ShellLab 实验报告

- 姓名：房向南。
- 学号：2022201560。

实验目的

通过编写一个简单的支持作业控制（job control）的 Unix shell 程序来熟悉进程控制和信号的概念。

实验内容

成果简述

- 完成baseline内容，trace01-16测试和tshref结果相同。
- 完成了一定程度上的错误检查。
- 完成了支持path环境变量的功能
- 完成了命令替换功能
- 完成了简单的重定向功能，支持>和>>的输出重定向
- 完成了简单的管道功能
- 完成了部分功能的混合支持
  注：以上内容在后续实验报告中会有详细解释说明。

baseline

命令提示符是tsh>
作业的前台运行，作业的后台运行：

```
myspin    output.txt  trace04.txt  trace10.txt  trace16.txt
myspin.c  README      trace05.txt  trace11.txt  tsh
tsh> echo $(ls) > output.txt
tsh> ./myspin 10
tsh> ./myspin 10 &
[2] (55011) ./myspin 10 &
tsh> ps
    PID TTY          TIME CMD
  54592 pts/0    00:00:00 bash
  54599 pts/0    00:00:00 tsh
  54618 pts/0    00:00:00 cat
  55011 pts/0    00:00:00 myspin
  55025 pts/0    00:00:00 ps
tsh>
```

crtlC，crtlZ终止或停止前台作业运行：

```
tsh> ./myspin 10
^CJob [2] (55053) terminated by signal 2
tsh> ./myspin 10
^ZJob [2] (55054) stopped by signal 20
tsh>
```

quit，jobs，bg，fg等内置命令：

```
tsh> jobs
[1] (54618) Stopped cat
[2] (55054) Stopped ./myspin 10
[3] (55072) Running ./myspin 20 &
[4] (55073) Running ./myspin 10 &
tsh> ./myspin 20 &
```

```
tsh> ./myspin 20 &
[3] (55141) ./myspin 20 &
tsh> fg %3
```

```
tsh> bg
bg command requires PID or %jobid argument
tsh> bg %3
%3: No such job
tsh> bg 10000
(10000): No such process
tsh> quit
yui@pc:~/Desktop/shlab-handout$
```

csapp实验指南里特别强调了信号的屏蔽和运用，特此列在这里。

```
if(!builtin_cmd(argv))
{//forking and execing a child fuction...
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    sigprocmask(SIG_BLOCK, &mask, NULL);
    char *executable;
    // 在 PATH 中搜索用户输入的命令对应的可执行文件
```

```
if((pid = fork()) == 0)
{//in child now
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    setpgid(0, 0);
    if(execv(executable,argv) < 0)
    {
        printf("Command not found!\n");
        exit(0);
    }
}
if(pathflag==1)free(executable);
addjob(jobs,pid,bg?BG:FG,cmdline);
sigprocmask(SIG_UNBLOCK, &mask, NULL);
if(!bg)
```

我把trace01-16的运行截图上传到一个公开的github仓库内，篇幅有限这里就不展示了。
仓库地址：shell lab

bonuswork

错误检查

实现了对于错误输入的检查，并将错误输出在标准输出上。如什么都没输入就按下回车的情况，即cmdline是'\0'，程序会直接返回。还有对输出重定向文件打开失败、PATH中找不到相关可执行文件、用户输入不存在的指令等情况都做了相应的报错。

```
void eval(char *cmdline)
            pathflag=1;
            executable = search_executable(argv[0]);
            if (executable == NULL) {
                printf("%s: command not found\n", argv[0]);
                return;
            }
        }else{
            executable=argv[0];
        }


        if((pid = fork()) == 0)
        {//in child now
            sigprocmask(SIG_UNBLOCK, &mask, NULL);
            setpgid(0, 0);
            if(execv(executable,argv) < 0)
            {
                printf("Command not found!\n");
                exit(0);
            }
        }
```

