Unix-like Shell Project Paper

In this proect, I implemented all the functions.

Login

Overview of Login

I have a function `login` called in the `main` function after initializing the jobs list, because I also need to add a **shell** `job` in the jobs list, in order to manage the entry under `proc`. The `login` function return the username that read from the stdin and assign to global variable `username`.

```
/* Initialize the job list */
initjobs(jobs);

/* Have a user log into the shell */
username = login();
  if (username == NULL) {
  return 0;
}
addentry("shell", getpid(), SsF);
addjob(jobs, getpid(), BG, "shell\n");
```

Above is the structure of the `login` function:

There is one big infinity loop, we jump out of the loop only if we enter `quit` to the username or enter the correct username and password.

In the loop, first, we read the username, password from the stdin, and open the `etc/passwd` file. Then keep comparing all the record in the `passwd` file.

if their is a match then it will return username and back to main function begin the shell; otherwise the shell responds with `"User Authentication failed. Please try again."`.

```
/*
* login - Performs user authentication for the shell
    * This function returns a string of the username that is logged in
    */
    char * login() {
        while (1) {
        /* Read username */
        ...
        if (strcmp(name, "quit") == 0) {
            return NULL;
        }

        /* Read password */
        ...

        /* Open passwd file */
        ...

        /* Read passwd file line by line */
        while((read = getline(&line, &len, fp)) != -1) {
            ...
            if (strcmp(recordValues[0], name) == 0) {   /* Find the username*/
                if (strcmp(recordValues[1], password) == 0) { /* password is correct*/
                    return name;
                }
            }
        }
        ...
        printf("User Authentication failed. Please try again.\n");
        }
    }
```

Read username and password part

Here is the code of reading username and password, we need to remove the `\n` in the `name` and `password` in order to match the record in `passwd` file.

If the `name` is `quit` then we need to return NULL and which terminates the shell in the `main` function.

```
/* Read username */
char *name = (char *) malloc(sizeof(char) * 40);
printf("username: ");
fgets(name, sizeof(name), stdin);
name[strcspn(name, "\n")] = 0;  // remove the "/n"
if (strcmp(name, "quit") == 0) {
    return NULL;
}

/* Read password */
char password[40];
printf("password: ");
fgets(password, sizeof(password), stdin);
password[strcspn(password, "\n")] = 0;  // remove the "/n"
```

Open and Read passwd file

Here is the code of opening and reading `passwd` file to perform user authentication by verifying the username and password matches one inside the `etc/passwd` file.

First, we open the `passwd` file, then read the file line by line and compare the record in it to verify the username and password. If mathes one, we close the file and return the username; otherwise, we responds with `"User Authentication failed. Please try again."` And will continuously keep asking the user to login.

```
/* Open passwd file */
FILE *fp = fopen("etc/passwd", "r");
if (fp == NULL){
    printf("file open failed!\n");
    exit(EXIT_FAILURE);
}
/* Read passwd file line by line */
int find = 0;
while((read = getline(&line, &len, fp)) != -1) {
    char *recordValues[30];
    char recordString[100];
    strcpy(recordString, line);
    convertRecord(recordString, recordValues);

    /* verifying the username and password*/
    if (strcmp(recordValues[0], name) == 0) {   /* Find the username*/
        if (strcmp(recordValues[1], password) == 0) { /* password is correct*/
            fclose(fp);
            return name;
        }
    }
}
fclose(fp);
if (line)
    free(line);

printf("User Authentication failed. Please try again.\n");
```

Add new user

Overview of adduser

`adduser` is the built-in command, here is the structure of `adduser` function.

```
/*
* adduser -  Creates a new user for the shell
    *  This function can only be done if the root user is logged in.
    */
    void adduser(char **argv) {
        /* Check the validation of argv */
        ...

    /* Crate a new home directory */
    ...

    /* Create .tsh_history file */
    ...

    /* Create an entry for the new user inside the etc/passwd file */
    ...
    }
```

Check the validation of argv

In this part, we need to make sure is user is `root` and we have both `username` and `password` argument.

```
/* Check the validation of argv */
if (strcmp(username, "root") != 0) {    /* If it's not root user*/
    printf("root privileges required to run adduser.\n");
    return;
}

if (argv[1] == NULL) {  /* Missing new_username */
    printf("%s command requires a new username\n", argv[0]);
    return;
} else if (argv[2] == NULL) {   /* Missing new_password */
    printf("%s command requires a new password\n", argv[0]);
    return;
}
```

