IMPLEMENTATION OF SHELL, EDITOR, WC.C,FS.C AND CFS IN XV6

A SKILLING PROJECT REPORT

Submitted towards the professional course

19CS2106A Operating Systems Design

MOHAMMAD SAMEER (2000030639)

BACHELOROF TECHNOLOGY

Branch: Computer Science and Engineering

OCTOBER 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

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 200

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

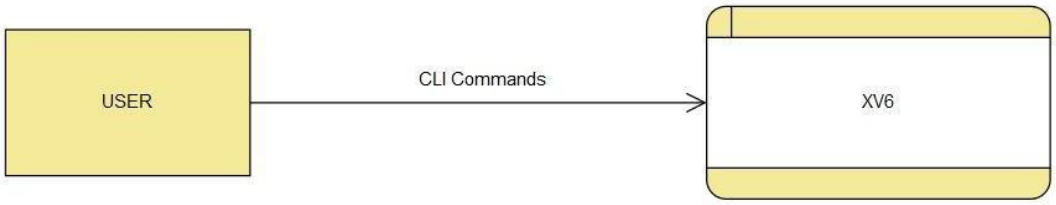


TABLE DESCRIPTIONS

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.

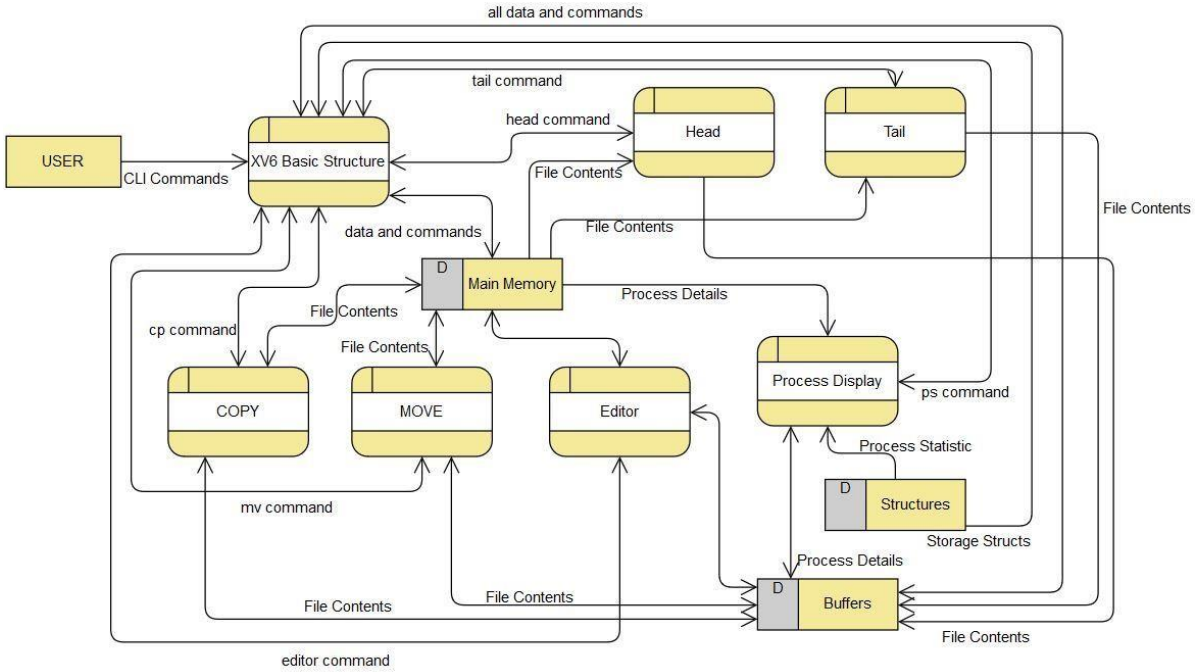
Buffers

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.

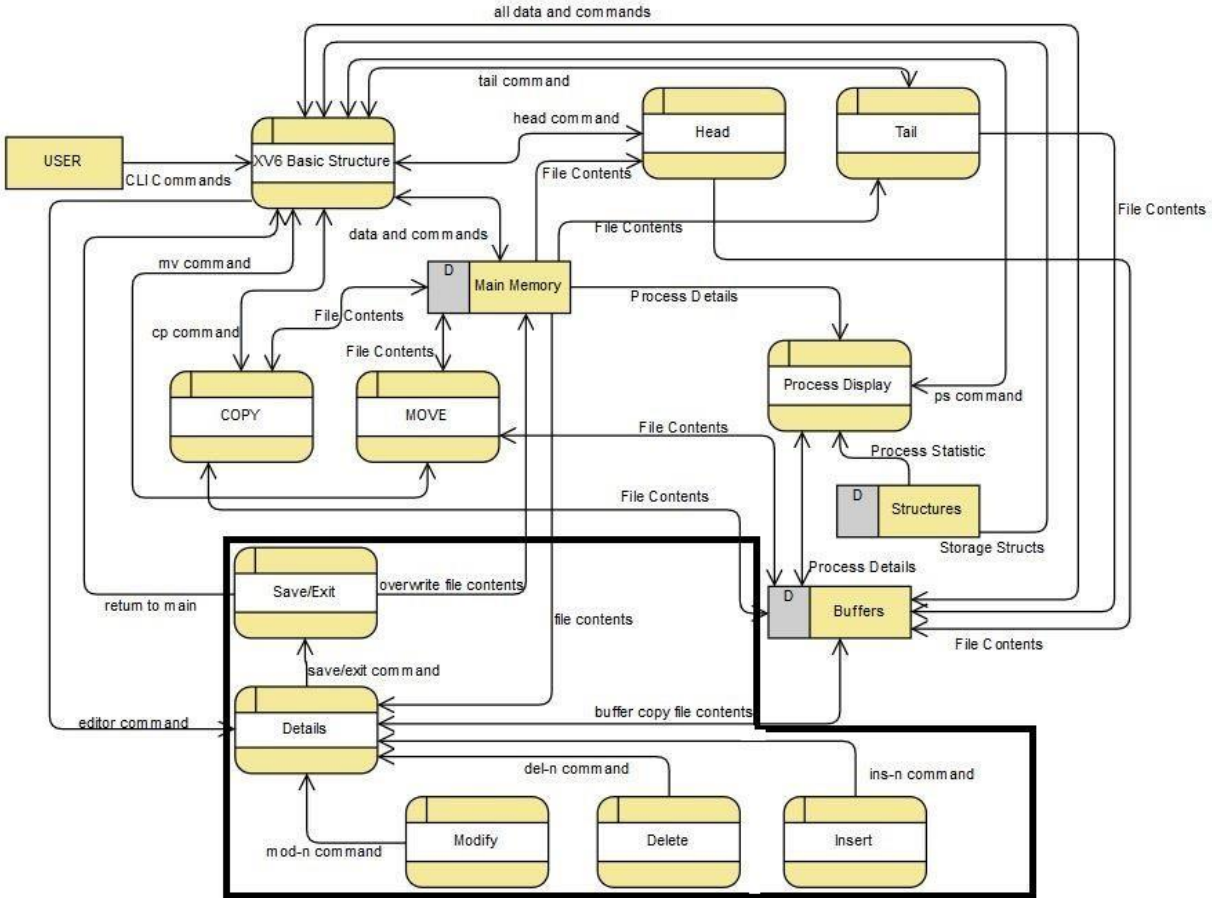
Structures

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


all data and commands
tail command
head command
Head
Tail
USER
CLI Commands
XV6 Basic Structure
File Contents
File Contents
data and commands
File Contents
D
Main Memory
Process Details
cp command
File Contents
File Contents
COPY
MOVE
Editor
Process Display
ps command
Process Statistic
D
Structures
Storage Structs
mv command
Process Details
D
Buffers
File Contents
File Contents
File Contents
editor command

Level 2 DFD


all data and commands
tail command
head command
Head
Tail
USER
CLI Commands
XV6 Basic Structure
File Contents
File Contents
data and commands
File Contents
mv command
D
Main Memory
Process Details
File Contents
cp command
File Contents
Process Display
ps command
COPY
MOVE
File Contents
Process Statistic
File Contents
D
Structures
Storage Structs
Process Details
Save/Exit
overwrite file contents
return to main
file contents
D
Buffers
File Contents
save/exit command
editor command
Details
buffer copy file contents
del-n command
ins-n command
mod-n command
Modify
Delete
Insert

MODULES DESCRIPTION

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.

