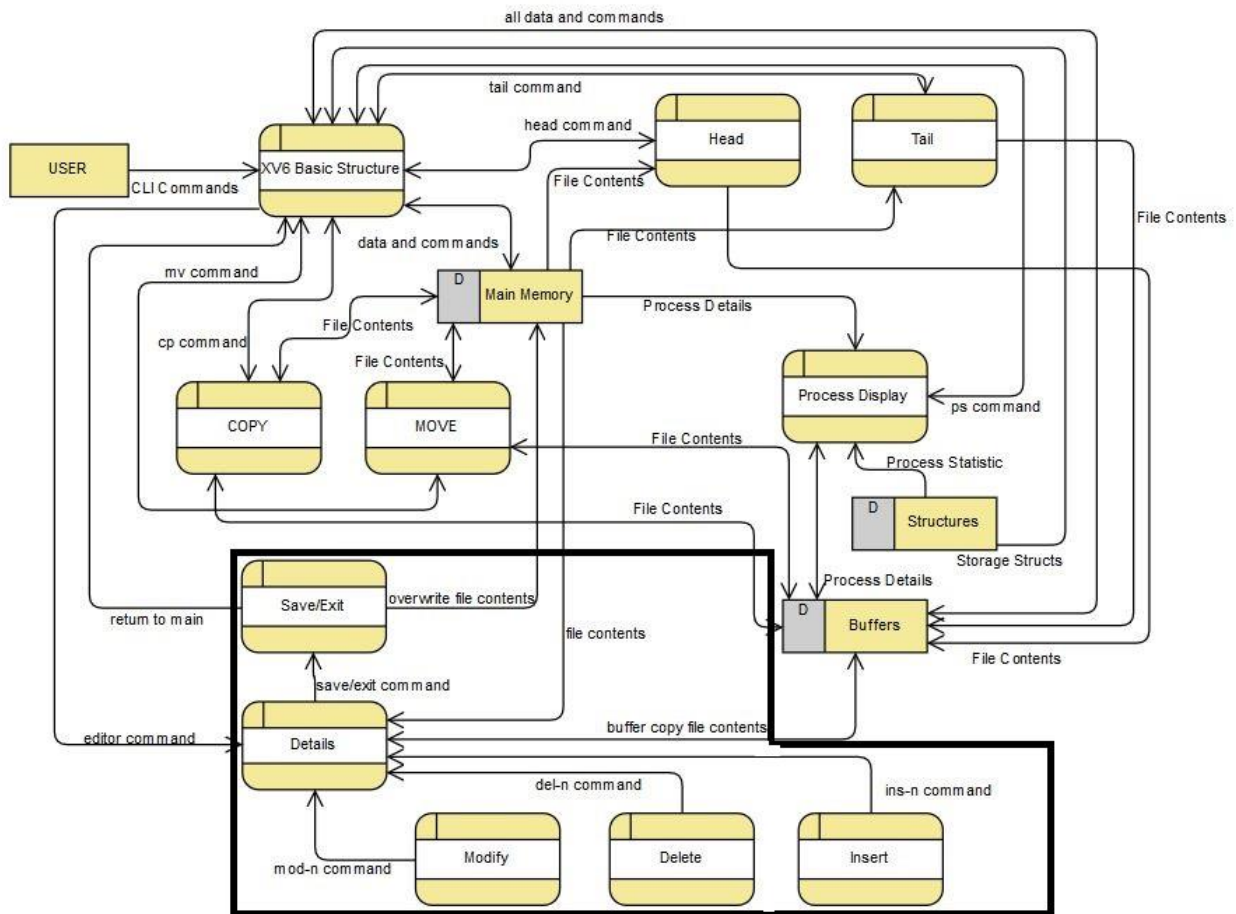
Structures

This Data Store stores all necessary structures required for functioning of a CPU. This table has predefined structures and cannot be modified unless the change is done directly to the source code. This data store houses the structures of Process Statistics or File Structures and is used for initiation of all core functionalities of a system.

Level 1 DFD



Level 2 DFD



MODULES DESCRIPTION

Copy

Syntax: cp file1 file2

Mandatory Parameters: file1, file2

This module is invoked with help of the command cp. this command copies the contents of file1 to file2. but internally what happens is that it reads the contents of file1 in a buffer and writes the content in file 2 from the buffer continuously until the end of file1, here the file2 is created if it doesn't exist else it is overwritten on the existing file specified. The contents of the file1 is unharmed. Since a copy of file1 is created, more space will be occupied in the memory Here all parameters are mandatory for the invocation of the module. This module is based on command line arguments where the inputs are passed as arguments to the module when it is invoked.

Move

Syntax: mv file1 file2

Mandatory Parameters: file1, file2

This module is invoked with help of the command mv. This command moves the contents of file1 to file2, but internally what happens is that it reads the contents of file1 in a buffer and writes the content in file 2 from the buffer continuously until the end of file1 and then in the end file1 is deleted from the memory, here the file2 is created if it doesn't exist else it is overwritten on the existing file specified. There will be no change in space of the memory since the file1 is deleted. Here all parameters are mandatory for the invocation of the module. This module is based on command line arguments where the inputs are passed as arguments to the module when it is invoked.

Head

Syntax: head file1 n

Mandatory Parameter: file1

This module is invoked when the command head is passed. This command prints the contents at the start of the file. Here the “n” parameter in the command specifies the number of lines to be printed from the start of file, by default it is taken as 10, it’s an optional parameter which means that the user doesn’t have to mention the value of “n” each time when the module has to be invoked. The internal working of the module is that it read the contents of the file1 through a buffer and then writes it to the terminal so that the user can read the first “n” lines if the value has been provided else the value 10 will be assigned for “n”. Here not all parameters are mandatory for the invocation of the module. This module is based on command line arguments where the inputs are passed as arguments to the module when it is invoked.