Create a new home directory

In this part, we create a new home directory (i.e., `home/new_username`)

```
/* Crate a new home directory */
char new_directory[40];
strcpy(new_directory, "home/");
strcat(new_directory, argv[1]);
struct stat st = {0};
if (stat(new_directory, &st) == -1) { /* If user not exists, create*/
    mkdir(new_directory, 0700);
} else {    /*if user already existed*/
    printf("User already exists!\n");
    return;
}
```

Create history and password file

In this part, we create an empty `.tsh_history` file under the new home directory and an entry for the new user inside the `etc/passwd` file

```
/* Create .tsh_history file */
strcat(new_directory, "/.tsh_history");
fclose(fopen(new_directory, "a"));

/* Create an entry for the new user inside the etc/passwd file */
FILE *fp = fopen("etc/passwd", "a");
fprintf(fp, "%s:%s:/home/%s\n", argv[1], argv[2], argv[1]);
fclose(fp);
```

Command Evaluation

Implementation of `eval` function

The structure of `eval` function is similar to the code in **m7.pdf** page **40**.

- First we need to initialize the signal and do some preparation, which we will talk later about in next section.
- Then we need to copy the cmdline to a buffer, because it is `char *` we may modify the the address of it when we call `parseline`function.
- Next we execute the function, If the first word is a built-in command, the shell immediately executes the command in the current process, the `builtin_cmd` function will do this. Otherwise, the shell forks a child process, then loads and runs the program in the context of the child.

  - By checking the `bg` whcih `parseline` returns, if it's *foreground* job, the shell waits for the job to terminate before awaiting the next command line.
  - If it's *backgroud* job, the shell does not wait for the job to terminate before printing the prompt and awaiting the next command line
- Finally, we call `savecmd` to save the `cmdline`.

```
void eval(char *cmdline)
{
    /* Initialization and preparation for signal */
    ...

    /* Copy the cmdline to a buffer, science we may modify the buffer */
    ...

    bg = parseline(buf, argv);

    /* Execute the command */
    if (!builtin_cmd(argv)) {
        /* Not build in command  */
       ...
        if (!bg) { /* Parent waits for foreground job to terminate */
            ...
        } else { /* The job run in background */
            ...
        }
    }

    /* Save command line */
    savecmd(cmdline);
    return;
}
```

Interaction with signal handlers and job control

As we talked before, we need to initialize the signal and do some preparation. We have a **sigset** `mask_all` that can block all the **signal**, and a **signet** that `mask_one` that only has `SIGCHILD`.

```
/* Initialization and preparation for signal */
sigset_t mask_all, mask_one, prev_one;

sigfillset(&mask_all);  // Add every signal number to set
sigemptyset(&mask_one); // Create empty set
sigaddset(&mask_one, SIGCHLD);  // Add SIGCHLD to the previous empty one
signal(SIGCHLD, sigchld_handler);   // install the child signal handler
```

Interaction in Executing the command

In the `builtin_cmd` function, we don't need to write code to interaction with signal, we let the function in it to do this for us. So if it's not build in comand, here is the interaction with signal handlers and job control:

We need to block `SIGCHILD`, because the signal might interfere our following action. Then we need to `fork` a job, also as mentioned in the **Hints & Tips**, the child process should call `setpgid(0,0)`, which ensures that there will be only one process, your shell, in the foreground process group. Then unblock the signal. I also has use `execve` function to check if the command can be found.

```
 /* Not build in command  */
sigprocmask(SIG_BLOCK, &mask_one, &prev_one);   /* Block SIGCHLD */

if ((pid = fork()) == 0) {  /* Child runs user job */
  if (setpgid(0, 0) < 0) {   /* Puts the child in a new process group */
    unix_error("setpgid error");
  }
  sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */

  if (execve(argv[0], argv, environ) < 0) {
    printf("%s: Command not found.\n", argv[0]);
    exit(0);
  }
}
```