LOCKING:

- xv6 runs on multiprocessors
- Computers with multiple CPUs executing independently
- These multiple CPUs share physical RAM, and xv6 exploits the sharing to maintain data structures that all CPUs read and write
- This sharing raises the possibility of one CPU reading a data structure while another CPU is mid-way through updating it

When multiple CPUs updating the same data simultaneously; without careful design such parallel access is likely to yield incorrect results or a broken data structure

Task-1:
Shell(sh.c)
Code:-

```
// Shell.

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

// Parsed command representation
#define EXEC  1
#define REDIR 2
#define PIPE  3
#define LIST  4
#define BACK  5

#define MAXARGS 10

struct cmd {
  int type;
};

struct execcmd {
  int type;
  char *argv[MAXARGS];
  char *eargv[MAXARGS];
};

struct redircmd {
  int type;
  struct cmd *cmd;
  char *file;
  char *efile;
  int mode;
  int fd;
};

struct pipecmd {
  int type;
  struct cmd *left;
```

```
  struct cmd *right;
};

struct listcmd {
  int type;
  struct cmd *left;
  struct cmd *right;
};

struct backcmd {
  int type;
  struct cmd *cmd;
};

int fork1(void);  // Fork but panics on failure.
void panic(char*);
struct cmd *parsecmd(char*);

// Execute cmd.  Never returns.
void
runcmd(struct cmd *cmd)
{
  int p[2];
  struct backcmd *bcmd;
  struct execcmd *ecmd;
  struct listcmd *lcmd;
  struct pipecmd *pcmd;
  struct redircmd *rcmd;

  if(cmd == 0)
    exit();

  switch(cmd->type){
  default:
    panic("runcmd");

  case EXEC:
    ecmd = (struct execcmd*)cmd;
    if(ecmd->argv[0] == 0)
      exit();
    exec(ecmd->argv[0], ecmd->argv);
```

```
  printf(2, "exec %s failed\n", ecmd->argv[0]);
  break;

 case REDIR:
  rcmd = (struct redircmd*)cmd;
  close(rcmd->fd);
  if(open(rcmd->file, rcmd->mode) < 0){
   printf(2, "open %s failed\n", rcmd->file);
   exit();
  }
  runcmd(rcmd->cmd);
  break;

 case LIST:
  lcmd = (struct listcmd*)cmd;
  if(fork1() == 0)
   runcmd(lcmd->left);
  wait();
  runcmd(lcmd->right);
  break;

 case PIPE:
  pcmd = (struct pipecmd*)cmd;
  if(pipe(p) < 0)
   panic("pipe");
  if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
   runcmd(pcmd->left);
  }
  if(fork1() == 0){
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   runcmd(pcmd->right);
  }
  close(p[0]);
  close(p[1]);
```

```
    wait();
    wait();
    break;

  case BACK:
    bcmd = (struct backcmd*)cmd;
    if(fork1() == 0)
      runcmd(bcmd->cmd);
    break;
  }
  exit();
}

int
getcmd(char *buf, int nbuf)
{
  printf(2, "$ ");
  memset(buf, 0, nbuf);
  gets(buf, nbuf);
  if(buf[0] == 0) // EOF
    return -1;
  return 0;
}

char* strcat(char* s1,char *s2)
{
    char *b=s1;
    while(*s1) ++s1;
    while(*s2) *s1++ = *s2++;
    *s1=0;
    return b;
}

int
main(void)
{
  static char buf[100],bufx[100];
  int fd;

  // Ensure that three file descriptors are open.
  while((fd = open("console", O_RDWR)) >= 0){
```

```
  if(fd >= 3){
    close(fd);
    break;
  }
}
int err=open("temp.pwd",O_CREATE|O_RDWR);
write(err,"/",1);
close(err);
// Read and run input commands.
while(getcmd(buf, sizeof(buf)) >= 0){
  memset(bufx,'\0',sizeof(bufx));
  if(strlen(buf)>1) bufx[0]='/';
  strcat(bufx,buf);
  //printf(1,"%s\n",bufx);
  if(bufx[1] == 'c' && bufx[2] == 'd' && bufx[3] == ' '){
    // Chdir must be called by the parent, not the child.
    bufx[strlen(bufx)-1] = 0;  // chop \n
    if(bufx[strlen(bufx)-1]=='/') bufx[strlen(bufx)-1]='\0';
    if(chdir(bufx+4) < 0)
    {
      printf(2, "cannot cd %s\n", bufx+4);
    }
    else
    {
      err=open("/temp.pwd",O_RDWR);
      char temp[100];
      int e=read(err,temp,sizeof(temp));
      if(e<0) exit();
      if(strcmp(bufx+4,".")==0) continue;
      if(strcmp(bufx+4,"..")==0)
      {
         temp[strlen(temp)-1]='\0';
         int nn=strlen(temp)-1;
         while(temp[nn]!='/'){
            temp[nn]='\0';
            //printf(1,"%s ",temp);
            nn--;
         }
         unlink("/temp.pwd");
         int err2=open("/temp.pwd",O_CREATE|O_RDWR);
         write(err2,temp,1);
```

```
          close(err2);
          //printf(1,"%s\n",temp);
          continue;
        }
        strcat(bufx,"/");
        write(err,bufx+4,strlen(bufx)-4);
        close(err);
        //printf(1,"~~ %s\n",bufx+4);
      }
      continue;
    }
    if(fork1() == 0)
      runcmd(parsecmd(bufx));
    wait();
  }
  exit();
}

void
panic(char *s)
{
  printf(2, "%s\n", s);
  exit();
}

int
fork1(void)
{
  int pid;

  pid = fork();
  if(pid == -1)
    panic("fork");
  return pid;
}

//PAGEBREAK!
// Constructors

struct cmd*
execcmd(void)
```

```
{
  struct execcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = EXEC;
  return (struct cmd*)cmd;
}

struct cmd*
redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
{
  struct redircmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = REDIR;
  cmd->cmd = subcmd;
  cmd->file = file;
  cmd->efile = efile;
  cmd->mode = mode;
  cmd->fd = fd;
  return (struct cmd*)cmd;
}

struct cmd*
pipecmd(struct cmd *left, struct cmd *right)
{
  struct pipecmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = PIPE;
  cmd->left = left;
  cmd->right = right;
  return (struct cmd*)cmd;
}

struct cmd*
listcmd(struct cmd *left, struct cmd *right)
{
```

```
  struct listcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = LIST;
  cmd->left = left;
  cmd->right = right;
  return (struct cmd*)cmd;
}

struct cmd*
backcmd(struct cmd *subcmd)
{
  struct backcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = BACK;
  cmd->cmd = subcmd;
  return (struct cmd*)cmd;
}
//PAGEBREAK!
// Parsing

char whitespace[] = " \t\r\n\v";
char symbols[] = "<|>&;()";

int
gettoken(char **ps, char *es, char **q, char **eq)
{
  char *s;
  int ret;

  s = *ps;
  while(s < es && strchr(whitespace, *s))
    s++;
  if(q)
    *q = s;
  ret = *s;
  switch(*s){
  case 0:
```

```
    break;
  case '|':
  case '(':
  case ')':
  case ';':
  case '&':
  case '<':
    s++;
    break;
  case '>':
    s++;
    if(*s == '>'){
      ret = '+';
      s++;
    }
    break;
  default:
    ret = 'a';
    while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
      s++;
    break;
  }
  if(eq)
    *eq = s;

  while(s < es && strchr(whitespace, *s))
    s++;
  *ps = s;
  return ret;
}

int
peek(char **ps, char *es, char *toks)
{
  char *s;

  s = *ps;
  while(s < es && strchr(whitespace, *s))
    s++;
  *ps = s;
  return *s && strchr(toks, *s);
```

```
}

struct cmd *parseline(char**, char*);
struct cmd *parsepipe(char**, char*);
struct cmd *parseexec(char**, char*);
struct cmd *nulterminate(struct cmd*);

struct cmd*
parsecmd(char *s)
{
  char *es;
  struct cmd *cmd;

  es = s + strlen(s);
  cmd = parseline(&s, es);
  peek(&s, es, "");
  if(s != es){
    printf(2, "leftovers: %s\n", s);
    panic("syntax");
  }
  nulterminate(cmd);
  return cmd;
}

struct cmd*
parseline(char **ps, char *es)
{
  struct cmd *cmd;

  cmd = parsepipe(ps, es);
  while(peek(ps, es, "&")){
    gettoken(ps, es, 0, 0);
    cmd = backcmd(cmd);
  }
  if(peek(ps, es, ";")){
    gettoken(ps, es, 0, 0);
    cmd = listcmd(cmd, parseline(ps, es));
  }
  return cmd;
}
```

