

IMPLEMENTATION OF SHELL, EDITOR, **COMMAND,**
SYSTEM CALL, AND ENHANCEMENT IN XV6

A SKILLING PROJECT REPORT

Submitted towards the professional course

20CS2103S Operating Systems Design

Sri Teja Chadalawada

BACHELOR OF TECHNOLOGY

Branch: Computer Science and Engineering



NOVEMBER 2021

Department of Computer Science and Engineering

K L Deemed to be University,

Green Fields, Vaddeswaram,

Guntur District, A.P., 522 502.

TABLE OF CONTENTS

CHAPTER NO.	TITLE
1	What is XV6?
2	SRS
3	Data Flow Diagrams
4	Modules Description
5	Codes with output
6	Conclusion

WHAT IS XV6?

Xv6 is a teaching operating system developed in the summer of 2006 for MIT's operating systems course, [6.828: Operating System Engineering](#). We hope that xv6 will be useful in other courses too. This page collects resources to aid the use of xv6 in other courses, including a commentary on the source code itself.

History and Background

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.

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

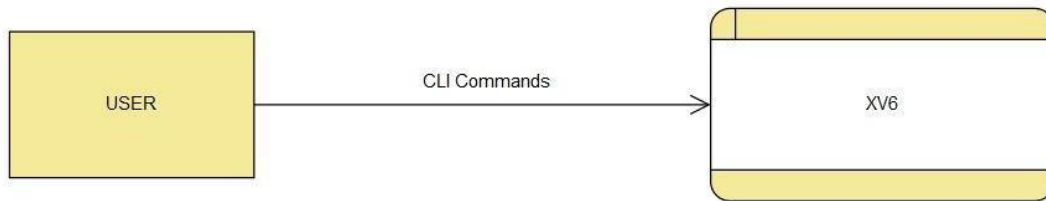


TABLE DESCRIPTIONS

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.

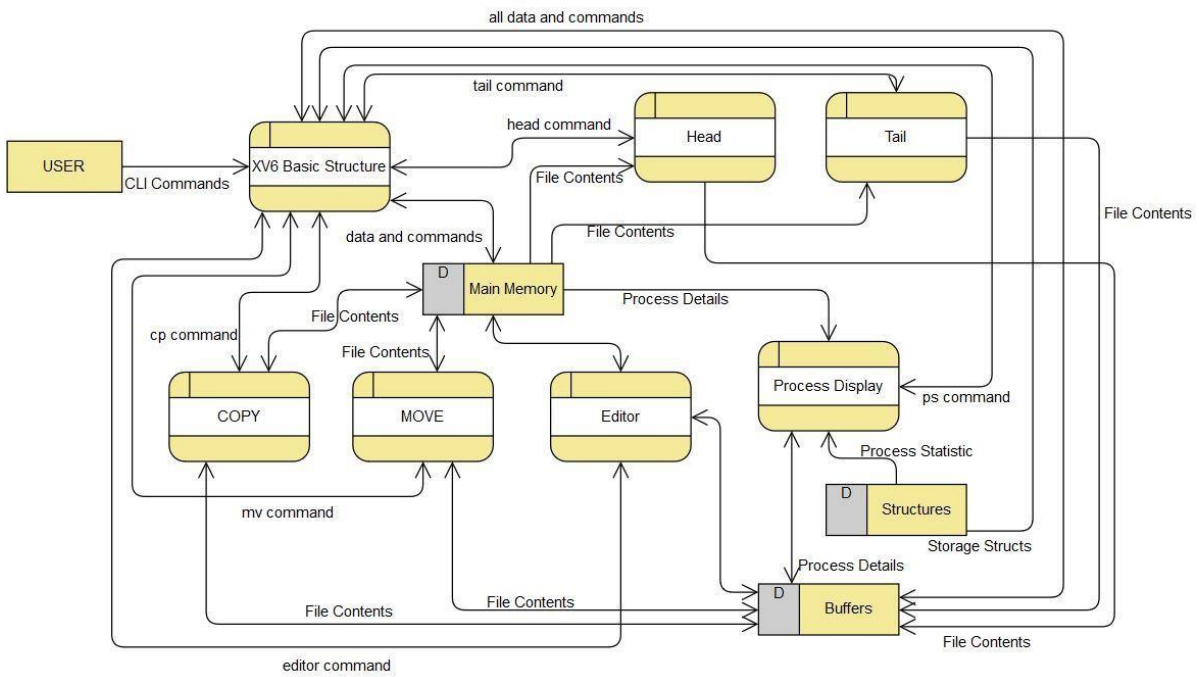
Buffers

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.