Interaction in Handling forground and background job

In this part, the foreground and backgound job both need to block and unblock the signal which same as the above code, and add job and entry. The only difference is that we need to up date the shell **STAT** from **Ss+** to **Ss** becasue shell is not foreground job now.

```
if (!bg) { /* Parent waits for foreground job to terminate */
  sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
  addjob(jobs, pid, FG, cmdline );
  addentry(argv[0], pid, FG);
  update_shell_status(SsB);   // Update shell STAT to Ss
  sigprocmask(SIG_SETMASK, &prev_one, NULL);  /* Unblock SIGCHLD */

    waitfg(pid)
  waitfg(pid);

} else { /* The job run in background */
  sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
  addjob(jobs, pid, BG, cmdline );
  addentry(argv[0], pid, BG);
  sigprocmask(SIG_SETMASK, &prev_one, NULL);  /* Unblock SIGCHLD */
}
```

Built-in commands

`quit`

Including one helper function `deleteentry`, which can help me delete all entry under `proc` folder and kill all the jobs.

```
/* clear the jobs */
for (int i = 0; i < MAXJOBS; i++) {
  deleteentry(jobs[i].pid);
}
```

Here is the helper function, because in `initjobs` , `pid ==0` means emtpy, we need to ignore the job that `pid <1`. Then we use `kill` function to kill the job, there also is a helper function `remove_directory` that can delete the whole folder, I put the code in the **Appendix**.

```
void deleteentry(pid_t pid) {
    if (pid < 1)
        return;
    char entry[40];
    sprintf(entry, "proc/%d", pid);
    kill(-pid, SIGINT);
    remove_directory(entry);
}
```

`logout`

This function is implemented in one big `for` loop, keep checking if there are suspended jobs, and call `deleteentry` on other jobs.

```
void logout(struct job_t *jobs) {
    for (int i = 0; i < MAXJOBS; i++) {
        if (jobs[i].state == ST) {
            printf("There are suspended jobs.\n");
            return ;
        } else {
            deleteentry(jobs[i].pid);
        }
    }
    exit(0);
}
```

history

Get the position of `.tsh_history` by `username`, and then read the file line by line.

```
void history() {
    /* history file path */
    char hist_file[40];
    sprintf(hist_file, "home/%s/.tsh_history", username);

    /* Try open history file */
    FILE *fp = fopen(hist_file, "r");
    if (fp == NULL) {
        printf("Cannot open %s's history file\n", username);
        exit(-1);
    }

    char *line = NULL;
    long int len = 0;
    int index = 0;
    while (getline(&line, &len, fp) != -1) {
        index++;
        printf("%d %s", index, line);
    }

    fclose(fp);
}
```

`jobs`

Iterate the `jobs` array and print the job `jid`, `pid` and `state` , with the specific `cmdline`.

```
void listjobs(struct job_t *jobs)
{
    int i;

    for (i = 0; i < MAXJOBS; i++) {
  if (jobs[i].jid != 0) {
      printf("[%d] (%d) ", jobs[i].jid, jobs[i].pid);
      switch (jobs[i].state) {
    case BG:
        printf("Running ");
        break;
    case FG:
        printf("Foreground ");
        break;
    case ST:
        printf("Stopped ");
        break;
      default:
        printf("listjobs: Internal error: job[%d].state=%d ",
         i, jobs[i].state);
      }
      printf("%s", jobs[i].cmdline);
  }
    }
}
```

`!N`

There is a helper function `countlines` which can return the number of line in `.tsh_history` file, it's not complicated so I put the code in Appendix.

In the `rerun_N` function, we first need to check if the line number `n` is valid. Then we open the `.tsh_history` file and use index to track the lines number, then call `eval` to rerun the *Nth* command again.

```
int rerun_N(char *command) {
    int n = atoi(&command[1]);
    if (n < 1 || n > countlines()) {
        printf("Line number is invalid\n");
        return 0;
    } else {
        /* history file path */
        /* history file path */
        char hist_file[40];
        sprintf(hist_file, "home/%s/.tsh_history", username);

        /* Try open history file */
        FILE *fp = fopen(hist_file, "r");
        if (fp == NULL) {
            printf("Cannot open %s's history file\n", username);
            exit(-1);
        }

        char *line = NULL;
        long int len = 0;
        int index = 0;
        while (getline(&line, &len, fp) != -1) {
            index++;
            if (index == n) {   /* find the Nth command line */
                eval(line);
            }
        }
        fclose(fp);
        return 1;
    }
}
```

`bg<job>` & `fg<job>`

In this function, we first need to check if the arguments are valid. Then we if the argument is **JID**, we call `getjobjid` to get the **job**, if it's **PID**, we can `getjobpid` to get the **job**, these two helper function are provided.