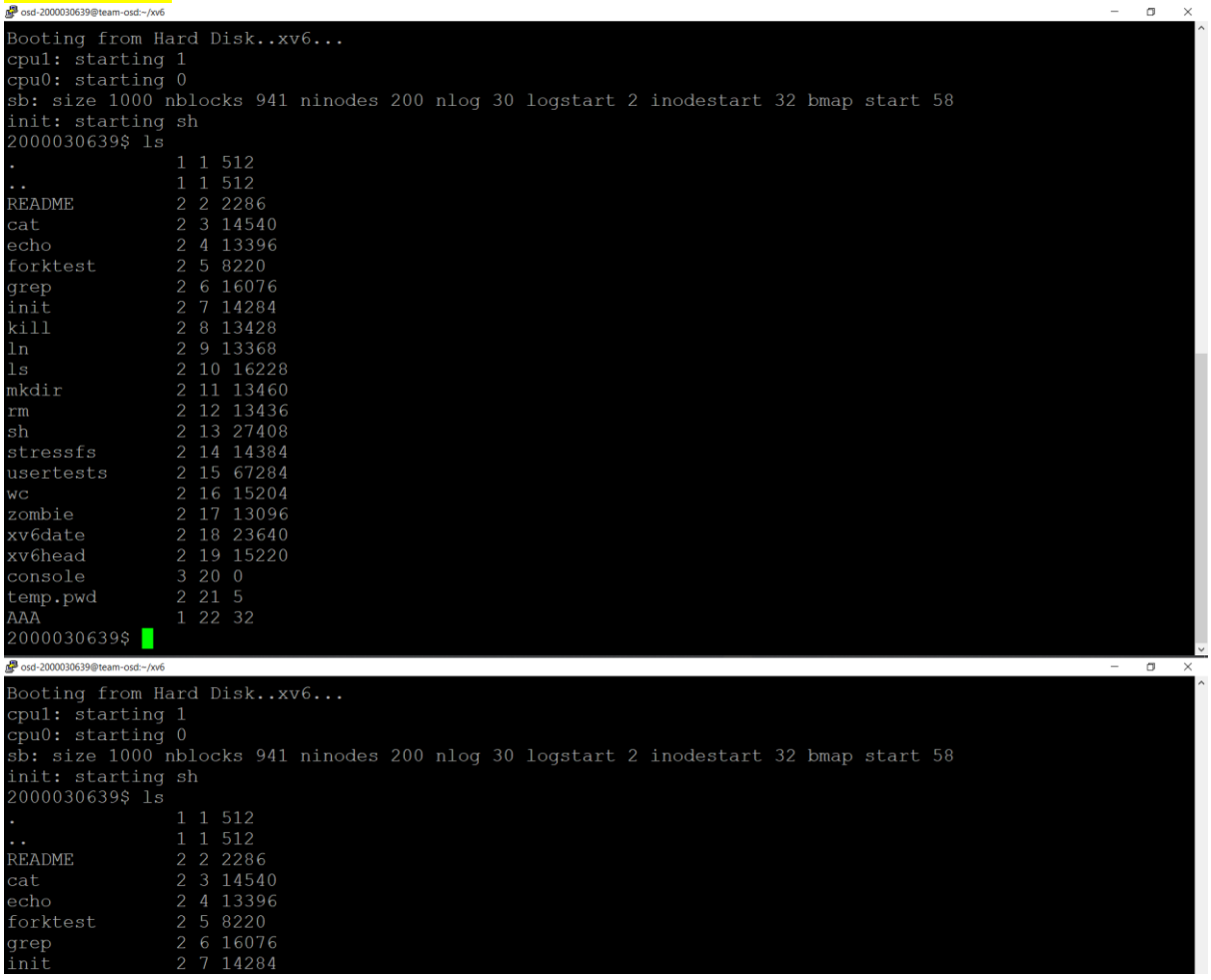
```
struct cmd*
parsepipe(char **ps, char *es)
{
  struct cmd *cmd;

  cmd = parseexec(ps, es);
  if(peek(ps, es, "|")){
    gettoken(ps, es, 0, 0);
    cmd = pipecmd(cmd, parsepipe(ps, es));
  }
  return cmd;
}

struct cmd*
parseredirs(struct cmd *cmd, char **ps, char *es)
{
  int tok;
  char *q, *eq;

  while(peek(ps, es, "<>")){
    tok = gettoken(ps, es, 0, 0);
    if(gettoken(ps, es, &q, &eq) != 'a')
      panic("missing file for redirection");
    switch(tok){
    case '<':
      cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
      break;
    case '>':
      cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
      break;
    case '+':  // >>
      cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
      break;
    }
  }
  return cmd;
}

struct cmd*
parseblock(char **ps, char *es)
{
```

```
  struct cmd *cmd;

  if(!peek(ps, es, "("))
    panic("parseblock");
  gettoken(ps, es, 0, 0);
  cmd = parseline(ps, es);
  if(!peek(ps, es, ")"))
    panic("syntax - missing )");
  gettoken(ps, es, 0, 0);
  cmd = parseredirs(cmd, ps, es);
  return cmd;
}

struct cmd*
parseexec(char **ps, char *es)
{
  char *q, *eq;
  int tok, argc;
  struct execcmd *cmd;
  struct cmd *ret;

  if(peek(ps, es, "("))
    return parseblock(ps, es);

  ret = execcmd();
  cmd = (struct execcmd*)ret;

  argc = 0;
  ret = parseredirs(ret, ps, es);
  while(!peek(ps, es, "|)&;")){
    if((tok=gettoken(ps, es, &q, &eq)) == 0)
      break;
    if(tok != 'a')
      panic("syntax");
    cmd->argv[argc] = q;
    cmd->eargv[argc] = eq;
    argc++;
    if(argc >= MAXARGS)
      panic("too many args");
    ret = parseredirs(ret, ps, es);
  }
```

```
  cmd->argv[argc] = 0;
  cmd->eargv[argc] = 0;
  return ret;
}

// NUL-terminate all the counted strings.
struct cmd*
nulterminate(struct cmd *cmd)
{
  int i;
  struct backcmd *bcmd;
  struct execcmd *ecmd;
  struct listcmd *lcmd;
  struct pipecmd *pcmd;
  struct redircmd *rcmd;

  if(cmd == 0)
    return 0;

  switch(cmd->type){
  case EXEC:
    ecmd = (struct execcmd*)cmd;
    for(i=0; ecmd->argv[i]; i++)
      *ecmd->eargv[i] = 0;
    break;

  case REDIR:
    rcmd = (struct redircmd*)cmd;
    nulterminate(rcmd->cmd);
    *rcmd->efile = 0;
    break;

  case PIPE:
    pcmd = (struct pipecmd*)cmd;
    nulterminate(pcmd->left);
    nulterminate(pcmd->right);
    break;

  case LIST:
    lcmd = (struct listcmd*)cmd;
    nulterminate(lcmd->left);
```

```
    nulterminate(lcmd->right);
    break;

  case BACK:
    bcmd = (struct backcmd*)cmd;
    nulterminate(bcmd->cmd);
    break;
  }
  return cmd;
}
```

Output:-



Task-2:
EDITOR

CODE

```c
#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();
    }
```

```
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     }
```

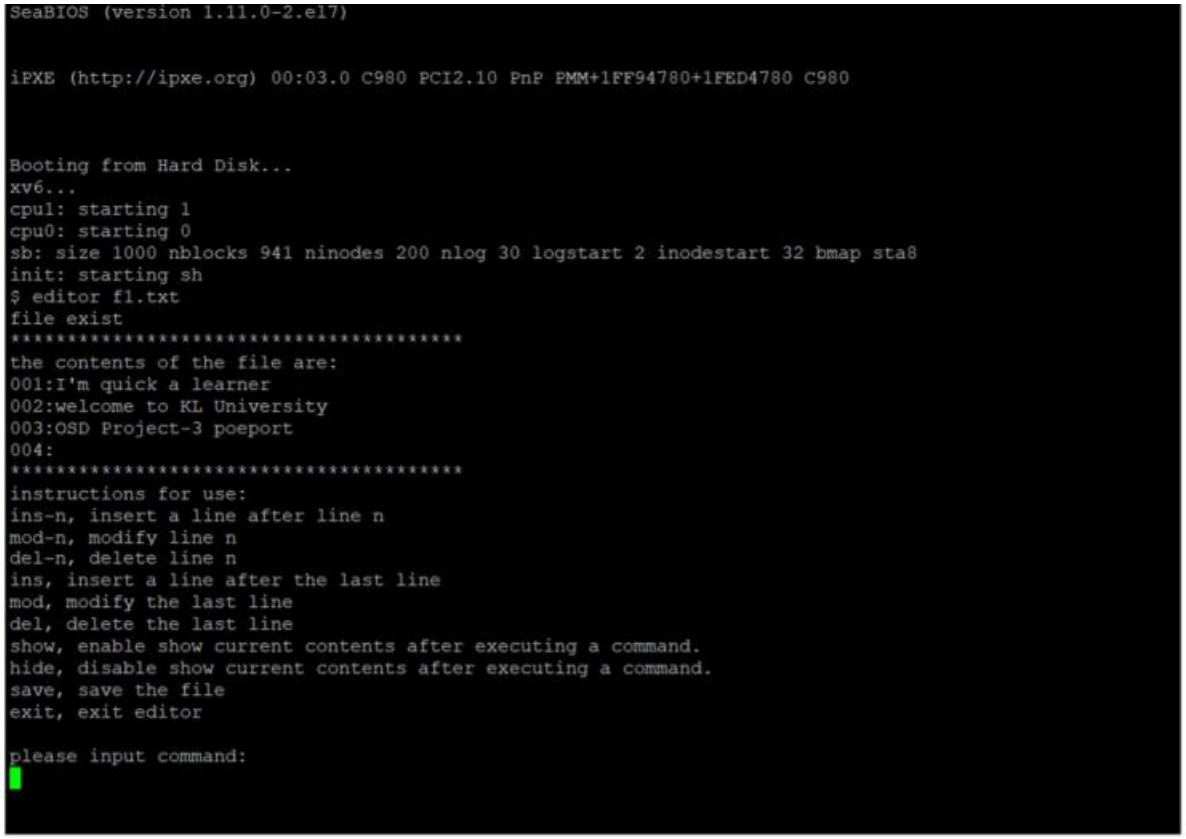
```
287     char input[MAX_LINE_LENGTH] = {};
288     if (*extra == '\0')
289     {
290         printf(1, "please input content:\n");
291         gets(input, MAX_LINE_LENGTH);
292         input[strlen(input)-1] = '\0';
293     }
294     else
295         strcpy(input, extra);
296     memset(text[n-1],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


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 sta8
init: starting sh
$ editor f1.txt
file exist
****
the contents of the file are:
001:I'm quick a learner
002:welcome to KL University
003:OSD Project-3 poeport
004:
****
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:

Task-3:-
WordCount(wc.c)
Code:-

```
#include
<config.h>

#include <stdio.h>
#include <assert.h>
#include <getopt.h>
#include <sys/types.h>
#include <wchar.h>
#include <wctype.h>

#include "system.h"
#include "argv-iter.h"
#include "die.h"
#include "error.h"
#include "fadvise.h"
#include "mbchar.h"
#include "physmem.h"
#include "readtokens0.h"
#include "safe-read.h"
#include "stat-size.h"
#include "xbinary-io.h"

#if !defined iswspace && !HAVE_ISWSPACE
# define iswspace(wc) \
    ((wc) == to_uchar (wc) && isspace (to_uchar (wc)))
#endif

/* The official name of this program (e.g., no 'g' prefix).  */
#define PROGRAM_NAME "wc"

#define AUTHORS \
  proper_name ("Paul Rubin"), \
  proper_name ("David MacKenzie")

/* Size of atomic reads. */
#define BUFFER_SIZE (16 * 1024)

/* Cumulative number of lines, words, chars and bytes in all files so far.
   max_line_length is the maximum over all files processed so far.  */
static uintmax_t total_lines;
static uintmax_t total_words;
static uintmax_t total_chars;
```

```
static uintmax_t total_bytes;
static uintmax_t max_line_length;

/* Which counts to print. */
static bool print_lines, print_words, print_chars, print_bytes;
static bool print_linelength;

/* The print width of each count.  */
static int number_width;

/* True if we have ever read the standard input. */
static bool have_read_stdin;

/* Used to determine if file size can be determined without reading.  */
static size_t page_size;