Tail

Syntax: tail file1 n

Mandatory Parameter: file1

This module is invoked when the command tail is passed. This command prints the contents at the end of the file. Here the “n” parameter in the command specifies the number of lines to be printed end of the file, by default it is taken as 10, it’s an optional parameter which means that the user doesn’t have to mention the value of “n” each time when the module has to be invoked. The internal working of the module is that it read the contents of the file1 through a buffer and then writes it to the terminal so that the user can read the last “n” lines if the value has been provided else the value 10 will be assigned for “n”. Here not all parameters are mandatory for the invocation of the module. This module is based on command line arguments where the inputs are passed as arguments to the module when it is invoked

Process display

Syntax: ps

Mandatory Parameters: None

This module is invoked when the command ps is passed. This is invoked when the user demands to see the currently running processes, Memory allocation to each process, total run time of a process, starting address of a process, size of a process, total CPU utilization of a process, id of a process, parent id for each process, current state of the process, name of the process and total no flags used by a process. The internal working of ps module is that it accesses the process structure and then it fetches the data it needs and then checks whether a process is "UNUSED" or not, if the former then the data is ignored and then the next set of data is fetched from the memory and checked else if it's the latter then the details mentioned above will be written to the output buffer and then onto the terminal. No parameters are required to invoke this module. This module is based on command line arguments where the inputs are passed as arguments to the module when it is invoked.

Editor

Syntax: editor file1 or bedit file1 mode

Mandatory Parameters: file1

This module is used to open a basic editor that can be used to create a new file or view and modify an existing file. The editor can be used to insert, modify or delete a particular line. It can also be used to insert a huge block of text. The editor can also be used to add lines at end of the file. The editor displays the number of lines at each line and that can be used to specify after which line you need to insert or modify. When invoked, the editor goes to fetch the filename and if its non-existent, it then goes on to create a file of the given name. It then prints the whole text along with line numbers and then shows all possible options to choose from and execute. At the end, you can choose to exit with or without saving all changes.

CODES WITH OUTPUT

CP

CODE

```
1  #include "types.h"
2  #include "fcntl.h"
3  #include "stat.h"
4  #include "user.h"
5  #include "fs.h"
6
7  int main(int argc, char *argv[])
8  {
9      if (argc != 3){
10         printf(2, "cp: too many or less args\n");
11         exit();
12     }
13     char buf[512];
14     int fd1,fd2,n;
15     struct stat st;
16     if((fd1 = open(argv[1], O_RDONLY)) < 0){
17         printf(2, "cp: Cannot open %s\n", argv[1]);
18         exit();
19     }
20     if(fstat(fd1, &st) < 0){
21         printf(2, "cp: Cannot stat %s\n", argv[1]);
22         close(fd1);
23         exit();
24     }
25     if (st.type == T_FILE){
26         if((fd2 = open(argv[2], O_CREATE)) < 0){
27             close(fd2);
28             close(0);
29             if((fd2 = open(argv[2], O_WRONLY) < 0)){
30                 printf(2, "cp: Cannot open %s\n", argv[2]);
31                 exit();
32             }
```

```

33     }
34     close(fd2);
35     close(0);
36     if((fd2 = open(argv[2], O_WRONLY) < 0)){
37         printf(2, "cp: Cannot open %s\n", argv[2]);
38         exit();
39     }
40     for(;;) {
41         if((n = read(fd1, buf, sizeof(buf))) < 0)
42         {
43             printf(2, "cp: Error while copying contents1\n");
44             exit();
45         }
46         if(n == 0)
47         {
48             break;
49         }
50         if(write(fd2, buf, n) != n)
51         {
52             printf(2, "cp: Error while copying contents2\n");
53             exit();
54         }
55     }
56     close(fd2);
57 }else{
58     printf(2, "cp: invalid args");
59 }
60 close(fd1);
61 exit();
62 }
63

```

OUTPUT

```
$ deatheater@ubuntu:~/Desktop/xv6-public$ make qemu
qemu-system-x86_64 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
qemu-system-x86_64: warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ cp README README1
$ cat README1
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, but is implemented for a modern x86-based multiprocessor using ANSI C.

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14, 2000)). See also https://pdos.csail.mit.edu/6.828/, which
```


MOVE

CODE

```
1  #include "types.h"
2  #include "fcntl.h"
3  #include "stat.h"
4  #include "user.h"
5  #include "fs.h"
6
7  int main(int argc, char *argv[])
8  {
9      if (argc != 3){
10         printf(2, "mv: too many or less args\n");
11         exit();
12     }
13     char buf[512];
14     int fd1, fd2, n;
15     struct stat st;
16     if((fd1 = open(argv[1], O_RDONLY)) < 0){
17         printf(2, "mv: Cannot open %s\n", argv[1]);
18         exit();
19     }
20     if(fstat(fd1, &st) < 0){
21         printf(2, "mv: Cannot stat %s\n", argv[1]);
22         close(fd1);
23         exit();
24     }
25     if (st.type == T_FILE){
26         if((fd2 = open(argv[2], O_CREATE)) < 0){
27             close(fd2);
28             close(0);
29             if((fd2 = open(argv[2], O_WRONLY)) < 0){
30                 printf(2, "mv: Cannot open %s\n", argv[2]);
31                 exit();
            }
```

```

32     }
33 }
34 close(fd2);
35 close(0);
36 if((fd2 = open(argv[2], O_WRONLY) < 0)){
37     printf(2, "mv: Cannot open %s\n", argv[2]);
38     exit();
39 }
40 for(;;) {
41     if((n = read(fd1,buf,sizeof(buf))) < 0){
42         printf(2,"mv: Error while copying contents1\n");
43         exit();
44     }
45     if(n == 0){
46         break;
47     }
48     if(write(fd2,buf,n) != n){
49         printf(2,"mv: Error while copying contents2\n");
50         exit();
51     }
52 }
53 close(fd2);
54 }else{
55     printf(2, "mv: invalid args");
56 }
57 close(fd1);
58 unlink(argv[1]);
59 exit();
60 }
61

```

OUTPUT

```
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ mv README abc
$ cat README
cat: cannot open README
$ cat abc
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,
but is implemented for a modern x86-based multiprocessor using ANSI C.
```

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14, 2000)). See also <https://pdos.csail.mit.edu/6.828/>, which provides pointers to on-line resources for v6.

