Modified code are UNDERLINED，BOLD！！！

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>

// Simplifed xv6 shell.

#define MAXARGS 10
#define MAXCMDS 10

// All commands have at least a type. Have looked at the type, the code
// typically casts the *cmd to some specific cmd type.
struct cmd {
    int type;          //  ' ' (exec), | (pipe), '<' or '>' for redirection
};

struct execcmd {
    int type;              // ' '
    char *argv[MAXARGS];   // arguments to the command to be exec-ed
};

struct redircmd {
    int type;          // < or >
    struct cmd *cmd;   // the command to be run (e.g., an execcmd)
    char *file;        // the input/output file
    int mode;          // the mode to open the file with
    int fd;            // the file descriptor number to use for the file
};

struct pipecmd {
    int type;          // |
    struct cmd *left;  // left side of pipe
    struct cmd *right; // right side of pipe
};

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

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

    if(cmd == 0)
        exit(0);
```

```
  switch(cmd->type){
    default:
      fprintf(stderr, "unknown runcmd\n");
//      exit(-1);

    case ' ':
      ecmd = (struct execcmd*)cmd;
//      fprintf(stderr, "%s",ecmd->argv[0]);


      if (fork1() == 0) {
        execvp(ecmd->argv[0], ecmd->argv);
      }
      wait(&r);
      break;

    case '>':
    case '<':
      rcmd = (struct redircmd*)cmd;
      fprintf(stderr, "redir not implemented\n");
      // Your code here ...
      runcmd(rcmd->cmd);
      break;

    case '|':
      pcmd = (struct pipecmd*)cmd;
      fprintf(stderr, "pipe not implemented\n");
      // Your code here ...
      break;
  }
  // exit(0);
}

int
getcmd(char *buf, int nbuf)
{

  if (isatty(fileno(stdin)))
    fprintf(stdout, "$ ");
  memset(buf, 0, nbuf);
  fgets(buf, nbuf, stdin);
  if(buf[0] == 0) // EOF
    return -1;
  return 0;
}

int
main(void)
{
  static char buf[1000];
  int fd, r;

  // Read and run input commands.
  while(getcmd(buf, sizeof(buf)) >= 0){
    if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
      // Clumsy but will have to do for now.
```

```
        // Chdir has no effect on the parent if run in the child.
        buf[strlen(buf)-1] = 0;  // chop \n
        if(chdir(buf+3) < 0)
          fprintf(stderr, "cannot cd %s\n", buf+3);
        continue;
      }

      char *sbuf,*ebuf; //start and end pointer of the buf
      sbuf=buf;
      ebuf=buf;
      char *es = buf + strlen(buf);
    if(fork1() == 0){
      while(ebuf<=es){

        while(!peek(&ebuf,es,":")&&ebuf<es)
          ebuf++;

        if(*ebuf==':'|ebuf==es){
          char bufpiece[100];  //bufpiece between sequence symbol ":bufpiece:bufpiece:..."

          memset(bufpiece, 0, sizeof(bufpiece));

          memcpy(bufpiece, sbuf, ebuf-sbuf);
          if(ebuf==es)
            bufpiece[strlen(bufpiece)-1] = 0;  // chop \n
          char *ssubbuf,*subcmd;  //start and end pointer of the bufpiece
          ssubbuf=bufpiece;
          char * esubbuf=ssubbuf+strlen(ssubbuf);

          if(*(esubbuf-1)=='&')
          {
            fprintf(stderr, "illigel input command\n");
            break;
          }
          //parse &
          while((parseParrel(&ssubbuf,&subcmd,esubbuf)))
          {
            if(fork1()==0)
            {

              //run the left side command of &
              runcmd(parsecmd(subcmd));
              exit(0);
            }
          }

          //run the right side command of &
          runcmd(parsecmd(ssubbuf));
          wait(&r);
        }
        wait(&r);
        ebuf++;
        sbuf=ebuf;
      }
      exit(0);
    }
```

```
    wait(&r);
  }
}

int parseParrel(char **ps,char **subcmd,char * es)
{
  char *s;
  s=*ps;

  while(!peek(&s,es,"&"))
  {
    s++;
    if(s>=es) break;
  }
  if(s>=es) return 0;

  *subcmd = malloc(sizeof(char) * (s-*ps));
  memcpy(*subcmd, *ps, s-*ps);
  *ps=s+1;
  return 1;

}
int
fork1(void)
{
  int pid;

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

struct cmd*
execcmd(void)
{
  struct execcmd *cmd;

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

struct cmd*
redircmd(struct cmd *subcmd, char *file, int type)
{
  struct redircmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = type;
  cmd->cmd = subcmd;
  cmd->file = file;
  cmd->mode = (type == '<') ?  O_RDONLY : O_WRONLY|O_CREAT|O_TRUNC;
  cmd->fd = (type == '<') ? 0 : 1;
```

```
    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 = '|';
    cmd->left = left;
    cmd->right = right;
    return (struct cmd*)cmd;
}

// 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 '<':
            s++;
            break;
        case '>':
            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*);

// make a copy of the characters in the input buffer, starting from s through es.
// null-terminate the copy to make it a string.
char
*mkcopy(char *s, char *es)
{
    int n = es - s;
    char *c = malloc(n+1);
    assert(c);
    strncpy(c, s, n);
    c[n] = 0;
    return c;
}

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

    es = s + strlen(s);
    cmd = parseline(&s, es);
    peek(&s, es, "");
    if(s != es){
        fprintf(stderr, "leftovers: %s\n", s);
        exit(-1);
    }
    return cmd;
}

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

    struct cmd *cmd;
    cmd = parsepipe(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') {
            fprintf(stderr, "missing file for redirection\n");
            exit(-1);
        }
        switch(tok){
            case '<':
                cmd = redircmd(cmd, mkcopy(q, eq), '<');
                break;
            case '>':
                cmd = redircmd(cmd, mkcopy(q, eq), '>');
                break;
        }
    }
    return cmd;
}

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

    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') {
            fprintf(stderr, "syntax error\n");
            exit(-1);
```

```
        }
        cmd->argv[argc] = mkcopy(q, eq);
        argc++;
        if(argc >= MAXARGS) {
            fprintf(stderr, "too many args\n");
            exit(-1);
        }
        ret = parseredirs(ret, ps, es);
    }
    cmd->argv[argc] = 0;
    return ret;
}
```