/* Enable to _not_ treat non breaking space as a word separator.  */
static bool posixly_correct;

/* The result of calling fstat or stat on a file descriptor or file.  */
struct fstatus
{
  /* If positive, fstat or stat has not been called yet.  Otherwise,
     this is the value returned from fstat or stat.  */
  int failed;

  /* If FAILED is zero, this is the file's status.  */
  struct stat st;
};

/* For long options that have no equivalent short option, use a
   non-character as a pseudo short option, starting with CHAR_MAX + 1.  */
enum
{
  FILES0_FROM_OPTION = CHAR_MAX + 1
};

static struct option const longopts[] =
{
  {"bytes", no_argument, NULL, 'c'},
  {"chars", no_argument, NULL, 'm'},
  {"lines", no_argument, NULL, 'l'},
  {"words", no_argument, NULL, 'w'},
  {"files0-from", required_argument, NULL, FILES0_FROM_OPTION},
  {"max-line-length", no_argument, NULL, 'L'},
  {GETOPT_HELP_OPTION_DECL},
  {GETOPT_VERSION_OPTION_DECL},
```

```
  {NULL, 0, NULL, 0}
};

void
usage (int status)
{
  if (status != EXIT_SUCCESS)
    emit_try_help ();
  else
    {
      printf (_("\
Usage: %s [OPTION]... [FILE]...\n\
  or:  %s [OPTION]... --files0-from=F\n\
"),
              program_name, program_name);
      fputs (_("\
Print newline, word, and byte counts for each FILE, and a total line if\n\
more than one FILE is specified.  A word is a non-zero-length sequence of\n\
characters delimited by white space.\n\
"), stdout);

      emit_stdin_note ();

      fputs (_("\
\n\
The options below may be used to select which counts are printed, always in\n\
the following order: newline, word, character, byte, maximum line length.\n\
  -c, --bytes            print the byte counts\n\
  -m, --chars            print the character counts\n\
  -l, --lines            print the newline counts\n\
"), stdout);
      fputs (_("\
      --files0-from=F    read input from the files specified by\n\
                           NUL-terminated names in file F;\n\
                           If F is - then read names from standard input\n\
  -L, --max-line-length  print the maximum display width\n\
  -w, --words            print the word counts\n\
"), stdout);
      fputs (HELP_OPTION_DESCRIPTION, stdout);
      fputs (VERSION_OPTION_DESCRIPTION, stdout);
      emit_ancillary_info (PROGRAM_NAME);
    }
  exit (status);
}

/* Return non zero if a non breaking space.  */
static int _GL_ATTRIBUTE_PURE
```

```
iswnbspace (wint_t wc)
{
  return ! posixly_correct
         && (wc == 0x00A0 || wc == 0x2007
             || wc == 0x202F || wc == 0x2060);
}

static int
isnbspace (int c)
{
  return iswnbspace (btowc (c));
}

/* FILE is the name of the file (or NULL for standard input)
   associated with the specified counters.  */
static void
write_counts (uintmax_t lines,
              uintmax_t words,
              uintmax_t chars,
              uintmax_t bytes,
              uintmax_t linelength,
              const char *file)
{
  static char const format_sp_int[] = " %*s";
  char const *format_int = format_sp_int + 1;
  char buf[INT_BUFSIZE_BOUND (uintmax_t)];

  if (print_lines)
    {
      printf (format_int, number_width, umaxtostr (lines, buf));
      format_int = format_sp_int;
    }
  if (print_words)
    {
      printf (format_int, number_width, umaxtostr (words, buf));
      format_int = format_sp_int;
    }
  if (print_chars)
    {
      printf (format_int, number_width, umaxtostr (chars, buf));
      format_int = format_sp_int;
    }
  if (print_bytes)
    {
      printf (format_int, number_width, umaxtostr (bytes, buf));
      format_int = format_sp_int;
    }
```

```
  if (print_linelength)
    {
      printf (format_int, number_width, umaxtostr (linelength, buf));
    }
  if (file)
    printf (" %s", strchr (file, '\n') ? quotef (file) : file);
  putchar ('\n');
}

/* Count words.  FILE_X is the name of the file (or NULL for standard
   input) that is open on descriptor FD.  *FSTATUS is its status.
   CURRENT_POS is the current file offset if known, negative if unknown.
   Return true if successful.  */
static bool
wc (int fd, char const *file_x, struct fstatus *fstatus, off_t current_pos)
{
  bool ok = true;
  char buf[BUFFER_SIZE + 1];
  size_t bytes_read;
  uintmax_t lines, words, chars, bytes, linelength;
  bool count_bytes, count_chars, count_complicated;
  char const *file = file_x ? file_x : _("standard input");

  lines = words = chars = bytes = linelength = 0;

  /* If in the current locale, chars are equivalent to bytes, we prefer
     counting bytes, because that's easier.  */
#if MB_LEN_MAX > 1
  if (MB_CUR_MAX > 1)
    {
      count_bytes = print_bytes;
      count_chars = print_chars;
    }
  else
#endif
    {
      count_bytes = print_bytes || print_chars;
      count_chars = false;
    }
  count_complicated = print_words || print_linelength;

  /* Advise the kernel of our access pattern only if we will read().  */
  if (!count_bytes || count_chars || print_lines || count_complicated)
    fdadvise (fd, 0, 0, FADVISE_SEQUENTIAL);

  /* When counting only bytes, save some line- and word-counting
     overhead.  If FD is a 'regular' Unix file, using lseek is enough
```

```
   to get its 'size' in bytes.  Otherwise, read blocks of BUFFER_SIZE
   bytes at a time until EOF.  Note that the 'size' (number of bytes)
   that wc reports is smaller than stats.st_size when the file is not
   positioned at its beginning.  That's why the lseek calls below are
   necessary.  For example the command
   '(dd ibs=99k skip=1 count=0; ./wc -c) < /etc/group'
   should make wc report '0' bytes.  */

if (count_bytes && !count_chars && !print_lines && !count_complicated)
  {
    bool skip_read = false;

    if (0 < fstatus->failed)
      fstatus->failed = fstat (fd, &fstatus->st);

    /* For sized files, seek to one st_blksize before EOF rather than to EOF.
       This works better for files in proc-like file systems where
       the size is only approximate.  */
    if (! fstatus->failed && usable_st_size (&fstatus->st)
        && 0 <= fstatus->st.st_size)
      {
        size_t end_pos = fstatus->st.st_size;
        if (current_pos < 0)
          current_pos = lseek (fd, 0, SEEK_CUR);

        if (end_pos % page_size)
          {
            /* We only need special handling of /proc and /sys files etc.
               when they're a multiple of PAGE_SIZE.  In the common case
               for files with st_size not a multiple of PAGE_SIZE,
               it's more efficient and accurate to use st_size.
               Be careful here.  The current position may actually be
               beyond the end of the file.  As in the example above.  */

            bytes = end_pos < current_pos ? 0 : end_pos - current_pos;
            skip_read = true;
          }
        else
          {
            off_t hi_pos = end_pos - end_pos % (ST_BLKSIZE (fstatus->st) + 1);
            if (0 <= current_pos && current_pos < hi_pos
                && 0 <= lseek (fd, hi_pos, SEEK_CUR))
              bytes = hi_pos - current_pos;
          }
      }

    if (! skip_read)
```

```
      {
        fdadvise (fd, 0, 0, FADVISE_SEQUENTIAL);
        while ((bytes_read = safe_read (fd, buf, BUFFER_SIZE)) > 0)
          {
            if (bytes_read == SAFE_READ_ERROR)
              {
                error (0, errno, "%s", quotef (file));
                ok = false;
                break;
              }
            bytes += bytes_read;
          }
      }
  }
else if (!count_chars && !count_complicated)
  {
    /* Use a separate loop when counting only lines or lines and bytes --
       but not chars or words.  */
    bool long_lines = false;
    while ((bytes_read = safe_read (fd, buf, BUFFER_SIZE)) > 0)
      {
        if (bytes_read == SAFE_READ_ERROR)
          {
            error (0, errno, "%s", quotef (file));
            ok = false;
            break;
          }

        bytes += bytes_read;

        char *p = buf;
        char *end = p + bytes_read;
        uintmax_t plines = lines;

        if (! long_lines)
          {
            /* Avoid function call overhead for shorter lines.  */
            while (p != end)
              lines += *p++ == '\n';
          }
        else
          {
            /* memchr is more efficient with longer lines.  */
            while ((p = memchr (p, '\n', end - p)))
              {
                ++p;
                ++lines;
```

```
                }
            }

          /* If the average line length in the block is >= 15, then use
             memchr for the next block, where system specific optimizations
             may outweigh function call overhead.
             FIXME: This line length was determined in 2015, on both
             x86_64 and ppc64, but it's worth re-evaluating in future with
             newer compilers, CPUs, or memchr() implementations etc.  */
          if (lines - plines <= bytes_read / 15)
            long_lines = true;
          else
            long_lines = false;
        }
    }
#if MB_LEN_MAX > 1
# define SUPPORT_OLD_MBRTOWC 1
  else if (MB_CUR_MAX > 1)
    {
      bool in_word = false;
      uintmax_t linepos = 0;
      mbstate_t state = { 0, };
      bool in_shift = false;
# if SUPPORT_OLD_MBRTOWC
      /* Back-up the state before each multibyte character conversion and
         move the last incomplete character of the buffer to the front
         of the buffer.  This is needed because we don't know whether
         the 'mbrtowc' function updates the state when it returns -2, --
         this is the ISO C 99 and glibc-2.2 behaviour - or not - amended
         ANSI C, glibc-2.1 and Solaris 5.7 behaviour.  We don't have an
         autoconf test for this, yet.  */
      size_t prev = 0; /* number of bytes carried over from previous round */
# else
      const size_t prev = 0;
# endif

      while ((bytes_read = safe_read (fd, buf + prev, BUFFER_SIZE - prev)) > 0)
        {
          const char *p;
# if SUPPORT_OLD_MBRTOWC
          mbstate_t backup_state;
# endif
          if (bytes_read == SAFE_READ_ERROR)
            {
              error (0, errno, "%s", quotef (file));
              ok = false;
              break;
```