path环境变量

用parse_path函数在主函数中获取所有的环境变量，并赋值全局变量Paths。
在eval函数中fork出子进程之前先判断argv0能不能找到，如找不到就利用search_executable在环境变量里搜索到相关文件。

```
yui@pc:~/Desktop/shlab-handout$ ./tsh
tsh> ls
1.md      mysplit     sdriver.pl   trace06.txt  trace12.txt  tsh.c
Makefile  mysplit.c   trace01.txt  trace07.txt  trace13.txt  tshref
myint     mystop      trace02.txt  trace08.txt  trace14.txt  tshref.out
myint.c   mystop.c    trace03.txt  trace09.txt  trace15.txt
myspin    output.txt  trace04.txt  trace10.txt  trace16.txt
myspin.c  README      trace05.txt  trace11.txt  tsh
tsh> ps
    PID TTY          TIME CMD
  54592 pts/0    00:00:00 bash
  55331 pts/0    00:00:00 tsh
  55333 pts/0    00:00:00 ps
tsh> echo 11
11
tsh>
```

命令替换

用command_substitution函数在调用parseline之前先对cmdline进行$()形式的命令替换，可以支持一个命令行多个命令替换。

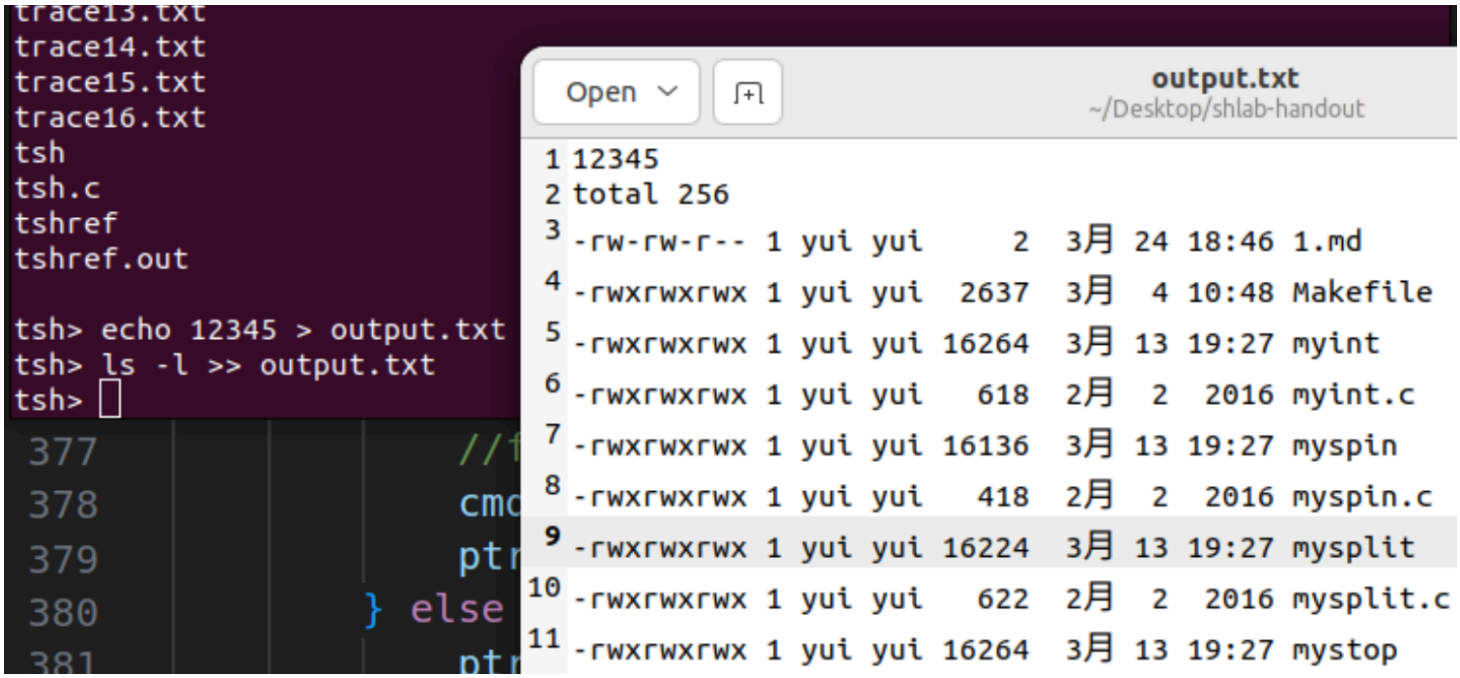
```
tsh> echo $(pwd)
/home/yui/Desktop/shlab-handout

tsh> echo $(ls)
1.md
Makefile
myint
myint.c
myspin
myspin.c
mysplit
mysplit.c
mystop
mystop.c
output.txt
README
sdriver.pl
trace01.txt
trace02.txt
trace03.txt
trace04.txt
trace05.txt
trace06.txt
trace07.txt
trace08.txt
trace09.txt
trace10.txt
trace11.txt
trace12.txt
trace13.txt
trace14.txt
```

```
tsh> echo $(pwd),$(ls)
/home/yui/Desktop/shlab-handout
,1.md
Makefile
myint
myint.c
myspin
myspin.c
mysplit
mysplit.c
mystop
mystop.c
output.txt
README
```