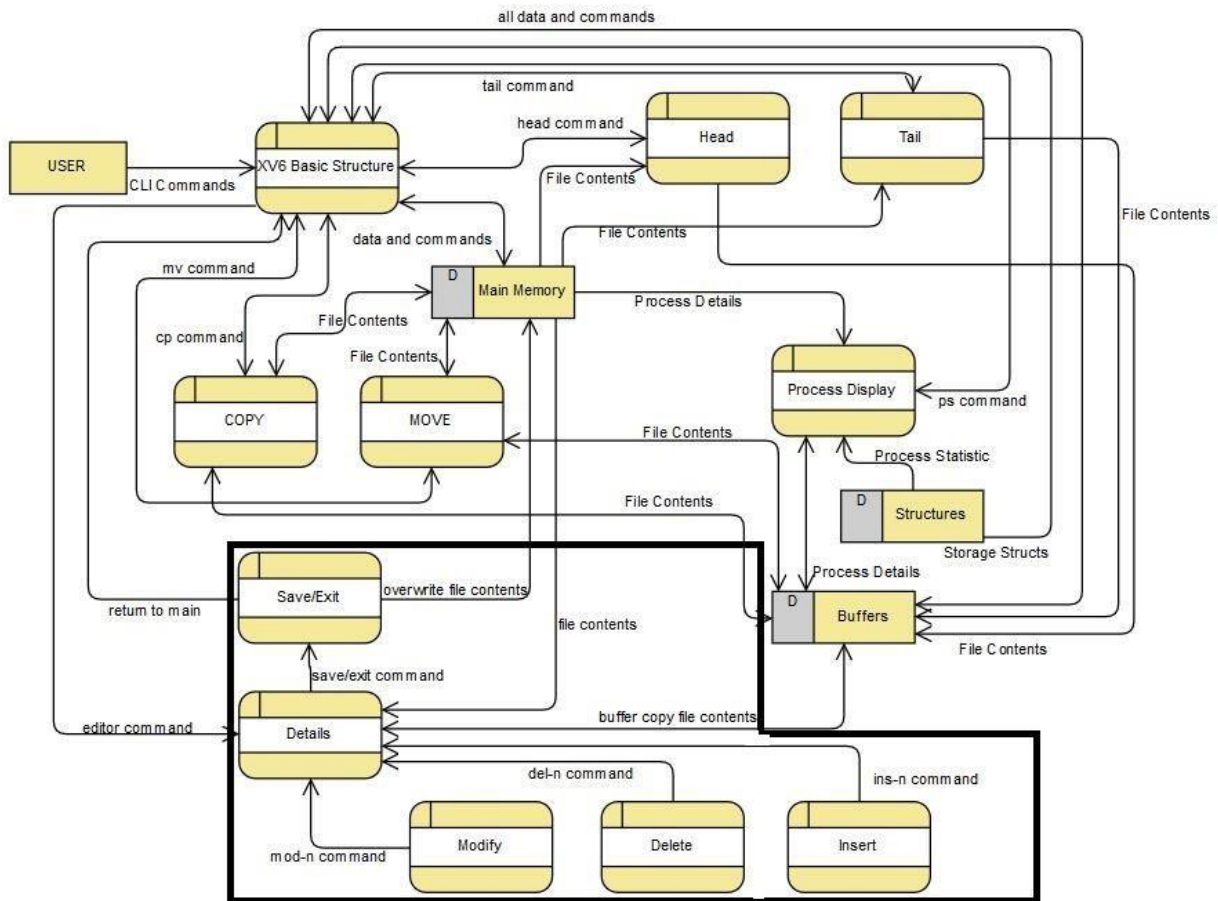
Structures

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

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

```
osd-2000030957@team-osd-~/xv6
SeaBIOS (version 1.11.0-2.el17)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 FmP FMM+1FF94780+1FED4780 C980

Booting from Hard Disk..xv6...
cpu0: starting 1
cpu0: starting 0
sh: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 hmap start 58
init: starting sh
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 14544
echo       2 4 13396
forktest   2 5 8220
grep       2 6 16080
init       2 7 14288
kill       2 8 13428
ln         2 9 13368
ls         2 10 16224
mkdir      2 11 13456
rm         2 12 13432
sh         2 13 24864
stressfs   2 14 14384
usertests  2 15 67280
wc         2 16 15204
zombie     2 17 13092
mycat      2 18 13640
myfork     2 19 13504
myccp      2 20 14144
ps         2 21 12960
dpro       2 22 13996
nice       2 23 13620
console    3 24 0
$ mkdir sss
$ cd sss
$ mkdir ddd
```

EDITOR

CODE

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
#include "fs.h"
#define BUF_SIZE 256
#define MAX_LINE_NUMBER 256
#define MAX_LINE_LENGTH 256
#define NULL 0
char* strcat_n(char* dest, char* src, int len);
int get_line_number(char *text[]);
void show_text(char *text[]);
void com_ins(char *text[], int n, char *extra);
void com_mod(char *text[], int n, char *extra);
void com_del(char *text[], int n);
void com_help(char *text[]);
void com_save(char *text[], char *path);
void com_exit(char *text[], char *path);
int stringtonumber(char* src);
int changed = 0;
int auto_show = 1;