```
            }

          bytes += bytes_read;
          p = buf;
          bytes_read += prev;
          do
            {
              wchar_t wide_char;
              size_t n;
              bool wide = true;

              if (!in_shift && is_basic (*p))
                {
                  /* Handle most ASCII characters quickly, without calling
                     mbrtowc().  */
                  n = 1;
                  wide_char = *p;
                  wide = false;
                }
              else
                {
                  in_shift = true;
# if SUPPORT_OLD_MBRTOWC
                  backup_state = state;
# endif
                  n = mbrtowc (&wide_char, p, bytes_read, &state);
                  if (n == (size_t) -2)
                    {
# if SUPPORT_OLD_MBRTOWC
                      state = backup_state;
# endif
                      break;
                    }
                  if (n == (size_t) -1)
                    {
                      /* Remember that we read a byte, but don't complain
                         about the error.  Because of the decoding error,
                         this is a considered to be byte but not a
                         character (that is, chars is not incremented).  */
                      p++;
                      bytes_read--;
                      continue;
                    }
                  if (mbsinit (&state))
                    in_shift = false;
                  if (n == 0)
                    {
```

```
        wide_char = 0;
        n = 1;
      }
  }

switch (wide_char)
  {
  case '\n':
    lines++;
    FALLTHROUGH;
  case '\r':
  case '\f':
    if (linepos > linelength)
      linelength = linepos;
    linepos = 0;
    goto mb_word_separator;
  case '\t':
    linepos += 8 - (linepos % 8);
    goto mb_word_separator;
  case ' ':
    linepos++;
    FALLTHROUGH;
  case '\v':
  mb_word_separator:
    words += in_word;
    in_word = false;
    break;
  default:
    if (wide && iswprint (wide_char))
      {
        /* wcwidth can be expensive on OSX for example,
           so avoid if uneeded.  */
        if (print_linelength)
          {
            int width = wcwidth (wide_char);
            if (width > 0)
              linepos += width;
          }
        if (iswspace (wide_char) || iswnbspace (wide_char))
          goto mb_word_separator;
        in_word = true;
      }
    else if (!wide && isprint (to_uchar (*p)))
      {
        linepos++;
        if (isspace (to_uchar (*p)))
          goto mb_word_separator;
```

```
                    in_word = true;
                  }
                break;
              }

            p += n;
            bytes_read -= n;
            chars++;
          }
        while (bytes_read > 0);

# if SUPPORT_OLD_MBRTOWC
        if (bytes_read > 0)
          {
            if (bytes_read == BUFFER_SIZE)
              {
                /* Encountered a very long redundant shift sequence.  */
                p++;
                bytes_read--;
              }
            memmove (buf, p, bytes_read);
          }
        prev = bytes_read;
# endif
      }
    if (linepos > linelength)
      linelength = linepos;
    words += in_word;
  }
#endif
else
  {
    bool in_word = false;
    uintmax_t linepos = 0;

    while ((bytes_read = safe_read (fd, buf, BUFFER_SIZE)) > 0)
      {
        const char *p = buf;
        if (bytes_read == SAFE_READ_ERROR)
          {
            error (0, errno, "%s", quotef (file));
            ok = false;
            break;
          }

        bytes += bytes_read;
        do
```

```
          {
            switch (*p++)
              {
              case '\n':
                lines++;
                FALLTHROUGH;
              case '\r':
              case '\f':
                if (linepos > linelength)
                  linelength = linepos;
                linepos = 0;
                goto word_separator;
              case '\t':
                linepos += 8 - (linepos % 8);
                goto word_separator;
              case ' ':
                linepos++;
                FALLTHROUGH;
              case '\v':
              word_separator:
                words += in_word;
                in_word = false;
                break;
              default:
                if (isprint (to_uchar (p[-1])))
                  {
                    linepos++;
                    if (isspace (to_uchar (p[-1]))
                        || isnbspace (to_uchar (p[-1])))
                      goto word_separator;
                    in_word = true;
                  }
                break;
              }
          }
        while (--bytes_read);
      }
    if (linepos > linelength)
      linelength = linepos;
    words += in_word;
  }

if (count_chars < print_chars)
  chars = bytes;

write_counts (lines, words, chars, bytes, linelength, file_x);
total_lines += lines;
```

```
  total_words += words;
  total_chars += chars;
  total_bytes += bytes;
  if (linelength > max_line_length)
    max_line_length = linelength;

  return ok;
}

static bool
wc_file (char const *file, struct fstatus *fstatus)
{
  if (! file || STREQ (file, "-"))
    {
      have_read_stdin = true;
      xset_binary_mode (STDIN_FILENO, O_BINARY);
      return wc (STDIN_FILENO, file, fstatus, -1);
    }
  else
    {
      int fd = open (file, O_RDONLY | O_BINARY);
      if (fd == -1)
        {
          error (0, errno, "%s", quotef (file));
          return false;
        }
      else
        {
          bool ok = wc (fd, file, fstatus, 0);
          if (close (fd) != 0)
            {
              error (0, errno, "%s", quotef (file));
              return false;
            }
          return ok;
        }
    }
}

/* Return the file status for the NFILES files addressed by FILE.
   Optimize the case where only one number is printed, for just one
   file; in that case we can use a print width of 1, so we don't need
   to stat the file.  Handle the case of (nfiles == 0) in the same way;
   that happens when we don't know how long the list of file names will be.  */

static struct fstatus *
get_input_fstatus (size_t nfiles, char *const *file)
```

```
{
  struct fstatus *fstatus = xnmalloc (nfiles ? nfiles : 1, sizeof *fstatus);

  if (nfiles == 0
      || (nfiles == 1
          && ((print_lines + print_words + print_chars
               + print_bytes + print_linelength)
              == 1)))
    fstatus[0].failed = 1;
  else
    {
      for (size_t i = 0; i < nfiles; i++)
        fstatus[i].failed = (! file[i] || STREQ (file[i], "-")
                             ? fstat (STDIN_FILENO, &fstatus[i].st)
                             : stat (file[i], &fstatus[i].st));
    }

  return fstatus;
}

/* Return a print width suitable for the NFILES files whose status is
   recorded in FSTATUS.  Optimize the same special case that
   get_input_fstatus optimizes.  */

static int _GL_ATTRIBUTE_PURE
compute_number_width (size_t nfiles, struct fstatus const *fstatus)
{
  int width = 1;

  if (0 < nfiles && fstatus[0].failed <= 0)
    {
      int minimum_width = 1;
      uintmax_t regular_total = 0;

      for (size_t i = 0; i < nfiles; i++)
        if (! fstatus[i].failed)
          {
            if (S_ISREG (fstatus[i].st.st_mode))
              regular_total += fstatus[i].st.st_size;
            else
              minimum_width = 7;
          }

      for (; 10 <= regular_total; regular_total /= 10)
        width++;
      if (width < minimum_width)
        width = minimum_width;
```

```
    }

  return width;
}


int
main (int argc, char **argv)
{
  bool ok;
  int optc;
  size_t nfiles;
  char **files;
  char *files_from = NULL;
  struct fstatus *fstatus;
  struct Tokens tok;

  initialize_main (&argc, &argv);
  set_program_name (argv[0]);
  setlocale (LC_ALL, "");
  bindtextdomain (PACKAGE, LOCALEDIR);
  textdomain (PACKAGE);

  atexit (close_stdout);

  page_size = getpagesize ();
  /* Line buffer stdout to ensure lines are written atomically and immediately
     so that processes running in parallel do not intersperse their output.  */
  setvbuf (stdout, NULL, _IOLBF, 0);

  posixly_correct = (getenv ("POSIXLY_CORRECT") != NULL);

  print_lines = print_words = print_chars = print_bytes = false;
  print_linelength = false;
  total_lines = total_words = total_chars = total_bytes = max_line_length = 0;

  while ((optc = getopt_long (argc, argv, "clLmw", longopts, NULL)) != -1)
    switch (optc)
      {
      case 'c':
        print_bytes = true;
        break;

      case 'm':
        print_chars = true;
        break;
```

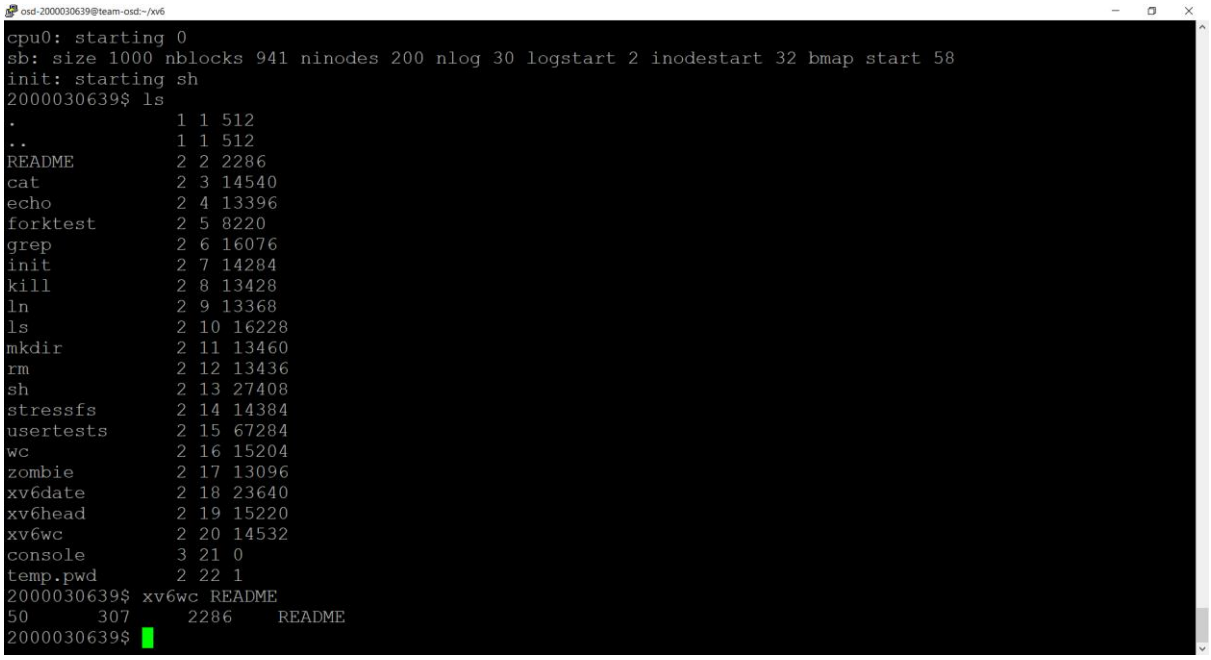
```
    case 'l':
      print_lines = true;
      break;