HEAD

CODE

```
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4  #define INT_MAX 2147483647
5
6  char buf[512];
7
8  void
9  Head(int fd,int Limit)
10 {
11     int Current_Read = 0;
12     int n;
13     while((n = read(fd, buf, 1)) > 0 && Limit-1>= Current_Read ) {
14         if (write(1, buf, n) != n) {
15             printf(1, "Head: write error\n");
16             exit();
17         }
18         if (buf[0] == '\n')
19             Current_Read += 1;
20     }
21     if(n < 0){
22         printf(1, "Head: read error\n");
23         exit();
24     }
25 }
26
27
28
29
30
```

```
31 int
32 main(int argc, char *argv[])
33 {
34     int fd;
35
36     if(argc <= 1){
37         Head(0,INT_MAX);
38         exit();
39     }
40
41
42     if((fd = open(argv[1], 0)) < 0){
43         printf(1, "Head: cannot open %s\n", argv[1]);
44         exit();
45     }
46     if (argc == 2)
47         Head(fd,10);
48     else if (argc == 3)
49         Head(fd,atoi(argv[2]));
50     else
51         printf(2,"Head: Too many args");
52     close(fd);
53     exit();
54 }
55
```

OUTPUT

```
$ head 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,
but is implemented for a modern x86-based multiprocessor using ANSI C.
```

ACKNOWLEDGMENTS

```
xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer
to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14,
2000)). See also https://pdos.csail.mit.edu/6.828/, which
provides pointers to on-line resources for v6.
```

TAIL

CODE

```
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4  #define INT_MAX 2147483647
5  char buf[512];
6  void
7  Tail(int fd,int Limit, char *Path)
8  {
9      int Total_Lines= 0;
10     int Counter = 0;
11     int Start;
12     int n;
13     while((n = read(fd, buf, 1)) > 0) {
14         if(buf[0] == '\n')
15             Total_Lines += 1;
16     }
17     if(n < 0){
18         printf(1,"Tail: read error\n");
19         exit();
20     }
21     Start = Total_Lines - Limit;
22     if (Start < 0){
23         printf(1, "Tail:Invalid no of lines");
24     }
25     close(fd);
26     close(0);
27     fd = open(Path,0);
28     while((n = read(fd, buf, 1)) > 0) {
29         if(Counter >= Start)
30         {
31             if((write(1,buf,1)) != 1)
```

```

32         {
33             printf(1, "Head: write error\n");
34             exit();
35         }
36     }
37     if(buf[0] == '\n')
38         Counter += 1;
39 }
40 close(fd);
41 }
42 int
43 main(int argc, char *argv[])
44 {
45     int fd;
46     if(argc <= 1){
47         Tail(0, INT_MAX, ".");
48         exit();
49     }
50     if((fd = open(argv[1], 0)) < 0){
51         printf(1, "Tail: cannot open %s\n", argv[1]);
52         exit();
53     }
54     if (argc == 2)
55         Tail(fd, 10, argv[1]);
56     else if (argc == 3)
57         Tail(fd, atoi(argv[2]), argv[1]);
58     else
59         printf(2, "Head: Too many args");
60     close(fd);
61     exit();
62 }
63

```


OUTPUT

```
$ tail README
simplifications and clarifications than new features.

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run
"make". On non-x86 or non-ELF machines (like OS X, even on x86), you
will need to install a cross-compiler gcc suite capable of producing
x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC
simulator and run "make qemu".
$
```

PS

CODE

```
1  #include "types.h"
2  #include "user.h"
3  #include "param.h"
4  #include "memlayout.h"
5  #include "mmu.h"
6  #include "x86.h"
7  #include "proc.h"
8  #include "spinlock.h"
9  #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
10 struct {
11     struct spinlock lock; struct proc proc[NPROC];
12 } ptable;
13 int
14 main(int argc, char *argv[]){
15     static char *states[] = {
16         [UNUSED]    "unused", [EMBRYO]    "embryo",
17         [SLEEPING]  "sleep ", [RUNNABLE]  "runble",
18         [RUNNING]   "run   ", [ZOMBIE]    "zombie"
19     };
20     char *state; struct proc *p;
21     printf(1, "F  S  UID  PID  PPID  SZ  WCHAN  COMD\n");
22     for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
23         if (p->state == UNUSED) continue;
24         if (p->state >= 0 && p->state < NELEM(states) && states[p->state])
25             state = states[p->state];
26         else
27             state = "???";
28         printf(1, "%2s  Root  %d  %d  %d  %d\n",
29             state, p->pid, p->parent->pid, p->sz, p->chan, p->name);
30     }
31     exit();
32 }
```

OUTPUT

```
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ ps
F    S    UID    PID    PPID    SZ    WCHAN    CMD
$ 1 sleep  init 80103f17 80103fb7 80104969 801059b9 8010570f
2 sleep  sh 80103edc 801002da 8010102c 80104c42 80104969 801059b9 8010570f
7
```

EDITOR

CODE

```
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4  #include "fcntl.h"
5  #include "fs.h"
6
7  #define BUF_SIZE 256
8  #define MAX_LINE_NUMBER 256
9  #define MAX_LINE_LENGTH 256
10 #define NULL 0
11
12 char* strcat_n(char* dest, char* src, int len);
13 int get_line_number(char *text[]);
14 void show_text(char *text[]);
15 void com_ins(char *text[], int n, char *extra);
16 void com_mod(char *text[], int n, char *extra);
17 void com_del(char *text[], int n);
18 void com_help(char *text[]);
19 void com_save(char *text[], char *path);
20 void com_exit(char *text[], char *path);
21 int stringtonumber(char* src);
22
23 int changed = 0;
24 int auto_show = 1;
25
26 int main(int argc, char *argv[])
27 {
28     if (argc == 1)
29     {
30         printf(1, "please input the command as [editor file_name]\n");
31         exit();
32     }
```

```

33
34 char *text[MAX_LINE_NUMBER] = {};
35 text[0] = malloc(MAX_LINE_LENGTH);
36 memset(text[0], 0, MAX_LINE_LENGTH);
37 int line_number = 0;
38 int fd = open(argv[1], O_RDONLY);
39 if (fd != -1)
40 {
41     printf(1, "file exist\n");
42     char buf[BUF_SIZE] = {};
43     int len = 0;
44     while ((len = read(fd, buf, BUF_SIZE)) > 0)
45     {
46         int i = 0;
47         int next = 0;
48         int is_full = 0;
49         while (i < len)
50         {
51             for (i = next; i < len && buf[i] != '\n'; i++)
52                 ;
53             strcat_n(text[line_number], buf+next, i-next);
54             if (i < len && buf[i] == '\n')
55             {
56                 if (line_number >= MAX_LINE_NUMBER - 1)
57                     is_full = 1;
58                 else
59                 {
60                     line_number++;
61                     text[line_number] = malloc(MAX_LINE_LENGTH);
62                     memset(text[line_number], 0, MAX_LINE_LENGTH);
63                 }