int main(int argc, char *argv[])
{
    if (argc == 1)
    {
        printf(1, "please input the command as [editor file_name]\n");
        exit();
    }
    char *text[MAX_LINE_NUMBER] = {};
    text[0] = malloc(MAX_LINE_LENGTH);
    memset(text[0], 0, MAX_LINE_LENGTH);
    int line_number = 0;
    int fd = open(argv[1], O_RDONLY);
    if (fd != -1)
    {
        printf(1, "file exist\n");
        char buf[BUF_SIZE] = {};
        int len = 0;
        while ((len = read(fd, buf, BUF_SIZE)) > 0)
        {
            int i = 0;
            int next = 0;
```



```

int is_full = 0;
while (i < len)
{
    for (i = next; i < len && buf[i] != '\n'; i++);
    strcat_n(text[line_number], buf+next, i-next);
    if (i < len && buf[i] == '\n')
    {
        if (line_number >= MAX_LINE_NUMBER - 1)
            is_full = 1;
        else
        {
            line_number++;
            text[line_number] = malloc(MAX_LINE_LENGTH);
            memset(text[line_number], 0, MAX_LINE_LENGTH);
        }
    }
    if (is_full == 1 || i >= len - 1)
        break;
    else
        next = i + 1;
}
if (is_full == 1)
    break;
}
close(fd);
} else{
    printf(1, "File do not exist\n");
    exit();
}
show_text(text);
com_help(text);
char input[MAX_LINE_LENGTH] = {};
while (1)
{
    printf(1, "\nplease input command:\n");
    memset(input, 0, MAX_LINE_LENGTH);
    gets(input, MAX_LINE_LENGTH);
    int len = strlen(input);
    input[len-1] = '\0';
    len--;
    int pos = MAX_LINE_LENGTH - 1;
    int j = 0;
    for (; j < 8; j++)
    {
        if (input[j] == ' ')
        {
            pos = j + 1;

```