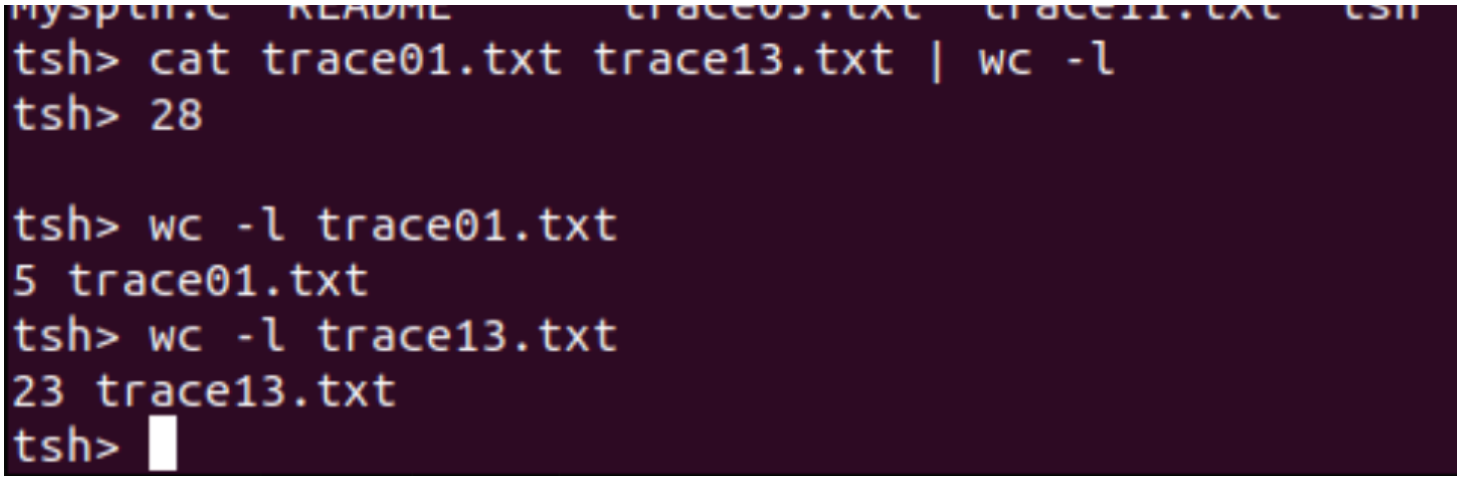
重定向

在分解完命令行以后，检查命令行中是否存在>或者>>，打开输出重定向的文件，将标准输出重定向到该文件。eval函数的最后重新改回标准输出，不然提示符tsh>不会正常输出。