```

```

64         }
65         if (is_full == 1 || i >= len - 1)
66             break;
67         else
68             next = i + 1;
69     }
70     if (is_full == 1)
71         break;
72 }
73 close(fd);
74 }
75 else
76 {
77     printf(1, "File do not exist\n");
78     unlink(argv[1]);
79     fd=open(argv[1],O_CREATE | O_WRONLY);
80 }
81
82 show_text(text);
83 com_help(text);
84
85 char input[MAX_LINE_LENGTH] = {};
86 while (1)
87 {
88     printf(1, "\nplease input command:\n");
89     memset(input, 0, MAX_LINE_LENGTH);
90     gets(input, MAX_LINE_LENGTH);
91     int len = strlen(input);
92     input[len-1] = '\0';
93     len--;
94     int pos = MAX_LINE_LENGTH - 1;

```

```

95     int j = 0;
96     for (; j < 8; j++)
97     {
98         if (input[j] == ' ')
99         {
100             pos = j + 1;
101             break;
102         }
103     }
104     //ins
105     if (input[0] == 'i' && input[1] == 'n' && input[2] == 's')
106     {
107         if (input[3] == '-'&&stringtonumber(&input[4])>=0)
108         {
109             com_ins(text, stringtonumber(&input[4]), &input[pos]);
110             line_number = get_line_number(text);
111         }
112         else if(input[3] == ' '||input[3] == '\\0')
113         {
114             com_ins(text, line_number+1, &input[pos]);
115             line_number = get_line_number(text);
116         }
117         else
118         {
119             printf(1, "invalid command.\\n");
120             com_help(text);
121         }
122     }
123     //mod
124     else if (input[0] == 'm' && input[1] == 'o' && input[2] == 'd')
125     {
126         if (input[3] == '-'&&stringtonumber(&input[4])>=0)

```

```

127         com_mod(text, atoi(&input[4]), &input[pos]);
128     else if(input[3] == ' ' || input[3] == '\0')
129         com_mod(text, line_number + 1, &input[pos]);
130     else
131     {
132         printf(1, "invalid command.\n");
133         com_help(text);
134     }
135 }
136 //del
137 else if (input[0] == 'd' && input[1] == 'e' && input[2] == 'l')
138 {
139     if (input[3] == '-' && stringtonumber(&input[4]) >= 0)
140     {
141         com_del(text, atoi(&input[4]));
142         line_number = get_line_number(text);
143     }
144     else if(input[3] == '\0')
145     {
146         com_del(text, line_number + 1);
147         line_number = get_line_number(text);
148     }
149     else
150     {
151         printf(1, "invalid command.\n");
152         com_help(text);
153     }
154 }
155 }
156 else if (strcmp(input, "show") == 0)
157 {
158     auto_show = 1;

```



```

159         printf(1, "enable show current contents after text changed.\n");
160     }
161     else if (strcmp(input, "hide") == 0)
162     {
163         auto_show = 0;
164         printf(1, "disable show current contents after text changed.\n");
165     }
166     else if (strcmp(input, "help") == 0)
167         com_help(text);
168     else if (strcmp(input, "save") == 0 || strcmp(input, "CTRL+S\n") == 0)
169         com_save(text, argv[1]);
170     else if (strcmp(input, "exit") == 0)
171         com_exit(text, argv[1]);
172     else
173     {
174         printf(1, "invalid command.\n");
175         com_help(text);
176     }
177 }
178 exit();
179 }
180
181 char* strcat_n(char* dest, char* src, int len)
182 {
183     if (len <= 0)
184         return dest;
185     int pos = strlen(dest);
186     if (len + pos >= MAX_LINE_LENGTH)
187         return dest;
188     int i = 0;
189     for (; i < len; i++)
190         dest[i+pos] = src[i];

```

```

191     dest[len+pos] = '\0';
192     return dest;
193 }
194
195 void show_text(char *text[])
196 {
197     printf(1, "*****\n");
198     printf(1, "the contents of the file are:\n");
199     int j = 0;
200     for (; text[j] != NULL; j++)
201         printf(1, "%d%d%d:%s\n", (j+1)/100, ((j+1)%100)/10, (j+1)%10, text[j]);
202 }
203
204 int get_line_number(char *text[])
205 {
206     int i = 0;
207     for (; i < MAX_LINE_NUMBER; i++)
208         if (text[i] == NULL)
209             return i - 1;
210     return i - 1;
211 }
212
213 int stringtonumber(char* src)
214 {
215     int number = 0;
216     int i=0;
217     int pos = strlen(src);
218     for(;i<pos;i++)
219     {
220         if(src[i]==' ') break;
221         if(src[i]>57||src[i]<48) return -1;
222         number=10*number+(src[i]-48);

```

```

223     }
224     return number;
225 }
226
227 void com_ins(char *text[], int n, char *extra)
228 {
229     if (n < 0 || n > get_line_number(text) + 1)
230     {
231         printf(1, "invalid line number\n");
232         return;
233     }
234     char input[MAX_LINE_LENGTH] = {};
235     if (*extra == '\\0')
236     {
237         printf(1, "please input content:\n");
238         gets(input, MAX_LINE_LENGTH);
239         input[strlen(input)-1] = '\\0';
240     }
241     else
242         strcpy(input, extra);
243     int i = MAX_LINE_NUMBER - 1;
244     for (; i > n; i--)
245     {
246         if (text[i-1] == NULL)
247             continue;
248         else if (text[i] == NULL && text[i-1] != NULL)
249         {
250             text[i] = malloc(MAX_LINE_LENGTH);
251             memset(text[i], 0, MAX_LINE_LENGTH);
252             strcpy(text[i], text[i-1]);
253         }
254         else if (text[i] != NULL && text[i-1] != NULL)