```

break;
}
}
if (input[0] == 'i' && input[1] == 'n' && input[2] == 's')
{
if (input[3] == '-'&&stringtonumber(&input[4])>=0)
{
com_ins(text, stringtonumber(&input[4]), &input[pos]);

line_number = get_line_number(text);
}
else if(input[3] == ' ' | input[3] == '\0')
{
com_ins(text, line_number+1, &input[pos]);
line_number = get_line_number(text);
}
else
{
printf(1, "invalid command.\n");
com_help(text);
}
}

else if (input[0] == 'm' && input[1] == 'o' && input[2] == 'd')
{
if (input[3] == '-'&&stringtonumber(&input[4])>=0)
com_mod(text, atoi(&input[4]), &input[pos]);
else if(input[3] == ' ' | input[3] == '\0')
com_mod(text, line_number + 1, &input[pos]);
else
{
printf(1, "invalid command.\n");
com_help(text);
}
}

else if (input[0] == 'd' && input[1] == 'e' && input[2] == 'l')
{
if (input[3] == '-'&&stringtonumber(&input[4])>=0)
{
com_del(text, atoi(&input[4]));

line_number = get_line_number(text);
}
else if(input[3]=='\0')
{
com_del(text, line_number + 1);
line_number = get_line_number(text);
}
}

```

```

}
else
{
printf(1, "invalid command.\n");
com_help(text);
}

}
else if (strcmp(input, "show") == 0)
{
auto_show = 1;
printf(1, "enable show current contents after text changed.\n");
}
else if (strcmp(input, "hide") == 0)
{
auto_show = 0;
printf(1, "disable show current contents after text changed.\n");
}
else if (strcmp(input, "help") == 0)
com_help(text);
else if (strcmp(input, "save") == 0 || strcmp(input, "CTRL+S\n") == 0)
com_save(text, argv[1]);
else if (strcmp(input, "exit") == 0)
com_exit(text, argv[1]);
else
{
printf(1, "invalid command.\n");
com_help(text);
}
}
exit();
}
char* strcat_n(char* dest, char* src, int len)
{
if (len <= 0)
return dest;
int pos = strlen(dest);
if (len + pos >= MAX_LINE_LENGTH)
return dest;
int i = 0;
for (; i < len; i++)
dest[i+pos] = src[i];
dest[len+pos] = '\0';
return dest;
}
void show_text(char *text[])
{

```

```

printf(1, "*****\n");
printf(1, "the contents of the file are:\n");
int j = 0;
for (; text[j] != NULL; j++)
printf(1, "%d%d%d:%s\n", (j+1)/100, ((j+1)%100)/10, (j+1)%10, text[j]);
}
int get_line_number(char *text[])
{
int i = 0;
for (; i < MAX_LINE_NUMBER; i++)
if (text[i] == NULL)
return i - 1;
return i - 1;
}
int stringtonumber(char* src)
{
int number = 0;
int i=0;
int pos = strlen(src);
for(;i<pos;i++)
{
if(src[i]==' ') break;
if(src[i]>57 || src[i]<48) return -1;
number=10*number+(src[i]-48);
}
return number;
}
void com_ins(char *text[], int n, char *extra)
{
if (n < 0 || n > get_line_number(text) + 1)
{
printf(1, "invalid line number\n");
return;
}
char input[MAX_LINE_LENGTH] = {};
if (*extra == '\0')
{
printf(1, "please input content:\n");
gets(input, MAX_LINE_LENGTH);
input[strlen(input)-1] = '\0';
}
else
strcpy(input, extra);
int i = MAX_LINE_NUMBER - 1;
for (; i > n; i--)
{
if (text[i-1] == NULL)

```

```

continue;
else if (text[i] == NULL && text[i-1] != NULL)
{
text[i] = malloc(MAX_LINE_LENGTH);
memset(text[i], 0, MAX_LINE_LENGTH);
strcpy(text[i], text[i-1]);
}
else if (text[i] != NULL && text[i-1] != NULL)
{
memset(text[i], 0, MAX_LINE_LENGTH);
strcpy(text[i], text[i-1]);
}
}
if (text[n] == NULL)
{
text[n] = malloc(MAX_LINE_LENGTH);
if (text[n-1][0] == '\0')
{
memset(text[n], 0, MAX_LINE_LENGTH);
strcpy(text[n-1], input);
changed = 1;
if (auto_show == 1)
show_text(text);
return;
}
}
memset(text[n], 0, MAX_LINE_LENGTH);
strcpy(text[n], input);
changed = 1;
if (auto_show == 1)
show_text(text);
}
void com_mod(char *text[], int n, char *extra)
{
if (n <= 0 || n > get_line_number(text) + 1)
{
printf(1, "invalid line number\n");
return;
}
char input[MAX_LINE_LENGTH] = {};
if (*extra == '\0')
{
printf(1, "please input content:\n");
gets(input, MAX_LINE_LENGTH);
input[strlen(input)-1] = '\0';
}
else

```