After we get the **job**, we need to determine whether it's background or foreground. We use `kill` and `SIGCONT` signal to continue the job in both situations. We also need to update the `job` status, and use `update` helper function to update the `STAT` field in `status` file under `proc/PID/` in both situations. The differences are that we need to use helper function `update_shell_status` update the `status` of `shell` if it is foreground command, because the `shell` is no longer foreground. We still use a helper function `waitfg` to wait the job finish.

For the two helper function `update` and `update_shell_status` we will talk about later.

```
void do_bgfg(char **argv)
{
    if (argv[1] == NULL) {  /* Missing pid or jid */
        printf("%s command requires a PID or %%jobid argument\n", argv[0]);
        return;
    }
    }

    if (!isdigit(argv[1][0]) && argv[1][0] != '%') {            /* If the second argument is invlid */
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }
    struct job_t *job;
    if (argv[1][0] == '%') {    /* JID */
        job =  getjobjid(jobs, atoi(&argv[1][1]));  // Get job by jid
        if (job == NULL) {  /* If job doesn't exist */
            printf("%s: No such job\n", argv[1]);
            return;
        }

    } else {    /* PID */
        job =  getjobpid(jobs, (pid_t) atoi(argv[1]));      //Get job by pid
        if (job == NULL) {  /* If job doesn't exist */
            printf("%s: No such process\n", argv[1]);
            return;
        }
    }

    if (strcmp(argv[0], "bg") == 0) {   /* If it's background command*/
        update(job->pid, job->state, BG);
        job->state = BG;
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
        kill(-(job->pid), SIGCONT);
    } else {    /* If it's foreground command */
        update(job->pid, job->state, FG);
        update_shell_status(SsB);
        job->state = FG;
        kill(-(job->pid), SIGCONT);
        waitfg(job->pid);
    }

    return;
}
```

In `waitfg`, the code is simple, implementing by one `while` loop keeps checking if the `pid` is equal to the foreground job's pid. If they are not equal means the current foreground job is finished.

```
void waitfg(pid_t pid)
{
    while(1) {
        if (pid != fgpid(jobs)) {
            break;
        } else {
            sleep(0.2);
        }

    }
  return;
}
```

`adduser`

Though this is a built in function, but we already talked about it in the **Login** secion.

Proc

Creating entry

For creating `proc/PID/status`, I use a function called `addentry` to implemented adding files in proc. The first thing we need to do is when shell starts run, adding the shell to `proc`, because we treat shell also as a job. The shell will always be **Ss** or **Ss+**, therefore I added more state using `#define`. I call `addentry` whenever `addjob` is called.

In this function, we take the following arguments： `name` which is `agrv[0]`, `pid` and `state` which is determined by what situation the function is called.

We first creat the entry by `pid` under `proc/`, then we create the `status` file under `proc/PID/`. In `status` file, first line, we write the name of process which is `argv[0]`. Then write the `Pid`, `PPid`, `PGid`, `Sid`,`STAT`, and `Username` to the file. For `Ppid`, I used `getppid()` function, for `PGid` i used `getpgid(pid)`.

```
void addentry(char * name, pid_t pid, int state) {
    if (pid < 1)
        return;

    /* Create new PID entry */
    char new_entry[40];
    sprintf(new_entry, "proc/%d", pid);
    struct stat st = {0};
    if (stat(new_entry, &st) == -1) { /* If PID entry not exists, create*/
        mkdir(new_entry, 0700);
    }

    /* Create status file */
    strcat(new_entry, "/status");
    FILE *fp = fopen(new_entry, "a");
    if (fp == NULL) {
        printf("Cannot create %d status file", pid);
        exit(-1);
    }

    fprintf(fp, "Name: %s\n", name);

    fprintf(fp, "Pid: %d\n", pid);
    if (strcmp(name, "shell") == 0) {
        fprintf(fp, "PPID: %d\n", getppid());
    } else {
        fprintf(fp, "PPID: %d\n", getpid());
    }

    fprintf(fp, "PGID: %d\n", getpgid(pid));
    fprintf(fp, "SID: %d\n", getpid());
    if (state == FG) {
        fprintf(fp, "STAT: R+\n");
    } else if (state == BG) {
        fprintf(fp, "STAT: R\n");
    } else if (state == SsF) {
        fprintf(fp, "STAT: Ss+\n");
    } else if (state == SsB) {
        fprintf(fp, "STAT: Ss\n");
    } else {
        printf("Unkown state");
    }
    fprintf(fp, "Username: %s\n", username);

    fclose(fp);
}
```