```

```

255     {
256         memset(text[i], 0, MAX_LINE_LENGTH);
257         strcpy(text[i], text[i-1]);
258     }
259 }
260 if (text[n] == NULL)
261 {
262     text[n] = malloc(MAX_LINE_LENGTH);
263     if (text[n-1][0] == '\0')
264     {
265         memset(text[n], 0, MAX_LINE_LENGTH);
266         strcpy(text[n-1], input);
267         changed = 1;
268         if (auto_show == 1)
269             show_text(text);
270         return;
271     }
272 }
273 memset(text[n], 0, MAX_LINE_LENGTH);
274 strcpy(text[n], input);
275 changed = 1;
276 if (auto_show == 1)
277     show_text(text);
278 }
279
280 void com_mod(char *text[], int n, char *extra)
281 {
282     if (n <= 0 || n > get_line_number(text) + 1)
283     {
284         printf(1, "invalid line number\n");
285         return;
286     }

```

```

287     char input[MAX_LINE_LENGTH] = {};
288     if (*extra == '\0')
289     {
290         printf(1, "please input content:\n");
291         gets(input, MAX_LINE_LENGTH);
292         input[strlen(input)-1] = '\0';
293     }
294     else
295     {
296         strcpy(input, extra);
297         memset(text[n-1], 0, MAX_LINE_LENGTH);
298         strcpy(text[n-1], input);
299         changed = 1;
300         if (auto_show == 1)
301             show_text(text);
302     }
303
304 void com_del(char *text[], int n)
305 {
306     if (n <= 0 || n > get_line_number(text) + 1)
307     {
308         printf(1, "invalid line number\n");
309         return;
310     }
311     memset(text[n-1], 0, MAX_LINE_LENGTH);
312     int i = n - 1;
313     for (; text[i+1] != NULL; i++)
314     {
315         strcpy(text[i], text[i+1]);
316         memset(text[i+1], 0, MAX_LINE_LENGTH);
317     }
318     if (i != 0)

```

```

318 {
319     free(text[i]);
320     text[i] = 0;
321 }
322 changed = 1;
323 if (auto_show == 1)
324     show_text(text);
325 }
326
327 void com_help(char *text[])
328 {
329     printf(1, "*****\n");
330     printf(1, "show, enable show current contents after executing a command.\n");
331     printf(1, "hide, disable show current contents after executing a command.\n");
332     printf(1, "instructions for use:\n");
333     printf(1, "ins-n, insert a line after line n\n");
334     printf(1, "mod-n, modify line n\n");
335     printf(1, "del-n, delete line n\n");
336     printf(1, "ins, insert a line after the last line\n");
337     printf(1, "mod, modify the last line\n");
338     printf(1, "del, delete the last line\n");
339     printf(1, "save, save the file\n");
340     printf(1, "exit, exit editor\n");
341 }
342
343 void com_save(char *text[], char *path)
344 {
345     unlink(path);
346     int fd = open(path, O_WRONLY|O_CREATE);
347     if (fd == -1)
348     {
349         printf(1, "save failed, file can't open\n");

```

```
350         exit();
351     }
352     if (text[0] == NULL)
353     {
354         close(fd);
355         return;
356     }
357     write(fd, text[0], strlen(text[0]));
358     int i = 1;
359     for (; text[i] != NULL; i++)
360     {
361         printf(fd, "\n");
362         write(fd, text[i], strlen(text[i]));
363     }
364     close(fd);
365     printf(1, "saved successfully\n");
366     changed = 0;
367     return;
368 }
369
370 void com_exit(char *text[], char *path)
371 {
372     while (changed == 1)
373     {
374         printf(1, "save the file? y/n\n");
375         char input[MAX_LINE_LENGTH] = {};
376         gets(input, MAX_LINE_LENGTH);
377         input[strlen(input)-1] = '\0';
```

```
378  if (strcmp(input, "y") == 0)
379      com_save(text, path);
380  else if(strcmp(input, "n") == 0)
381      break;
382  else
383      printf(2, "wrong answer?\n");
384  }
385  int i = 0;
386  for (; text[i] != NULL; i++)
387  {
388      free(text[i]);
389      text[i] = 0;
390  }
391  exit();
392  }
393
394
395
396
```


OUTPUTS

```
QEMU
init: starting sh
$ editor abc
file exist
*****
the contents of the file are:
001:aaaaaaaaaaaaaaaa
002:bbbbbbbbbbbbbbbb
003:cccccccccccccc
004:
*****
show, enable show current contents after executing a command.
hide, disable show current contents after executing a command.
instructions for use:
ins-n, insert a line after line n
mod-n, modify line n
del-n, delete line n
ins, insert a line after the last line
mod, modify the last line
del, delete the last line
save, save the file
exit, exit editor

please input command:
_
```