```

strcpy(input, extra);
memset(text[n-1], 0, MAX_LINE_LENGTH);
strcpy(text[n-1], input);
changed = 1;
if (auto_show == 1)
show_text(text);
}
void com_del(char *text[], int n)
{
if (n <= 0 || n > get_line_number(text) + 1)
{
printf(1, "invalid line number\n");
return;
}
memset(text[n-1], 0, MAX_LINE_LENGTH);
int i = n - 1;
for (; text[i+1] != NULL; i++)
{
strcpy(text[i], text[i+1]);
memset(text[i+1], 0, MAX_LINE_LENGTH);
}
if (i != 0)
{
free(text[i]);
text[i] = 0;
}
changed = 1;
if (auto_show == 1)
show_text(text);
}
void com_help(char *text[])
{
printf(1, "*****\n");
printf(1, "instructions for use:\n");
printf(1, "ins-n, insert a line after line n\n");
printf(1, "mod-n, modify line n\n");
printf(1, "del-n, delete line n\n");
printf(1, "ins, insert a line after the last line\n");
printf(1, "mod, modify the last line\n");
printf(1, "del, delete the last line\n");
printf(1, "show, enable show current contents after executing a command.\n");
printf(1, "hide, disable show current contents after executing a command.\n");
printf(1, "save, save the file\n");
printf(1, "exit, exit editor\n");
}
void com_save(char *text[], char *path)
{

```

```

unlink(path);
int fd = open(path, O_WRONLY|O_CREATE);
if (fd == -1)
{
printf(1, "save failed, file can't open:\n");
exit();
}
if (text[0] == NULL)
{
close(fd);
return;
}
write(fd, text[0], strlen(text[0]));
int i = 1;
for (; text[i] != NULL; i++)
{
printf(fd, "\n");
write(fd, text[i], strlen(text[i]));
}
close(fd);
printf(1, "saved successfully\n");
changed = 0;
return;
}
void com_exit(char *text[], char *path)
{
while (changed == 1)
{
printf(1, "save the file? y/n\n");
char input[MAX_LINE_LENGTH] = {};
gets(input, MAX_LINE_LENGTH);
input[strlen(input)-1] = '\0';
if (strcmp(input, "y") == 0)
com_save(text, path);
else if (strcmp(input, "n") == 0)
break;
else
printf(2, "wrong answer?\n");
}
int i = 0;
for (; text[i] != NULL; i++)
{
free(text[i]);
text[i] = 0;
}
exit();
}

```

OUTPUTS

```
osd-2000030957@team-osd:~/xv6
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 14544
echo      2 4 13396
forktest  2 5 8220
grep      2 6 16080
init       2 7 14288
kill       2 8 13428
ln         2 9 13368
ls         2 10 16224
mkdir     2 11 13456
rm         2 12 13432
sh         2 13 24864
stressfs  2 14 14384
usertests  2 15 67280
wc         2 16 15204
zombie    2 17 13092
mycat     2 18 13640
myfork    2 19 13504
myccp     2 20 14144
ps         2 21 12960
dpro      2 22 13996
nice       2 23 13620
xv6editor  2 24 26852
console   3 25 0
$ cat>editorfile
this is a sample file for editor program
$ xv6editor editorfile
file exist
*****
the contents of the file are:
001:this is a sample file for editor program
002:
*****
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
show, enable show current contents after executing a command.
hide, disable show current contents after executing a command.
save, save the file
exit, exit editor
please input command:
```


Square

CODE

Adding square in “syscall.c”