Updating entry

Updating entry

For updating the `proc/PID/status`, now we are going to talk about the two helper function `update` and `update_shell_status`.

We first talked about `update_shell_status()`, because the `status` file of shell is different which maybe influenced by other jobs, for example, if one job changes from background to foreground then the `state` filed in shell's `status` file also need to be changed.

Because we cannot modify specific line in a file using C programming, therefore I delete the shell `status` file and call `addentry()` to create a new one with the new `state`.

```c
void update_shell_status(int state) {
    /* shell status file path*/
    char shell_status_file[40];
    sprintf(shell_status_file, "proc/%d/status", getpid());

    /* remove shell status file */
    char shell_rm_cmd[80];
    sprintf(shell_rm_cmd, "rm %s", shell_status_file);
    system(shell_rm_cmd);

    /* Create new shell status file*/
    addentry("shell", getpid(), state);

}
```

In the `update` function, we face the same issue that we cannot modify specific line in a file using C programming. Therefore, I need to remove the file and then create a new one. I use `sprintf()` to generate the **Unix commands** and use `system()` to execute **Unix commands**, **Unix commands** include:

- `sed` for substituting text and use data redirection to the `temp` file
- `rm` to remove the file
- `mv` to move the `temp` file to the `newfile`

Significantly, in `update()` , if the new `state` of job is `FG` I also need to change the `status` file of `shell` by using `update_shell_status()`, if the old `state` of job is `FG`, we also need to call `update_shell_status()`.

```c
void update(pid_t pid, int old, int new) {
    char new_state[10];
    char old_state[10];

    /* shell status file path*/
    char shell_status_file[40];
    sprintf(shell_status_file, "proc/%d/status", getpid());
    /* remove shell status file command */
    char shell_rm_cmd[80];
    sprintf(shell_rm_cmd, "rm %s", shell_status_file);

    if (new == BG) {
        strcpy(new_state, "STAT: R");
    } else if (new == FG) {
        strcpy(new_state, "STAT: R+");
        update_shell_status(SsB);
    } else if (new == ST) {
        strcpy(new_state, "STAT: T");
    }

    if (old == BG) {
        strcpy(old_state, "STAT: R");
    } else if (old == FG) {
        strcpy(old_state, "STAT: R+");
        update_shell_status(SsF);
    } else if (old == ST) {
        strcpy(old_state, "STAT: T");
    }

    /* update the status file */
    char status_file[40];
    sprintf(status_file, "proc/%d/status", pid);

    char sed_cmd[80], rm_cmd[80], mv_cmd[80];
    sprintf(sed_cmd, "sed 's/%s/%s/' %s > temp", old_state, new_state, status_file);
    sprintf(rm_cmd, "rm %s", status_file);
    sprintf(mv_cmd, "mv temp %s", status_file);

    system(sed_cmd);
    system(rm_cmd);
    system(mv_cmd);

    return;

}
```

Job Control

I already talked about `waitfg` and `do_bgfg` in build in **Build-in commands -> bg &fg** section. To avoid repetition, it's not described in this here.

sigchld

sigchld

First we set the signal and block, then if the child is foreground we update the `shell` to foreground. Then we need to check the status of child using `WIFEXITED` , `WIFSTOPPED` and `WIFSIGNALED`. With different status, we have different action. Then we unblock the signal.