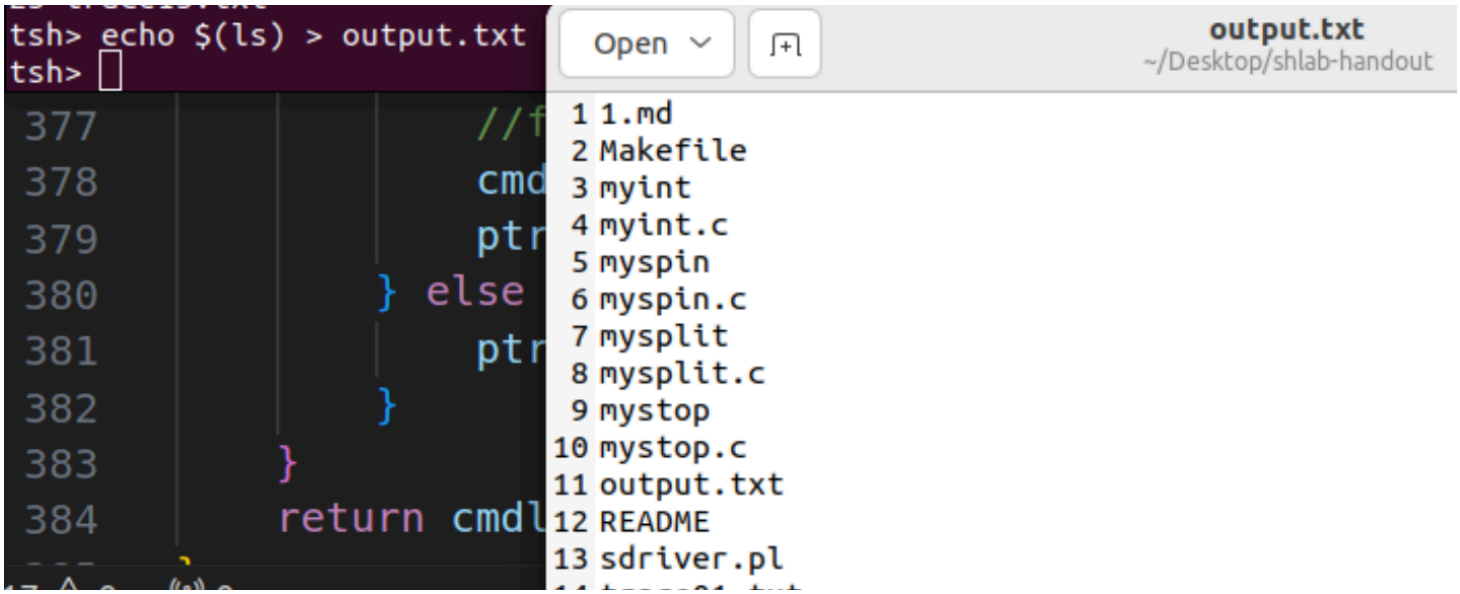
管道

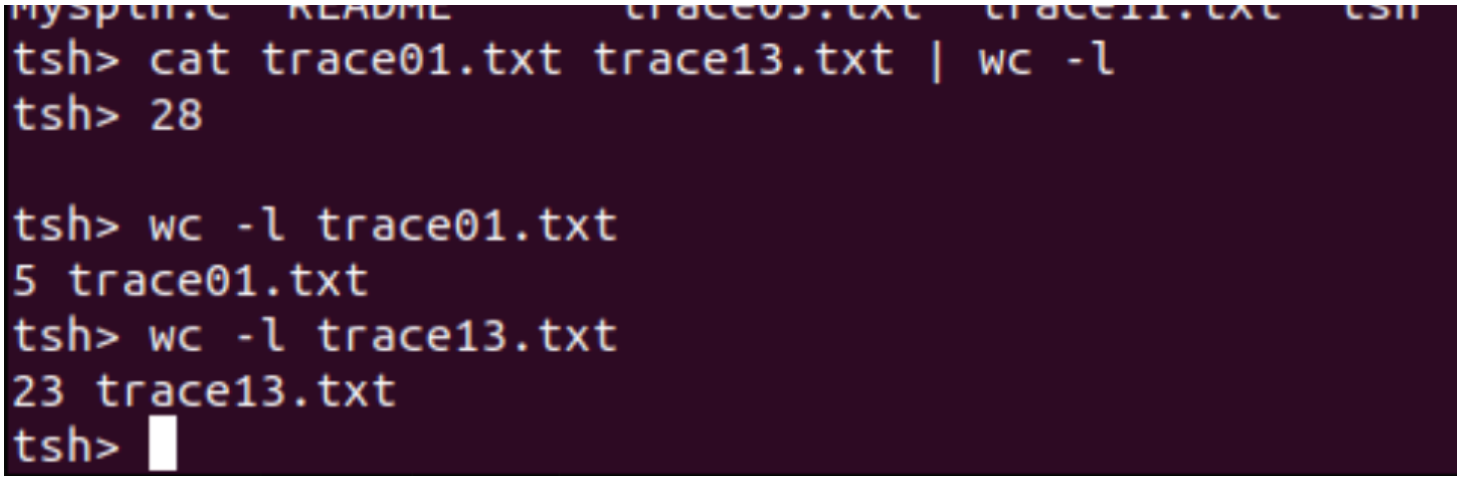
遍历cmdline看看是否存在'|'，若果存在就去另一个和eval基本相同但是支持管道功能的函数：evalpipe。因为部分指令tsh没办法运行，我只测试了少数指令。目前发现cat tracexx.txt tracexx.txt | wc -l 等指令是可以运行的。