```
osd-2000030957@team-osd:~/xv6bc
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_square(void);

static int (*syscalls[]) (void) = {
[SYS_fork]      sys_fork,
[SYS_exit]     sys_exit,
[SYS_wait]     sys_wait,
[SYS_pipe]     sys_pipe,
[SYS_read]     sys_read,
[SYS_kill]     sys_kill,
[SYS_exec]     sys_exec,
[SYS_fstat]    sys_fstat,
[SYS_chdir]    sys_chdir,
[SYS_dup]      sys_dup,
[SYS_getpid]   sys_getpid,
[SYS_sbrk]     sys_sbrk,
[SYS_sleep]    sys_sleep,
[SYS_uptime]   sys_uptime,
[SYS_open]     sys_open,
[SYS_write]    sys_write,
[SYS_mknod]    sys_mknod,
[SYS_unlink]   sys_unlink,
[SYS_link]     sys_link,
[SYS_mkdir]    sys_mkdir,
[SYS_close]    sys_close,
[SYS_square]   sys_square,
};

void
```

Adding in “syscall.h”

osd-2000030957@team-osd:~/xv6bc

```
// System call numbers
#define SYS_fork    1
#define SYS_exit   2
#define SYS_wait   3
#define SYS_pipe   4
#define SYS_read   5
#define SYS_kill   6
#define SYS_exec   7
#define SYS_fstat  8
#define SYS_chdir  9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_square 22
```

Adding in “sysproc.c”

```
osd-2000030957@team-osd:~/xv6bc
int n;

if(argint(0, &n) < 0)
    return -1;
addr = myproc()->sz;
if(growproc(n) < 0)
    return -1;
return addr;
}

int
sys_sleep(void)
{
    int n;
    uint ticks0;

    if(argint(0, &n) < 0)
        return -1;
    acquire(&tickslock);
    ticks0 = ticks;
    while(ticks - ticks0 < n){
        if(myproc()->killed){
            release(&tickslock);
            return -1;
        }
        sleep(ticks, &tickslock);
    }
    release(&tickslock);
    return 0;
}

// return how many clock tick interrupts have occurred
// since start.
int
sys_uptime(void)
{
    uint xticks;

    acquire(&tickslock);
    xticks = ticks;
    release(&tickslock);
    return xticks;
}

int sys_square(void) {
    int num;
    argptr(0, (void *)&num, sizeof(num)); // extract and store argument into num
    return num * num;
}
```

Adding in “user.h”

```
osd-2000030957@team-osd:~/xv6bc
struct stat;
struct rtcdate;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int square(int);
```

Adding in “useys.S”

```
osd-2000030957@team-osd:~/xv6bc
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
    .globl name; \
    name: \
        movl $SYS_ ## name, %eax; \
        int $T_SYSCALL; \
        ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(square)
~
~
~
```

vi square.c

```
osd-2000030957@team-osd:~/xv6bc
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(void)
{
    printf(1, "square of numbers 7 is %d\n", square(7));
    exit();
}
~
~
~
~
~
~
~
~
~
~
~
~
~
```

OUTPUT:

```
osd-2000030957@team-osd:~/xv6bc
SeaBIOS (version 1.11.0-2.el7)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF94780+1FED4780 C980

Booting from Hard Disk..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
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 14496
echo       2 4 13352
forktest  2 5 8184
grep       2 6 16032
init       2 7 14244
kill       2 8 13380
ln         2 9 13328
ls         2 10 16184
mkdir      2 11 13416
rm         2 12 13392
sh         2 13 24824
stressfs   2 14 14336
usertests  2 15 67240
wc         2 16 15164
zombie     2 17 13052
square     2 18 13176
mycat      2 19 13600
myccp      2 20 14100
myfork     2 21 13464
mycp       2 22 14164
mywcUser   2 23 14420
console    3 24 0
$ square
square of numbers 7 is 49
$
```

Date.c


Code:

```
#include "types.h"
#include "user.h"
#include "date.h"
int
main(int argc, char *argv[])
{
    struct rtcdate r;
    if (date(&r)) {
        printf(2, "date failed\n");
        exit();
    }
    printf(1, "%d-%d-%d %d:%d:%d\n", r.year, r.month, r.day, r.hour,
        r.minute, r.second);
    exit();
}
```

 osd-2000030957@team-osd:~/xv6