MODIFIED SH.C for cd and pwd

CODE

```
1 // Shell.
2
3 #include "types.h"
4 #include "user.h"
5 #include "param.h"
6 #include "mmu.h"
7 #include "fcntl.h"
8 #include "proc.h"
9 #include "spinlock.h"
10 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
11
12
13 // Parsed command representation
14 #define EXEC 1
15 #define REDIR 2
16 #define PIPE 3
17 #define LIST 4
18 #define BACK 5
19 #define NULL 0
20
21 #define MAXARGS 10
22 // #define INT_MAX 2147483647
23
24
25 /// By Us
26 char *strcat(char *strg1, char *strg2)
27 {
28     char *start = strg1;
29
30     while(*strg1 != '\0')
31     {
32         strg1++;
33     }
34
35     while(*strg2 != '\0')
36     {
```

```
37 *strg1 = *strg2;
38 strg1++;
39 strg2++;
40 }
41
42 *strg1 = '\0';
43 return start;
44 }
45
46 struct cmd {
47 int type;
48 };
49
50 struct execcmd {
51 int type;
52 char *argv[MAXARGS];
53 char *eargv[MAXARGS];
54 };
55
56 struct redircmd {
57 int type;
58 struct cmd *cmd;
59 char *file;
60 char *efile;
61 int mode;
62 int fd;
63 };
64
65 struct pipecmd {
66 int type;
67 struct cmd *left;
68 struct cmd *right;
69 };
70
71 struct listcmd {
72 int type;
73 struct cmd *left;
74 struct cmd *right;
75 };
76
```

```

77 struct backcmd {
78 int type;
79 struct cmd *cmd;
80 };
81 /// pwd
82 struct directory{
83 char string[100];
84 struct directory *Next;
85 struct directory *Before;
86 };
87
88 int fork1(void); // Fork but panics on failure.
89 void panic(char*);
90 struct cmd *parsecmd(char*);
91
92 struct {
93 struct spinlock lock;
94 struct proc proc[NPROC];
95 } ptable;
96
97 ///Build Directory
98 struct directory* CreateNode(char *Str)
99 {
100 struct directory* Temp = malloc(sizeof(struct directory));
101 //Temp->string = malloc(sizeof(Str));
102 strcpy(Temp->string, Str);
103 Temp->Before = Temp->Next = NULL;
104 return Temp;
105 }
106
107 // Execute cmd. Never returns.
108 void
109 runcmd(struct cmd *cmd)
110 {
111 int p[2];
112 struct backcmd *bcmd;
113 struct execcmd *ecmd;
114 struct listcmd *lcmd;
115 struct pipecmd *pcmd;
116 struct redircmd *rcmd;

```

```
117 char Point[] = "/" ;
118
119 if(cmd == 0)
120 exit();
121
122 switch(cmd->type){
123 default:
124 panic("runcmd");
125
126 case EXEC:
127 ecmd = (struct execcmd*)cmd;
128 if(ecmd->argv[0] == 0)
129 exit();
130 exec(strcat(Point,ecmd->argv[0]), ecmd->argv);
131 printf(2, "exec %s failed\n", ecmd->argv[0]);
132 break;
133
134 case REDIR:
135 rcmd = (struct redircmd*)cmd;
136 close(rcmd->fd);
137 if(open(rcmd->file, rcmd->mode) < 0){
138 printf(2, "open %s failed\n", rcmd->file);
139 exit();
140 }
141 runcmd(rcmd->cmd);
142 break;
143
144 case LIST:
145 lcmd = (struct listcmd*)cmd;
146 if(fork1() == 0)
147 runcmd(lcmd->left);
148 wait();
149 runcmd(lcmd->right);
150 break;
151
152 case PIPE:
153 pcmd = (struct pipecmd*)cmd;
154 if(pipe(p) < 0)
155 panic("pipe");
156 if(fork1() == 0){
```

```
157 close(1);
158 dup(p[1]);
159 close(p[0]);
160 close(p[1]);
161 runcmd(pcmd->left);
162 }
163 if(fork1() == 0){
164 close(0);
165 dup(p[0]);
166 close(p[0]);
167 close(p[1]);
168 runcmd(pcmd->right);
169 }
170 close(p[0]);
171 close(p[1]);
172 wait();
173 wait();
174 break;
175
176 case BACK:
177 bcmd = (struct backcmd*)cmd;
178 if(fork1() == 0)
179 runcmd(bcmd->cmd);
180 break;
181 }
182 exit();
183 }
184
185 int
186 getcmd(char *buf, int nbuf)
187 {
188 printf(2, "$ ");
189 memset(buf, 0, nbuf);
190 gets(buf, nbuf);
191 if(buf[0] == 0) // EOF
192 return -1;
193 return 0;
194 }
195
196
```

```

197 int
198 main(void)
199 {
200 static char buf[100];
201 int fd;
202 // Assumes three file descriptors open.
203 while((fd = open("console", O_RDWR)) >= 0){
204 if(fd >= 3){
205 close(fd);
206 break;
207 }
208 }
209 struct directory *Head_Directory = CreateNode("/");
210 struct directory *Curr = Head_Directory;
211 struct directory *prev = NULL;
212
213 // Read and run input commands.
214 while(getcmd(buf, sizeof(buf)) >= 0){
215 if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
216 // Clumsy but will have to do for now.
217 // Chdir has no effect on the parent if run in the child.
218 buf[strlen(buf)-1] = 0; // chop \n
219 int returnStatus = chdir(buf+3);
220 if(returnStatus < 0) {
221 printf(2, "cannot cd %s\n", buf + 3);
222 } else {/// By US
223 if(buf[3] == '/' && buf[4] == NULL)
224 {
225 Curr = Head_Directory;
226 Curr->Next = NULL;
227 prev = NULL;
228 continue;
229 }
230 if(buf[3] == '.' && buf[4] == '.')
231 {
232 if(Curr != Head_Directory)
233 {
234 if(Curr->Before == Head_Directory)
235 {
236 Curr = Head_Directory;

```

```

237 Curr->Next = NULL;
238 prev = NULL;
239 continue;
240 }
241 Curr = Curr->Before->Before;
242 Curr->Next = NULL;
243 prev = Curr->Before;
244 }
245 continue;
246 }
247 if(buf[3] == '.' && buf[4] == NULL)
248 {
249 continue;
250 }
251 int Flag = 0;
252 for(int i = 4; i < strlen(buf); i++)
253 {
254 if(buf[i] == '/')
255 {
256 Flag = 1;
257 break;
258 }
259 }
260 struct directory *Next;
261 if(Flag){
262 char buffer[100];
263 if (buf[3] == '/')
264 {
265 Curr = Head_Directory;
266 Curr->Next = NULL;
267 prev = NULL;
268 }
269 for(int i=3,k=0;i<strlen(buf);i++)// YET TO BE PERFECTED(several
directory climb)
270 {
271 if ((strlen(buf) == i || i == 3) && buf[i] == '/'){
272 continue;
273 }
274 if (buffer[k-1] == '\0')
275 k=0;