    case 'w':
      print_words = true;
      break;

    case 'L':
      print_linelength = true;
      break;

    case FILES0_FROM_OPTION:
      files_from = optarg;
      break;

    case_GETOPT_HELP_CHAR;

    case_GETOPT_VERSION_CHAR (PROGRAM_NAME, AUTHORS);

    default:
      usage (EXIT_FAILURE);
    }

if (! (print_lines || print_words || print_chars || print_bytes
       || print_linelength))
  print_lines = print_words = print_bytes = true;

bool read_tokens = false;
struct argv_iterator *ai;
if (files_from)
  {
    FILE *stream;

    /* When using --files0-from=F, you may not specify any files
       on the command-line.  */
    if (optind < argc)
      {
        error (0, 0, _("extra operand %s"), quoteaf (argv[optind]));
        fprintf (stderr, "%s\n",
                 _("file operands cannot be combined with --files0-from"));
        usage (EXIT_FAILURE);
      }

    if (STREQ (files_from, "-"))
      stream = stdin;
    else
```

```
      {
        stream = fopen (files_from, "r");
        if (stream == NULL)
          die (EXIT_FAILURE, errno, _("cannot open %s for reading"),
               quoteaf (files_from));
      }

    /* Read the file list into RAM if we can detect its size and that
       size is reasonable.  Otherwise, we'll read a name at a time.  */
    struct stat st;
    if (fstat (fileno (stream), &st) == 0
        && S_ISREG (st.st_mode)
        && st.st_size <= MIN (10 * 1024 * 1024, physmem_available () / 2))
      {
        read_tokens = true;
        readtokens0_init (&tok);
        if (! readtokens0 (stream, &tok) || fclose (stream) != 0)
          die (EXIT_FAILURE, 0, _("cannot read file names from %s"),
               quoteaf (files_from));
        files = tok.tok;
        nfiles = tok.n_tok;
        ai = argv_iter_init_argv (files);
      }
    else
      {
        files = NULL;
        nfiles = 0;
        ai = argv_iter_init_stream (stream);
      }
  }
else
  {
    static char *stdin_only[] = { NULL };
    files = (optind < argc ? argv + optind : stdin_only);
    nfiles = (optind < argc ? argc - optind : 1);
    ai = argv_iter_init_argv (files);
  }

if (!ai)
  xalloc_die ();

fstatus = get_input_fstatus (nfiles, files);
number_width = compute_number_width (nfiles, fstatus);

ok = true;
for (int i = 0; /* */; i++)
  {
```

```
bool skip_file = false;
enum argv_iter_err ai_err;
char *file_name = argv_iter (ai, &ai_err);
if (!file_name)
  {
    switch (ai_err)
      {
      case AI_ERR_EOF:
        goto argv_iter_done;
      case AI_ERR_READ:
        error (0, errno, _("%s: read error"),
               quotef (files_from));
        ok = false;
        goto argv_iter_done;
      case AI_ERR_MEM:
        xalloc_die ();
      default:
        assert (!"unexpected error code from argv_iter");
      }
  }
if (files_from && STREQ (files_from, "-") && STREQ (file_name, "-"))
  {
    /* Give a better diagnostic in an unusual case:
       printf - | wc --files0-from=- */
    error (0, 0, _("when reading file names from stdin, "
                   "no file name of %s allowed"),
           quoteaf (file_name));
    skip_file = true;
  }

if (!file_name[0])
  {
    /* Diagnose a zero-length file name.  When it's one
       among many, knowing the record number may help.
       FIXME: currently print the record number only with
       --files0-from=FILE.  Maybe do it for argv, too?  */
    if (files_from == NULL)
      error (0, 0, "%s", _("invalid zero-length file name"));
    else
      {
        /* Using the standard 'filename:line-number:' prefix here is
           not totally appropriate, since NUL is the separator, not NL,
           but it might be better than nothing.  */
        unsigned long int file_number = argv_iter_n_args (ai);
        error (0, 0, "%s:%lu: %s", quotef (files_from),
               file_number, _("invalid zero-length file name"));
      }
```

```
            skip_file = true;
          }

        if (skip_file)
          ok = false;
        else
          ok &= wc_file (file_name, &fstatus[nfiles ? i : 0]);

        if (! nfiles)
          fstatus[0].failed = 1;
      }
  argv_iter_done:

    /* No arguments on the command line is fine.  That means read from stdin.
       However, no arguments on the --files0-from input stream is an error
       means don't read anything.  */
    if (ok && !files_from && argv_iter_n_args (ai) == 0)
      ok &= wc_file (NULL, &fstatus[0]);

    if (read_tokens)
      readtokens0_free (&tok);

    if (1 < argv_iter_n_args (ai))
      write_counts (total_lines, total_words, total_chars, total_bytes,
                    max_line_length, _("total"));

    argv_iter_free (ai);

    free (fstatus);

    if (have_read_stdin && close (STDIN_FILENO) != 0)
      die (EXIT_FAILURE, errno, "-");

    return ok ? EXIT_SUCCESS : EXIT_FAILURE;
  }
```

Output:-



Task-4:

Double Indirect Block FileSystem(fs.c)

Code:

```
#define
_POSIX_C_SOURCE
200809L
                    #include <stdlib.h>
                    #include <stdio.h>
                    #include <dirent.h>
                    #include <string.h>
                    #include <errno.h>
                    #include <sys/types.h>
                    #include <sys/stat.h>
                    #include <unistd.h>
                    #include <fcntl.h>
                    #include "fs.h"

                    void
                    fs_error (const char *prefix) {
                      char fmt[256];
                      sprintf(fmt, "fs: %s: error", prefix);
                      perror(fmt);
                    }


                    FILE *
                    fs_open (const char *path, const char *flags) {
                      return fopen(path, flags);
```

```
}


int
fs_close (FILE *file) {
  return fclose(file);
}


int
fs_rename (const char *from, const char *to) {
  return rename(from, to);
}


fs_stats *
fs_stat (const char *path) {
  fs_stats *stats = (fs_stats*) malloc(sizeof(fs_stats));
  int e = stat(path, stats);
  if (-1 == e) {
    free(stats);
    return NULL;
  }
  return stats;
}


fs_stats *
fs_fstat (FILE *file) {
  if (NULL == file) return NULL;
  fs_stats *stats = (fs_stats*) malloc(sizeof(fs_stats));
  int fd = fileno(file);
  int e = fstat(fd, stats);
  if (-1 == e) {
    free(stats);
    return NULL;
  }
  return stats;
}


fs_stats *
fs_lstat (const char *path) {
  fs_stats *stats = (fs_stats*) malloc(sizeof(fs_stats));
#ifdef _WIN32
  int e = stat(path, stats);
#else
```

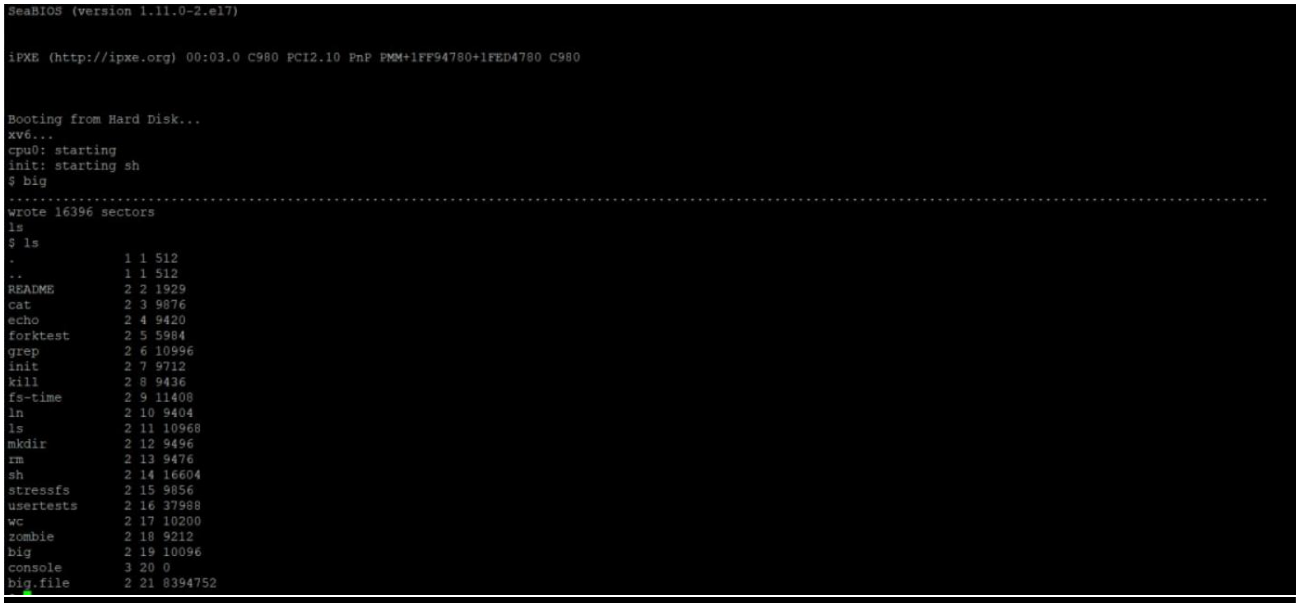
```
  int e = lstat(path, stats);
#endif
  if (-1 == e) {
    free(stats);
    return NULL;
  }
  return stats;
}