```
#include "types.h"
#include "user.h"
#include "date.h"
int
main(int argc, char *argv[])
{
    struct rtcdate r;
    if (date(&r)) {
        printf(2, "date failed\n");
        exit();
    }
    printf(1, "%d-%d-%d %d:%d:%d\n", r.year, r.month, r.day, r.hour, r.minute, r.second);
    exit();
}
```

OUTPUT:

 osd-2000030957@team-osd:~/xv6

SeaBIOS (version 1.11.0-2.el7)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF94780+1FED4780 C980

Booting from Hard Disk..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

\$ ls

.	1	1	512
..	1	1	512
README	2	2	2286
cat	2	3	14564
echo	2	4	13416
forktest	2	5	8248
grep	2	6	16100
init	2	7	14308
kill	2	8	13448
ln	2	9	13392
ls	2	10	16252
mkdir	2	11	13476
rm	2	12	13460
sh	2	13	24892
stressfs	2	14	14404
usertests	2	15	67304
wc	2	16	15224
zombie	2	17	13116
mycat	2	18	13660
myfork	2	19	13528
myccp	2	20	14164
ps	2	21	12984
dpro	2	22	14016
nice	2	23	13640
xv6editor	2	24	26876
xv6date	2	25	13660
console	3	26	0

\$ xv6date

2021-10-22 15:10:26

FCFS

CODE

osd-2000030957@team-osd:~

```
#include <stdio.h>
typedef struct fcfs
{
    int process;
    int burst;
    int arrival;
    int tat;
    int wt;
}fcfs;
int sort(fcfs [], int);
int main()
{
    int n, i, temp = 0, AvTat = 0, AvWt = 0;
    printf ("Enter the number of processes: ");
    scanf ("%d", &n);
    fcfs arr[n];
    int tct[n];
    for (i = 0; i < n; i++)
    {
        arr[i].process = i;
        printf ("Enter the process %d data\n", arr[i].process);
        printf ("Enter CPU Burst: ");
        scanf ("%d", &(arr[i].burst));
        printf ("Enter the arrival time: ");
        scanf ("%d", &(arr[i].arrival));
    }
    sort(arr, n);
    printf ("Process\t\tBurst Time\tArrival Time\tTurn Around Time\tWaiting Time\n");
    for (i = 0; i < n; i++)
    {
        tct[i] = temp + arr[i].burst;
        temp = tct[i];
        arr[i].tat = tct[i] - arr[i].arrival;
        arr[i].wt = arr[i].tat - arr[i].burst;
        AvTat = AvTat + arr[i].tat;
        AvWt = AvWt + arr[i].wt;
        printf ("%5d\t%15d\t\t%9d\t%12d\t%12d\n", arr[i].process, arr[i].burst, arr[i].arrival, arr[i].tat, arr[i].wt);
    }
    printf ("Average Turn Around Time: %d\nAverage Waiting Time: %d\n", AvTat / n, AvWt / n);
    return 0;
}

int sort(fcfs arr[], int n)
{
    int i, j;
    fcfs k;
    for (i = 0; i < n - 1; i++)
    {
        for (j = i + 1; j < n; j++)
        {
```


osd-2000030957@team-osd:~

```
{
int n, i, temp = 0, AvTat = 0, AvWt = 0;
printf ("Enter the number of processes: ");
scanf ("%d", &n);
fcfs arr[n];
int tct[n];
for (i = 0; i < n; i++)
{
arr[i].process = i;
printf ("Enter the process %d data\n", arr[i].process);
printf ("Enter CPU Burst: ");
scanf ("%d", &(arr[i].burst));
printf ("Enter the arrival time: ");
scanf ("%d", &(arr[i].arrival));
}
sort(arr, n);
printf ("Process\t\tBurst Time\tArrival Time\tTurn Around Time\tWaiting Time\n");
for (i = 0; i < n; i++)
{
tct[i] = temp + arr[i].burst;
temp = tct[i];
arr[i].tat = tct[i] - arr[i].arrival;
arr[i].wt = arr[i].tat - arr[i].burst;
AvTat = AvTat + arr[i].tat;
AvWt = AvWt + arr[i].wt;
printf ("%5d\t%15d\t%9d\t%12d\t%12d\n", arr[i].process, arr[i].burst, arr[i].arrival, arr[i].tat, arr[i].wt);
}
printf ("Average Turn Around Time: %d\nAverage Waiting Time: %d\n", AvTat / n, AvWt / n);
return 0;
}

