```c
/********** sh.c file *************/
#include "ucode.c"

//extern int in, out, err;

char cmdline[100];
char commandLine[100];
char *_argv[10];//first argv

int do_IOredirection(int i);
int do_pipe(int i);

int main(int argc, char *argv[ ])
{
  int i = 0;
  int ioNeeded = 0; // flag for whether io redirection is needed
  int pid = getpid();
  printf("Yuzhu: PROC %d running shell program\n", pid);

  // printf("argc = %d\n", argc);
  // for (i=0; i<argc; i++)
  //   printf("argv[%d] = %s\n", i, argv[i]);

  while(1)
  {
        printf("Enter command>");
        gets(cmdline);
        interpreter(cmdline);//interpret the command
  }

  printf("PROC %d exit\n", pid);
}

int interpreter(char *cmdline)
{
        //function to interpret command
        if(strcmp(cmdline,"") ==0 )
                continue;

        cmdline[strlen(cmdline)] = 0;//kill the '\n'
        strcpy(commandLine,cmdline);//copy the command line
        parseCmdHelper(cmdline);//parse the command line
        i = 0;
        ioNeeded = 0;//set the flag for io redirection to 0

        if(strcmp(_argv[0],"cd") != 0 && strcmp(_argv[0],"logout") != 0){
                //detect I/0 redirection

                while(i < 10)
                {
                        if(_argv[i] == 0)
                        {
                                break;//get out of while loop
                        }
                        if(strcmp(_argv[i],">") == 0 || strcmp(_argv[i],"<") == 0 ||
strcmp(_argv[i],">>") == 0)
                        {
                                //detected I/O redirection
                                pid = fork();
                                if(pid)
                                {
                                        wait(&pid);
```

```
                                }
                                else
                                {
                                        //child process
                                        do_IOredirection(i);
                                        ioNeeded = 1;
                                }
                                ioNeeded = 1;
                        }
                        if(strcmp(_argv[i],"|") == 0)
                        {
                                //detected pipe symbol
                                pid = fork();
                                if(pid)
                                {
                                        //parent
                                        wait(&pid);
                                }
                                else
                                {
                                        //to do: we need to
                                        do_pipe(i);
                                        ioNeeded = 1;//set io needed flag
                                }
                        }
                        i++;//increment i
                }

                if(!ioNeeded)
                {
                        pid = fork();
                        if(pid)
                        {
                                //parent
                                wait(&pid);//parent waits
                        }
                        else
                        {
                                //child
                                exec(commandLine);//child execute the command
                                //exit(0);
                        }
                }


        }
        else if(strcmp(_argv[0],"logout") == 0)
        {
                prints("shell runs logout\n");
                break;//get out of the while loop
        }
        else
        {
                prints("running cd\n");
                chdir(_argv[1]);
        }
}

int parseCmdHelper(char *line)
{
  prints("parsing command...\n");
  char *cp = line;
```

```
  argc = 0;
  while (*cp != 0){
    while (*cp == ' ')  *cp++ = 0;
    if (*cp != 0)// skip over blanks // token start
      _argv[argc++] = cp; // pointed by argv[ ]
    while (*cp != ' ' && *cp != 0) // scan token chars
      cp++;
    if (*cp != 0)
      *cp = 0;
    else // end of token
      break; // end of line
    cp++;// continue scan
  } //end outer while
  _argv[argc] = 0; // argv[argc]=0
}

int do_IOredirection(int i)
{
        //suppose we are only doing a simple I/O redirection like : cat f1 > f2
        int fd = -1;//file descriptor I/O redirection
        int pid;
        int status;

        prints("doing I/O redirection\n");
        char cmd[50]="";//command for before the I/O redirection symbol
        for(int j = 0 ; j < i ;j++)
        {
                strcat(cmd,_argv[j]);
                strcat(cmd," ");//add a space character
        }
        cmd[strlen(cmd)-1] = 0;//kill the last space character
        printf("cmd=%s\n",cmd);
        char outfile[50]="";//command after the I/O rediection symbol
        if(_argv[i+1] != 0)
        {
                strcpy(outfile,_argv[i+1]);
                printf("outfile=%s\n",outfile);
                //start I/O redirection
                //bug arises here.
                fd = open(outfile,O_WRONLY | O_CREAT);//open file descriptor
                if(fd < 0)
                {
                        printf("open failed\n");
                        return -1;
                }
                printf("the opened fd=%d\n",fd);

                close(1);//close file descriptor 1
                dup(fd);//duplicate file descriptor
                close(fd);
                /*
                //fork a child to do IO
                pid = fork();
                if(pid)
                {
                        //parent
                        wait(&pid);
                        prints("file descriptor opened\n");
                        close(1);//close file descriptor 1
                        int out2 = open("/dev/tty0", O_WRONLY); // //reopen the
stdout file descriptor
                        printf("out2=%d\n",out2);
```

```
                        printf("fd=%d\n",fd);
                        close(fd);
                }
                else
                {
                        exec(cmd);
                        exec("exit");
                }
                */

                exec(cmd);//execute the command: using printf() to print stuff to the
outfile
                //prints("file descriptor opened\n");
                close(1);//close file descriptor 1
                int out2 = open("/dev/tty0", O_WRONLY); // //reopen the stdout file
descriptor
                printf("out2=%d\n",out2);
                //printf("fd=%d\n",fd);
                //close(fd);


        }
        else
        {
                prints("No output file specified\n");
                return fd;
        }
        return 0;
}

int do_pipe(int i)
{
        //do_pipe algorithm
        char cmd1[50]="";//command  before the vertical bar
        char cmd2[50] = "";//command after the vertical bar
        for(int j = 0 ; j < i ;j++)
        {
                strcat(cmd1,_argv[j]);
                strcat(cmd1," ");//add a space character
        }
        cmd1[strlen(cmd1)-1] = 0;//kill the last space character
        printf("cmd1=%s\n",cmd1);

        for(int j = i+1 ; j < 10 ; j ++)
        {
                if(_argv[j] == 0)
                        break;
                strcat(cmd2,_argv[j]);
                strcat(cmd2," ");//add space
        }
        cmd2[strlen(cmd2)-1] = 0;//kill the space character
        printf("cmd2=%s\n",cmd2);

        int pid, pd[2];
        pipe(pd); //create a pipe: pd[0]

        pid = fork(); //fork a child to share the pipe
        if(pid)
        {
                //parent : as the pipe reader
                close(pd[1]); //close pipe WRITE end
                dup2(pd[0],0); //redirect stdin to pipe READ end;
```

```
                exec(cmd2);
        }
        else
        {
                close(pd[0]); //close pipe READ end
                dup2(pd[1],1); // redirect stdout to pipe WRITE end
                exec(cmd1);
        }

}
```