int
fs_ftruncate (FILE *file, int len) {
  int fd = fileno(file);
  return ftruncate(fd, (off_t) len);
}


int
fs_truncate (const char *path, int len) {
#ifdef _WIN32
  int ret = -1;
  int fd = open(path, O_RDWR | O_CREAT, S_IREAD | S_IWRITE);
  if (fd != -1) {
    ret = ftruncate(fd, (off_t) len);
    close(fd);
  }
  return ret;
#else
  return truncate(path, (off_t) len);
#endif
}


int
fs_chown (const char *path, int uid, int gid) {
#ifdef _WIN32
  errno = ENOSYS;
  return -1;
#else
  return chown(path, (uid_t) uid, (gid_t) gid);
#endif
}


int
fs_fchown (FILE *file, int uid, int gid) {
#ifdef _WIN32
```

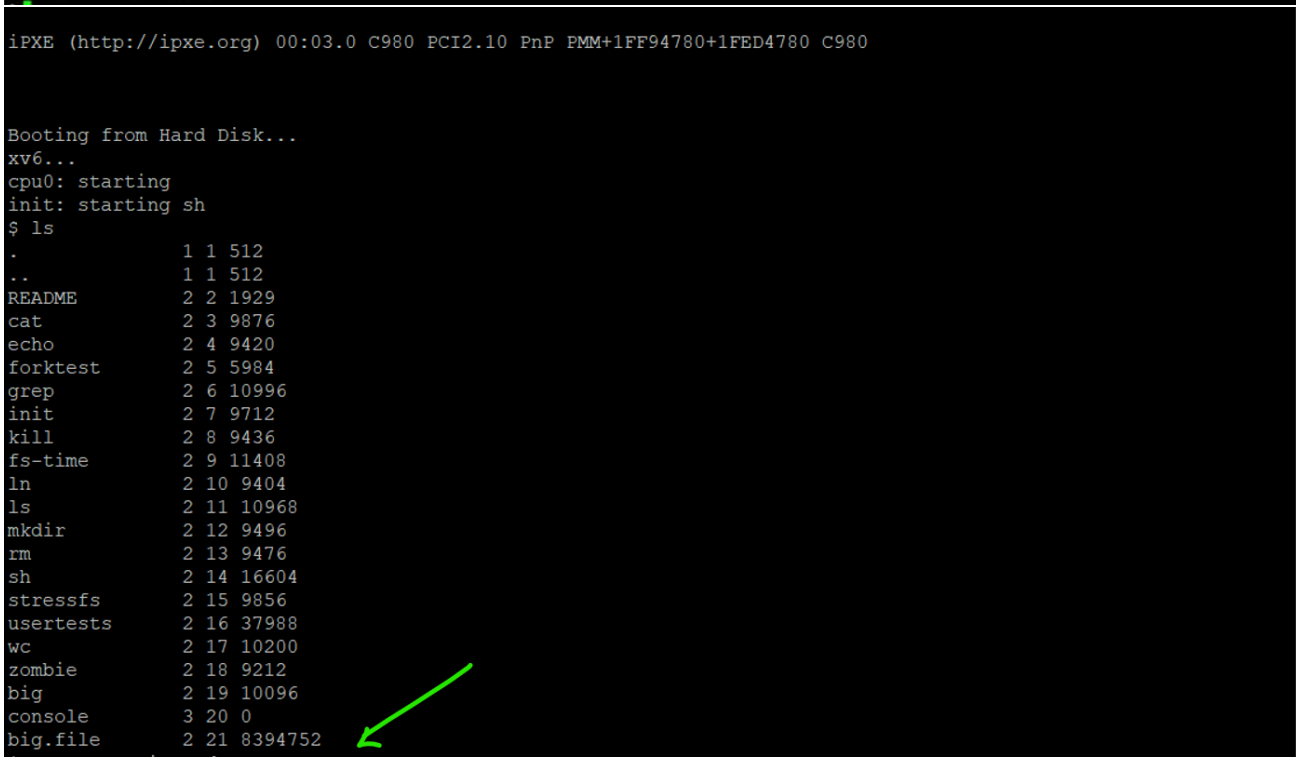
```
  errno = ENOSYS;
  return -1;
#else
  int fd = fileno(file);
  return fchown(fd, (uid_t) uid, (gid_t) gid);
#endif
}


int
fs_lchown (const char *path, int uid, int gid) {
#ifdef _WIN32
  errno = ENOSYS;
  return -1;
#else
  return lchown(path, (uid_t) uid, (gid_t) gid);
#endif
}


size_t
fs_size (const char *path) {
  size_t size;
  FILE *file = fs_open(path, FS_OPEN_READ);
  if (NULL == file) return -1;
  fseek(file, 0, SEEK_END);
  size = ftell(file);
  fs_close(file);
  return size;
}


size_t
fs_fsize (FILE *file) {
  // store current position
  unsigned long pos = ftell(file);
  rewind(file);
  fseek(file, 0, SEEK_END);
  size_t size = ftell(file);
  fseek(file, pos, SEEK_SET);
  return size;
}


char *
fs_read (const char *path) {
  FILE *file = fs_open(path, FS_OPEN_READ);
```

```
  if (NULL == file) return NULL;
  char *data = fs_fread(file);
  fclose(file);
  return data;
}


char *
fs_nread (const char *path, int len) {
  FILE *file = fs_open(path, FS_OPEN_READ);
  if (NULL == file) return NULL;
  char *buffer = fs_fnread(file, len);
  fs_close(file);
  return buffer;
}


char *
fs_fread (FILE *file) {
  size_t fsize = fs_fsize(file);
  return fs_fnread(file, fsize);
}


char *
fs_fnread (FILE *file, int len) {
  char *buffer = (char*) malloc(sizeof(char) * (len + 1));
  size_t n = fread(buffer, 1, len, file);
  buffer[n] = '\0';
  return buffer;
}


int
fs_write (const char *path, const char *buffer) {
  return fs_nwrite(path, buffer, strlen(buffer));
}


int
fs_nwrite (const char *path, const char *buffer, int len) {
  FILE *file = fs_open(path, FS_OPEN_WRITE);
  if (NULL == file) return -1;
  int result = fs_fnwrite(file, buffer, len);
  fclose(file);
  return result;
}
```

```
int
fs_fwrite (FILE *file, const char *buffer) {
  return fs_fnwrite(file, buffer, strlen(buffer));
}


int
fs_fnwrite (FILE *file, const char *buffer, int len) {
  return (int) fwrite(buffer, 1, len, file);
}


int
fs_mkdir (const char *path, int mode) {
#ifdef _WIN32
  return mkdir(path);
#else
  return mkdir(path, (mode_t) mode);
#endif
}


int
fs_rmdir (const char *path) {
  return rmdir(path);
}


int
fs_exists (const char *path) {
  struct stat b;
  return stat(path, &b);
}
```

Output:





Task-5:

Completely Fair Scheduler(CFS)(proc.c)

Code:

```
#include
"types.h"
            #include "defs.h"
            #include "param.h"
            #include "memlayout.h"
            #include "mmu.h"
            #include "x86.h"
```

```c
#include "proc.h"
#include "spinlock.h"

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;

static struct proc *initproc;

int nextpid = 1;
extern void forkret(void);
extern void trapret(void);

static void wakeup1(void *chan);

void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
}

//PAGEBREAK: 32
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
```

```
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;

  return p;
}

//PAGEBREAK: 32
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");
```

```
  p->state = RUNNABLE;
}

// Grow current process's memory by n bytes.
// Return 0 on success, -1 on failure.
int
growproc(int n)
{
  uint sz;

  sz = proc->sz;
  if(n > 0){
    if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  proc->sz = sz;
  switchuvm(proc);
  return 0;
}

// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
fork(void)
{
  int i, pid;
  struct proc *np;

  // Allocate process.
  if((np = allocproc()) == 0)
    return -1;

  // Copy process state from p.
  if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = proc->sz;
  np->parent = proc;
  *np->tf = *proc->tf;
```

```
  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(proc->ofile[i])
      np->ofile[i] = filedup(proc->ofile[i]);
  np->cwd = idup(proc->cwd);

  pid = np->pid;
  np->state = RUNNABLE;
  safestrcpy(np->name, proc->name, sizeof(proc->name));
  return pid;
}

// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
void
exit(void)
{
  struct proc *p;
  int fd;

  if(proc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(proc->ofile[fd]){
      fileclose(proc->ofile[fd]);
      proc->ofile[fd] = 0;
    }
  }

  iput(proc->cwd);
  proc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(proc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == proc){
      p->parent = initproc;
```

```
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  proc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}

// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
wait(void)
{
  struct proc *p;
  int havekids, pid;

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for zombie children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != proc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->state = UNUSED;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || proc->killed){
      release(&ptable.lock);
      return -1;
```

```
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(proc, &ptable.lock);  //DOC: wait-sleep
  }
}