int sort(fcfs arr[], int n)
{
int i, j;
fcfs k;
for (i = 0; i < n - 1; i++)
{
for (j = i + 1; j < n; j++)
{
if (arr[i].arrival > arr[j].arrival)
{
k = arr[i];
arr[i] = arr[j];
arr[j] = k;
}
}
}
return 0;
}
}
```

OUTPUT

```
[osd-2000030957@team-osd ~]$ vi myfcfs.c
[osd-2000030957@team-osd ~]$ cc myfcfs.c
[osd-2000030957@team-osd ~]$ ./a.out
Enter the number of processes: 5
Enter the process 0 data
Enter CPU Burst: 6
Enter the arrival time: 2
Enter the process 1 data
Enter CPU Burst: 2
Enter the arrival time: 5
Enter the process 2 data
Enter CPU Burst: 8
Enter the arrival time: 1
Enter the process 3 data
Enter CPU Burst: 3
Enter the arrival time: 0
Enter the process 4 data
Enter CPU Burst: 4
Enter the arrival time: 4


| Process | Burst Time | Arrival Time | Turn Around Time | Waiting Time |
|---------|------------|--------------|------------------|--------------|
| 3       | 3          | 0            | 3                | 0            |
| 2       | 8          | 1            | 10               | 2            |
| 0       | 6          | 2            | 15               | 9            |
| 4       | 4          | 4            | 17               | 13           |
| 1       | 2          | 5            | 18               | 16           |


Average Turn Around Time: 12
Average Waiting Time: 8
[osd-2000030957@team-osd ~]$
```

lseek

CODE

```
#include "types.h"
#include "user.h"
#include "fcntl.h"

#define MAX_SIZE 1024

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

    int fd, offset, length, i, j, k;
    char data[MAX_SIZE];
    char user_string[MAX_SIZE];

    if (argc < 5) {
        printf(1, "Usage : %s 'filename' 'offset' 'len' 'string'\n", argv[0]);
        exit();
    }

    fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        printf(1, "Error while opening the file\n");
        exit();
    }

    offset = atoi(argv[2]);
```

```
length = atoi(argv[3]);
```

```
lseek(fd, offset, SEEK_SET);
```

```
read(fd, data, length);
```

```
data[length] = '\0';
```

```
k = 0;
```

```
for (i = 4; i < argc; i++) {
```

```
    for (j = 0; argv[i][j] != '\0'; j++) {
```

```
        user_string[k++] = argv[i][j];
```

```
    }
```

```
    user_string[k++] = ' ';
```

```
}
```

```
user_string[k - 1] = '\0';
```

```
if (!(strcmp(data, user_string))) {
```

```
    printf(1, "Strings matched\n");
```

```
}
```

```
else {
```

```
    printf(1, "Strings did not matched\n");
```

```
}
```

```
close(fd);
```

```
exit();
```

```
}
```

OUTPUT

```
[osd-2000030957@team-osd ~]$ vi lseek.c
[osd-2000030957@team-osd ~]$ vi mylseek.c
[osd-2000030957@team-osd ~]$ cc mylseek.c
[osd-2000030957@team-osd ~]$ ./a.out
[osd-2000030957@team-osd ~]$ vi lseek.c
[osd-2000030957@team-osd ~]$ vi mylseek.c
[osd-2000030957@team-osd ~]$ cc lseek.c
[osd-2000030957@team-osd ~]$ ./a.out
1234567890abcdefghij[osd-2000030957@team-osd ~]$ cc mylseek.c
[osd-2000030957@team-osd ~]$ ./a.out
1234567890fghijxxxxx[osd-2000030957@team-osd ~]$
```

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.