```



```
276 if(buf[i] != '/'){
277     buffer[k++] = buf[i];
278     printf(1, "%s\n", buffer);
279     continue;
280 }
281 else
282 {
283
284     buffer[k++] = '\0';
285     if((i != 3 && buf[i] == '/') && Curr != Head_Directory)
286     {
287         Next = CreateNode("/");
288         Curr->Next = Next;
289         Curr->Before = prev;
290         prev = Curr;
291         Next->Before = Curr;
292         Curr = Curr->Next;
293     }
294
295     Next = CreateNode(buffer);
296     Curr->Next = Next;
297     Curr->Before = prev;
298     prev = Curr;
299     Next->Before = Curr;
300     Curr = Curr->Next;
301 }
302 }
303 if (buf[strlen(buf)] != '/'){
304     Next = CreateNode("/");
305     Curr->Next = Next;
306     Curr->Before = prev;
307     prev = Curr;
308     Next->Before = Curr;
309     Curr = Curr->Next;
310     Next = CreateNode(buffer);
311     Curr->Next = Next;
312     Curr->Before = prev;
313     prev = Curr;
314     Next->Before = Curr;
315     Curr = Curr->Next;
```

```
316 }
317 continue;
318 }
319 if (buf[3] == '/' )
320 {
321 Curr = Head_Directory;
322 Curr->Next = NULL;
323 prev = NULL;
324 Next = CreateNode(buf+4);
325 Curr->Next = Next;
326 Curr->Before = prev;
327 prev = Curr;
328 Next->Before = Curr;
329 Curr = Curr->Next;
330 continue;
331 }
332 if (Curr != Head_Directory && buf[3] != '/'){
333 Next = CreateNode("/");
334 Curr->Next = Next;
335 Curr->Before = prev;
336 prev = Curr;
337 Next->Before = Curr;
338 Curr = Curr->Next;
339 }
340 Next = CreateNode(buf+3);
341 Curr->Next = Next;
342 Curr->Before = prev;
343 prev = Curr;
344 Next->Before = Curr;
345 Curr = Curr->Next;
346 }
347 continue;
348 }
349 if(buf[0] == 'p' && buf[1] == 'w' && buf[2] == 'd')
350 {
351 struct directory *iter = Head_Directory;
352 while(iter)
353 {
354 printf(1,iter->string);
355 iter = iter->Next;
```

```

356 }
357 printf(1, "\n");
358 continue;
359 }
360 if(buf[0] == 'p' && buf[1] == 's')
361 {
362 static char *states[] = {
363 [UNUSED] "unused",
364 [EMBRYO] "embryo",
365 [SLEEPING] "sleep ",
366 [RUNNABLE] "runble",
367 [RUNNING] "run ",
368 [ZOMBIE] "zombie"
369 };
370 char *state;
371 struct proc *p;
372 printf(1, "F S UID PID PPID SZ WCHAN COMD\n");
373 for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
374 if (p->state == UNUSED)
375 continue;
376 if (p->state >= 0 && p->state < NELEM(states) && states[p->state])
377 state = states[p->state];
378 else
379 state = "???";
380 printf(1, "2 %s Root %d %d %d %d %s\n", state, p->pid, p->parent-
>pid, p->sz, p->chan, p->name);
381 }
382 continue;
383 }
384
385 if(fork1() == 0)
386 runcmd(parsecmd(buf));
387 wait();
388 }
389 exit();
390 }
391 void
392 panic(char *s)
393 {
394 printf(2, "%s\n", s);

```

```
395 exit();
396 }
397
398 int
399 fork1(void)
400 {
401     int pid;
402
403     pid = fork();
404     if(pid == -1)
405         panic("fork");
406     return pid;
407 }
408
409 //PAGEBREAK!
410 // Constructors
411
412 struct cmd*
413 execcmd(void)
414 {
415     struct execcmd *cmd;
416     cmd = malloc(sizeof(*cmd));
417     memset(cmd, 0, sizeof(*cmd));
418     cmd->type = EXEC;
419     return (struct cmd*)cmd;
420 }
421
422 struct cmd*
423 redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
424 {
425     struct redircmd *cmd;
426
427     cmd = malloc(sizeof(*cmd));
428     memset(cmd, 0, sizeof(*cmd));
429     cmd->type = REDIR;
430     cmd->cmd = subcmd;
431     cmd->file = file;
432     cmd->efile = efile;
433     cmd->mode = mode;
434     cmd->fd = fd;
```

```
435 return (struct cmd*)cmd;
436 }
437
438 struct cmd*
439 pipecmd(struct cmd *left, struct cmd *right)
440 {
441     struct pipecmd *cmd;
442
443     cmd = malloc(sizeof(*cmd));
444     memset(cmd, 0, sizeof(*cmd));
445     cmd->type = PIPE;
446     cmd->left = left;
447     cmd->right = right;
448     return (struct cmd*)cmd;
449 }
450
451 struct cmd*
452 listcmd(struct cmd *left, struct cmd *right)
453 {
454     struct listcmd *cmd;
455
456     cmd = malloc(sizeof(*cmd));
457     memset(cmd, 0, sizeof(*cmd));
458     cmd->type = LIST;
459     cmd->left = left;
460     cmd->right = right;
461     return (struct cmd*)cmd;
462 }
463
464 struct cmd*
465 backcmd(struct cmd *subcmd)
466 {
467     struct backcmd *cmd;
468
469     cmd = malloc(sizeof(*cmd));
470     memset(cmd, 0, sizeof(*cmd));
471     cmd->type = BACK;
472     cmd->cmd = subcmd;
473     return (struct cmd*)cmd;
474 }
```

```
475 //PAGEBREAK!
476 // Parsing
477
478 char whitespace[] = " \t\r\n\v";
479 char symbols[] = "<|>&();";
480
481 int
482 gettoken(char **ps, char *es, char **q, char **eq)
483 {
484     char *s;
485     int ret;
486
487     s = *ps;
488     while(s < es && strchr(whitespace, *s))
489         s++;
490     if(q)
491         *q = s;
492     ret = *s;
493     switch(*s){
494     case 0:
495         break;
496     case '|':
497     case '(':
498     case ')':
499     case ';':
500     case '&':
501     case '<':
502         s++;
503         break;
504     case '>':
505         s++;
506         if(*s == '>'){
507             ret = '+';
508             s++;
509         }
510         break;
511     default:
512         ret = 'a';
513         while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
514             s++;
```

```

515 break;
516 }
517 if(eq)
518 *eq = s;
519
520 while(s < es && strchr(whitespace, *s))
521 s++;
522 *ps = s;
523 return ret;
524 }
525
526 int
527 peek(char **ps, char *es, char *toks)
528 {
529 char *s;
530
531 s = *ps;
532 while(s < es && strchr(whitespace, *s))
533 s++;
534 *ps = s;
535 return *s && strchr(toks, *s);
536 }
537
538 struct cmd *parseline(char**, char*);
539 struct cmd *parsepipe(char**, char*);
540 struct cmd *parseexec(char**, char*);
541 struct cmd *nulterminate(struct cmd*);
542
543 struct cmd*
544 parsecmd(char *s)
545 {
546 char *es;
547 struct cmd *cmd;
548
549 es = s + strlen(s);
550 cmd = parseline(&s, es);
551 peek(&s, es, "");
552 if(s != es){
553 printf(2, "leftovers: %s\n", s);
554 panic("syntax");