简单的混合支持

1. 在命令替换中支持PATH环境变量。
2. 在管道中支持PATH环境变量。
3. 命令替换支持输出重定向。





实验总结以及参考资料说明

总结

1. 通过本实验，我更好的认识了unixshell的工作原理以及更好的掌握了进程控制、信号机制。
2. 本实验提高了我的代码水平，对于复杂代码的编写也有了一定经验。
3. 本实验和课本内容高度贴合，很好地检查了实验者对课本知识是否熟悉。
4. 本实验层次合理，难易结合。

参考资料

1. 本实验的完成参考了YouTube上的shell lab1，shell lab2。这位博主给我的实验了一个很好的开启思路，也介绍了不少的调错知识，但是不涉及很多的代码和可以“抄袭”的部分。

代码

```
// 解析 PATH 环境变量，返回一个包含目录路径的字符串数组
char **parse_path()
{
  char *path = getenv("PATH");
  if (path == NULL)
  {
    fprintf(stderr, "PATH environment variable is not set\n");
    exit(1);
  }

  char *token;
  char **paths = malloc(MAXARGS * sizeof(char *));
  if (paths == NULL)
  {
    perror("malloc");
    exit(1);
  }

  int i = 0;
  token = strtok(path, ":");
  while (token != NULL)
  {
    paths[i++] = token;
    token = strtok(NULL, ":");
  }
  paths[i] = NULL;

  return paths;
}

// 在指定路径中搜索指定命令名称对应的可执行文件
char *search_executable(char *cmd)
{
  char *executable = malloc(MAXLINE * sizeof(char));
  if (executable == NULL)
  {
    perror("malloc");
    exit(1);
  }
```

```
  int i = 0;
  while (paths[i] != NULL)
  {
    snprintf(executable, MAXLINE, "%s/%s", paths[i], cmd);
    if (access(executable, X_OK) == 0)
    {
      return executable; // 找到了可执行文件，返回其路径
    }
    i++;
  }

  free(executable);
  return NULL; // 未找到可执行文件
}
void evalpipe(char *cmdline)
{
  pid_t pid;
  sigset_t mask;
  int pipefd[2]; // 管道文件描述符数组
  char *left_argv[MAXARGS];
  char *right_argv[MAXARGS];
  volatile int pathflag = 0;
  int is_pipe = 0;                          // 是否存在管道
  int bg = parseline(cmdline, left_argv); // 解析命令行
  int saved_stdout;                         // 保存标准输出的文件描述符
  // 保存标准输出的文件描述符
  saved_stdout = dup(STDOUT_FILENO);
  // 检查命令行参数中是否包含管道符号
  for (int i = 0; left_argv[i] != NULL; i++)
  {
    if (strcmp(left_argv[i], "|") == 0)
    {
      is_pipe = 1;
      left_argv[i] = NULL; // 将管道符号替换为 NULL
      // 将管道符号后的参数作为右边的命令
      int j;
      for (j = 0; left_argv[i + 1 + j] != NULL; j++)
      {
        right_argv[j] = left_argv[i + 1 + j];
        left_argv[i + 1 + j] = NULL; // 清空左边命令的参数
      }
      right_argv[j] = NULL;
      break;
```

```
  }
}

// 创建管道
if (is_pipe && pipe(pipefd) < 0)
{
  fprintf(stderr, "Pipe creation failed\n");
  return;
}

if (!builtin_cmd(left_argv))
{
  sigemptyset(&mask);
  sigaddset(&mask, SIGCHLD);
  sigprocmask(SIG_BLOCK, &mask, NULL);
  if ((pid = fork()) == 0)
  {
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    setpgid(0, 0);
    if (is_pipe)
    {
      // 子进程执行左边的命令，将标准输出重定向到管道写端
      close(pipefd[0]); // 关闭管道的读端
      dup2(pipefd[1], STDOUT_FILENO);
      close(pipefd[1]); // 关闭管道的写端
    }
    char *executable;
    // 在 PATH 中搜索用户输入的命令对应的可执行文件
    if (access(left_argv[0], X_OK) != 0)
    {
      pathflag = 1;
      executable = search_executable(left_argv[0]);