- If the child terminated normally, we delete the job and entry under `proc`.
- If the child is stopped, we update the status
- If the child is terminated by catched signal, we print the signal and delete the job and entry under `proc`.

```
void sigchld_handler(int sig)
{   int olderrno = errno;   /* store old errno */
    int status; /* used to trace pid's status */
    sigset_t mask_all, prev_all;
    pid_t pid;
    sigfillset(&mask_all);

    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {  /* Reap child */
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all);

         /* If the job was foreground */
        if (getjobpid(jobs, pid)->state == FG) {
                update_shell_status(SsF);
        }

        int jid = pid2jid(pid);
        if (WIFEXITED(status)) {    /* If the child terminated normally */

            deletejob(jobs, pid);   /* Delete the child from the job list */
            deleteentry(pid);
        } else if (WIFSTOPPED(status)) {    /* If the child is stopped */
            update(pid, getjobpid(jobs, pid)->state, ST);
            getjobpid(jobs, pid)->state = ST;
            printf("Job [%d] (%d) Stopped by signal %d\n", jid, pid, WSTOPSIG(status));

        } else if (WIFSIGNALED(status)) {   /* Child is terminated by catched signal */
            printf("Job [%d] (%d) terminated by signal %d\n", jid, pid, WTERMSIG(status));
            deletejob(jobs, pid);
            deleteentry(pid);
        }

        sigprocmask(SIG_SETMASK, &prev_all, NULL);  /* Unblock SIGCHLD */
    }

    errno = olderrno;
    return;
}
```

sigint

This is simple, because this signal is going to be catched by the foreground job, we just need to use `fgpid` to get the foreground pid, and then use `kill` to signal, delete the job and entry under `proc`

```
void sigint_handler(int sig)
{
    pid_t pid = fgpid(jobs);
    if (pid != 0) {
        kill(-pid, sig);
        deletejob(jobs,pid);
        deleteentry(pid);
        printf("sigint_handler: Job [%d] and its entire foreground jobs with same process
group are killed\n", (int)pid);
    }

    return;
}
```

sigtstp

Similary to `sigint`, use `fgpid` to get the foreground pid, and use `kill` to signal.

```
void sigtstp_handler(int sig)
{
    pid_t pid = fgpid(jobs);
    if (pid != 0) {
        kill(-pid, sig); // signals to the entire foreground process group
        printf("sigtstp_handler: Job [%d] and its entire foreground jobs with same process
group are stoped\n", (int)pid);
    }
    return;
}
```

Questions

The most challenging aspect of the project:

The most challenging aspect of the project.

The signal handler is one of the most challenging aspect, what's more is that we need to modify the state of each job, but C programming doesn't support modify part of file. After trying many methods, I finnaly used **Unix Command** to implemented this.

Handle `mount` and `unmount`

We need to handle signal that realted to i/o and file, this might be one of challenges. What's more we need to manage the file system structure, we need to control the permission, we need to change the permission, treat device as files.

Implement pipes

I think we need to modify the `eval()`, in the past we only execute one command, but now we need to execute multiple command one by one. We can separate commands by `|`, then execute them one by one, and use the result of previous one as the next one.

Appendix

```c
int remove_directory(const char *path) {
   DIR *d = opendir(path);
   size_t path_len = strlen(path);
   int r = -1;

   if (d) {
      struct dirent *p;

      r = 0;
      while (!r && (p=readdir(d))) {
          int r2 = -1;
          char *buf;
          size_t len;

          /* Skip the names "." and ".." as we don't want to recurse on them. */
          if (!strcmp(p->d_name, ".") || !strcmp(p->d_name, ".."))
             continue;

          len = path_len + strlen(p->d_name) + 2;
          buf = malloc(len);

          if (buf) {
             struct stat statbuf;
             snprintf(buf, len, "%s/%s", path, p->d_name);
             if (!stat(buf, &statbuf)) {
                if (S_ISDIR(statbuf.st_mode))
                   r2 = remove_directory(buf);
                else
                   r2 = unlink(buf);
             }
             free(buf);
          }
          r = r2;
      }
      closedir(d);
   }

   if (!r)
      r = rmdir(path);

   return r;
}
```

```
int countlines() {
    /* path of history file */
    char hist_file[40];
    sprintf(hist_file, "home/%s/.tsh_history", username);

    /* Open and read history file */
    FILE *fp = fopen(hist_file, "r");
    if (fp == NULL) {
        printf("Cannot open %s's history file to caculate lines number\n", username);
        exit(-1);
    }

    int ch = 0;
    int lines = 0;
    while (!feof(fp)) {
        ch = fgetc(fp);
        if (ch == '\n') {
            lines++;
        }
    }
    fclose(fp);

    return lines;
}
```