//PAGEBREAK: 42
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```

```
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state.
void
sched(void)
{
  int intena;

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(proc->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler);
  cpu->intena = intena;
}

// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  proc->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}

// A fork child's very first scheduling by scheduler()
// will swtch here.  "Return" to user space.
void
forkret(void)
{
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);

  if (first) {
    // Some initialization functions must be run in the context
    // of a regular process (e.g., they call sleep), and thus cannot
    // be run from main().
    first = 0;
    initlog();
  }
```

```
  // Return to "caller", actually trapret (see allocproc).
}

// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
  if(proc == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }

  // Go to sleep.
  proc->chan = chan;
  proc->state = SLEEPING;
  sched();

  // Tidy up.
  proc->chan = 0;

  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}

//PAGEBREAK!
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
```

```
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}

// Kill the process with the given pid.
// Process won't exit until it returns
// to user space (see trap in trap.c).
int
kill(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}

//PAGEBREAK: 36
// Print a process listing to console.  For debugging.
// Runs when user types ^P on console.
// No lock to avoid wedging a stuck machine further.
void
procdump(void)
{
  static char *states[] = {
```

```
  [UNUSED]    "unused",
  [EMBRYO]    "embryo",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };
  int i;
  struct proc *p;
  char *state;
  uint pc[10];

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    else
      state = "???";
    cprintf("%d %s %s", p->pid, state, p->name);
    if(p->state == SLEEPING){
      getcallerpcs((uint*)p->context->ebp+2, pc);
      for(i=0; i<10 && pc[i] != 0; i++)
        cprintf(" %p", pc[i]);
    }
    cprintf("\n");
  }
}
```

Explaination For CFS:-

OVERVIEW

CFS stands for "Completely Fair Scheduler," and is the new "desktop" process scheduler implemented by Ingo Molnar and merged in Linux 2.6.23. It is the replacement for the previous vanilla scheduler's SCHED_OTHER interactivity code.

80% of CFS's design can be summed up in a single sentence: CFS basically models an "ideal, precise multi-tasking CPU" on real hardware.

"Ideal multi-tasking CPU" is a (non-existent :-)) CPU that has 100% physical power and which can run each task at precise equal speed, in parallel, each at 1/nr_running speed. For example: if there are 2 tasks running, then it runs each at 50% physical power --- i.e., actually in parallel.

On real hardware, we can run only a single task at once, so we have to introduce the concept of "virtual runtime." The virtual runtime of a task specifies when its next timeslice would start execution on the ideal multi-tasking CPU described above. In practice, the virtual runtime of a task is its actual runtime normalized to the total number of running tasks.

FEW IMPLEMENTATION DETAILS

In CFS the virtual runtime is expressed and tracked via the per-task p->se.vruntime (nanosec-unit) value. This way, it's possible to accurately timestamp and measure the "expected CPU time" a task should have gotten.

[small detail: on "ideal" hardware, at any time all tasks would have the same p->se.vruntime value --- i.e., tasks would execute simultaneously and no task would ever get "out of balance" from the "ideal" share of CPU time.]

CFS's task picking logic is based on this p->se.vruntime value and it is thus very simple: it always tries to run the task with the smallest p->se.vruntime value (i.e., the task which executed least so far). CFS always tries to split up CPU time between runnable tasks as close to "ideal multitasking hardware" as possible.

Most of the rest of CFS's design just falls out of this really simple concept, with a few add-on embellishments like nice levels, multiprocessing and various algorithm variants to recognize sleepers.

THE RBTREE

CFS's design is quite radical: it does not use the old data structures for the runqueues, but it uses a time-ordered rbtree to build a "timeline" of future task execution, and thus has no "array switch" artifacts (by which both the previous vanilla scheduler and RSDL/SD are affected).

CFS also maintains the rq->cfs.min_vruntime value, which is a monotonic increasing value tracking the smallest vruntime among all tasks in the runqueue. The total amount of work done by the system is tracked using min_vruntime; that value is used to place newly activated entities on the left side of the tree as much as possible.

The total number of running tasks in the runqueue is accounted through the rq->cfs.load value, which is the sum of the weights of the tasks queued on the runqueue.

CFS maintains a time-ordered rbtree, where all runnable tasks are sorted by the p->se.vruntime key. CFS picks the "leftmost" task from this tree and sticks to it.
As the system progresses forwards, the executed tasks are put into the tree more and more to the right --- slowly but surely giving a chance for every task to become the "leftmost task" and thus get on the CPU within a deterministic amount of time.

Summing up, CFS works like this: it runs a task a bit, and when the task schedules (or a scheduler tick happens) the task's CPU usage is "accounted for": the (small) time it just spent using the physical CPU is added to p->se.vruntime. Once p->se.vruntime gets high enough so that another task becomes the "leftmost task" of the time-ordered rbtree it maintains (plus a small amount of "granularity" distance relative to the leftmost task so that we do not over-schedule tasks and trash the cache), then the new leftmost task is picked and the current task is preempted.

SOME FEATURES OF CFS

CFS uses nanosecond granularity accounting and does not rely on any jiffies or other HZ detail. Thus the CFS scheduler has no notion of "timeslices" in the way the previous scheduler had, and has no heuristics whatsoever. There is only one central tunable (you have to switch on CONFIG_SCHED_DEBUG):

   /proc/sys/kernel/sched_min_granularity_ns

which can be used to tune the scheduler from "desktop" (i.e., low latencies) to "server" (i.e., good batching) workloads. It defaults to a setting suitable for desktop workloads. SCHED_BATCH is handled by the CFS scheduler module too.

Due to its design, the CFS scheduler is not prone to any of the "attacks" that exist today against the heuristics of the stock scheduler: fiftyp.c, thud.c, chew.c, ring-test.c, massive_intr.c all work fine and do not impact interactivity and produce the expected behavior.

The CFS scheduler has a much stronger handling of nice levels and SCHED_BATCH than the previous vanilla scheduler: both types of workloads are isolated much more aggressively.

SMP load-balancing has been reworked/sanitized: the runqueue-walking assumptions are gone from the load-balancing code now, and iterators of the scheduling modules are used. The balancing code got quite a bit simpler as a result.

Scheduling policies

CFS implements three scheduling policies:

- SCHED_NORMAL (traditionally called SCHED_OTHER): The scheduling policy that is used for regular tasks.

- SCHED_BATCH: Does not preempt nearly as often as regular tasks would, thereby allowing tasks to run longer and make better use of caches but at the cost of interactivity. This is well suited for batch jobs.

- SCHED_IDLE: This is even weaker than nice 19, but its not a true idle timer scheduler in order to avoid to get into priority inversion problems which would deadlock the machine.

SCHED_FIFO/_RR are implemented in sched/rt.c and are as specified by POSIX.

The command chrt from util-linux-ng 2.13.1.1 can set all of these except SCHED_IDLE.

SCHEDULING CLASSES

The new CFS scheduler has been designed in such a way to introduce "Scheduling Classes," an extensible hierarchy of scheduler modules. These modules encapsulate scheduling policy details and are handled by the scheduler core without the core code assuming too much about them.

sched/fair.c implements the CFS scheduler described above.

sched/rt.c implements SCHED_FIFO and SCHED_RR semantics, in a simpler way than the previous vanilla scheduler did. It uses 100 runqueues (for all 100 RT priority levels, instead of 140 in the previous scheduler) and it needs no expired array.

Scheduling classes are implemented through the sched_class structure, which contains hooks to functions that must be called whenever an interesting event occurs.

This is the (partial) list of the hooks:

- enqueue_task(...)

  Called when a task enters a runnable state.
  It puts the scheduling entity (task) into the red-black tree and
  increments the nr_running variable.

- dequeue_task(...)

  When a task is no longer runnable, this function is called to keep the
  corresponding scheduling entity out of the red-black tree. It decrements
  the nr_running variable.

- yield_task(...)

  This function is basically just a dequeue followed by an enqueue, unless the
  compat_yield sysctl is turned on; in that case, it places the scheduling
  entity at the right-most end of the red-black tree.

- check_preempt_curr(...)

  This function checks if a task that entered the runnable state should
  preempt the currently running task.

- pick_next_task(...)

  This function chooses the most appropriate task eligible to run next.

- set_curr_task(...)

  This function is called when a task changes its scheduling class or changes
  its task group.

- task_tick(...)

  This function is mostly called from time tick functions; it might lead to
  process switch. This drives the running preemption.

GROUP SCHEDULER EXTENSIONS TO CFS

Normally, the scheduler operates on individual tasks and strives to provide
fair CPU time to each task. Sometimes, it may be desirable to group tasks and
provide fair CPU time to each such task group. For example, it may be
desirable to first provide fair CPU time to each user on the system and then to
each task belonging to a user.

CONFIG_CGROUP_SCHED strives to achieve exactly that. It lets tasks to be
grouped and divides CPU time fairly among such groups.

CONFIG_RT_GROUP_SCHED permits to group real-time (i.e., SCHED_FIFO and
SCHED_RR) tasks.

CONFIG_FAIR_GROUP_SCHED permits to group CFS (i.e., SCHED_NORMAL and
SCHED_BATCH) tasks.

   These options need CONFIG_CGROUPS to be defined, and let the administrator
   create arbitrary groups of tasks, using the "cgroup" pseudo filesystem. See
   Documentation/cgroup-v1/cgroups.txt for more information about this
filesystem.

When CONFIG_FAIR_GROUP_SCHED is defined, a "cpu.shares" file is created for each
group created using the pseudo filesystem. See example steps below to create

task groups and modify their CPU share using the "cgroups" pseudo filesystem.

```
# mount -t tmpfs cgroup_root /sys/fs/cgroup
# mkdir /sys/fs/cgroup/cpu
# mount -t cgroup -ocpu none /sys/fs/cgroup/cpu
# cd /sys/fs/cgroup/cpu

# mkdir multimedia     # create "multimedia" group of tasks
# mkdir browser        # create "browser" group of tasks

# #Configure the multimedia group to receive twice the CPU bandwidth
# #that of browser group

# echo 2048 > multimedia/cpu.shares
# echo 1024 > browser/cpu.shares

# firefox &     # Launch firefox and move it to "browser" group
# echo <firefox_pid> > browser/tasks
```

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.