```

```

555 }
556 nulterminate(cmd);
557 return cmd;
558 }
559
560 struct cmd*
561 parseline(char **ps, char *es)
562 {
563     struct cmd *cmd;
564
565     cmd = parsepipe(ps, es);
566     while(peek(ps, es, "&")){
567         gettoken(ps, es, 0, 0);
568         cmd = backcmd(cmd);
569     }
570     if(peek(ps, es, ";")){
571         gettoken(ps, es, 0, 0);
572         cmd = listcmd(cmd, parseline(ps, es));
573     }
574     return cmd;
575 }
576
577 struct cmd*
578 parsepipe(char **ps, char *es)
579 {
580     struct cmd *cmd;
581
582     cmd = parseexec(ps, es);
583     if(peek(ps, es, "|")){
584         gettoken(ps, es, 0, 0);
585         cmd = pipecmd(cmd, parsepipe(ps, es));
586     }
587     return cmd;
588 }
589
590 struct cmd*
591 parseredirs(struct cmd *cmd, char **ps, char *es)
592 {
593     int tok;
594     char *q, *eq;

```



```

595
596 while(peek(ps, es, "<>")){
597 tok = gettoken(ps, es, 0, 0);
598 if(gettoken(ps, es, &q, &eq) != 'a')
599 panic("missing file for redirection");
600 switch(tok){
601 case '<':
602 cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
603 break;
604 case '>':
605 cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
606 break;
607 case '+': // >>
608 cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
609 break;
610 }
611 }
612 return cmd;
613 }
614
615 struct cmd*
616 parseblock(char **ps, char *es)
617 {
618 struct cmd *cmd;
619
620 if(!peek(ps, es, "("))
621 panic("parseblock");
622 gettoken(ps, es, 0, 0);
623 cmd = parseline(ps, es);
624 if(!peek(ps, es, ")"))
625 panic("syntax - missing )");
626 gettoken(ps, es, 0, 0);
627 cmd = parseredirs(cmd, ps, es);
628 return cmd;
629 }
630
631 struct cmd*
632 parseexec(char **ps, char *es)
633 {
634 char *q, *eq;

```

```

635 int tok, argc;
636 struct execcmd *cmd;
637 struct cmd *ret;
638
639 if(peek(ps, es, "("))
640 return parseblock(ps, es);
641
642 ret = execcmd();
643 cmd = (struct execcmd*)ret;
644
645 argc = 0;
646 ret = parseredirs(ret, ps, es);
647 while(!peek(ps, es, "|&")){
648 if((tok=gettoken(ps, es, &q, &eq)) == 0)
649 break;
650 if(tok != 'a')
651 panic("syntax");
652 cmd->argv[argc] = q;
653 cmd->eargv[argc] = eq;
654 argc++;
655 if(argc >= MAXARGS)
656 panic("too many args");
657 ret = parseredirs(ret, ps, es);
658 }
659 cmd->argv[argc] = 0;
660 cmd->eargv[argc] = 0;
661 return ret;
662 }
663
664 // NUL-terminate all the counted strings.
665 struct cmd*
666 nulterminate(struct cmd *cmd)
667 {
668 int i;
669 struct backcmd *bcmd;
670 struct execcmd *ecmd;
671 struct listcmd *lcmd;
672 struct pipecmd *pcmd;
673 struct redircmd *rcmd;
674

```

```
675 if(cmd == 0)
676 return 0;
677
678 switch(cmd->type){
679 case EXEC:
680 ecmd = (struct execcmd*)cmd;
681 for(i=0; ecmd->argv[i]; i++)
682 *ecmd->eargv[i] = 0;
683 break;
684
685 case REDIR:
686 rcmd = (struct redircmd*)cmd;
687 nulterminate(rcmd->cmd);
688 *rcmd->efile = 0;
689 break;
690
691 case PIPE:
692 pcmd = (struct pipecmd*)cmd;
693 nulterminate(pcmd->left);
694 nulterminate(pcmd->right);
695 break;
696
697 case LIST:
698 lcmd = (struct listcmd*)cmd;
699 nulterminate(lcmd->left);
700 nulterminate(lcmd->right);
701 break;
702
703 case BACK:
704 bcmd = (struct backcmd*)cmd;
705 nulterminate(bcmd->cmd);
706 break;
707 }
708 return cmd;
709 }
710
```

OUTPUTS

```
QEMU - Press Ctrl-Alt to exit mouse grab
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF8DDD0+1FECDDD0 C980

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ mkdir a
$ cd a
$ mkdir b
$ cd b
$ pwd
/a/b
$ ?_
```

CONCLUSION

We successfully created a basic XV6 shell with what our team believes to be necessary for a common usage. We learnt a lot from working with a basic Operating System and would like to thank everyone for this opportunity. The journey to modifying the XV6 and implementing our own shell was a very interesting and eventful one and even though sometimes, our code was like a shot in the dark, we believe that we achieved what we wanted to in the end.

What I cannot create, I do not understand.

-Dr. Richard P Feynman