      if (executable == NULL)
      {
        printf("%s: command not found\n", left_argv[0]);
        dup2(saved_stdout, STDOUT_FILENO);
        close(saved_stdout);
        return;
      }
    }
    else
    {
      executable = left_argv[0];
```

```
    }
    // 执行左边的命令
    execvp(executable, left_argv);
    fprintf(stderr, "Command not found: %s\n", left_argv[0]);
    exit(EXIT_FAILURE);
    if (pathflag == 1)
      free(executable);
  }
}
pathflag = 0;
if (!builtin_cmd(right_argv) && is_pipe)
{
  sigemptyset(&mask);
  sigaddset(&mask, SIGCHLD);
  sigprocmask(SIG_BLOCK, &mask, NULL);
  if ((pid = fork()) == 0)
  {
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    setpgid(0, 0);
    // 子进程执行右边的命令，将标准输入重定向到管道读端
    close(pipefd[1]); // 关闭管道的写端
    dup2(pipefd[0], STDIN_FILENO);
    close(pipefd[0]); // 关闭管道的读端
    // 执行右边的命令
    char *executable;
    // 在 PATH 中搜索用户输入的命令对应的可执行文件
    if (access(right_argv[0], X_OK) != 0)
    {
      pathflag = 1;
      executable = search_executable(right_argv[0]);
      if (executable == NULL)
      {
        printf("%s: command not found\n", right_argv[0]);
        dup2(saved_stdout, STDOUT_FILENO);
        close(saved_stdout);
        return;
      }
    }
    else
    {
      executable = right_argv[0];
    }
    execvp(executable, right_argv);
```

```
      fprintf(stderr, "Command not found: %s\n", right_argv[0]);
      exit(EXIT_FAILURE);
      if (pathflag == 1)
        free(executable);
    }
  }

  if (!is_pipe)
  {
    addjob(jobs, pid, bg ? BG : FG, cmdline);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    if (!bg)
    {
      waitfg(pid);
    }
    else
    {
      printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
    }
  }

  // 关闭管道两端的文件描述符
  if (is_pipe)
  {
    close(pipefd[0]);
    close(pipefd[1]);
  }
  dup2(saved_stdout, STDOUT_FILENO);
  close(saved_stdout);
  return;
}

// 在 cmdline 中执行命令替换
char *command_substitution(char *cmdline)
{
  char *ptr = cmdline;
  char *start_replace, *end_replace;

  while ((start_replace = strchr(ptr, '$')) != NULL)
  {
    if (*(start_replace + 1) == '(' && (end_replace = strchr(start_replace, ')')) != NULL)
    {
      *start_replace = '\0';              // 将 $ 替换为空字符，截断命令替换之前的部分
```

```
      *end_replace = '\0';                // 将命令替换的右括号替换为空字符，截断命令替换之后的部分
      char *command = start_replace + 2; // 获取命令替换的部分
      char result[MAXLINE];
      FILE *fp = popen(command, "r"); // 执行命令替换的命令
      if (fp == NULL)
      {
        printf("Error executing command substitution\n");
        return cmdline;
      }
      char *tempcmdline = malloc(strlen(cmdline) + MAXLINE); // 分配足够的空间来存储新的字符串
      if (tempcmdline == NULL)
      {
        printf("Memory allocation error\n");
        return cmdline;
      }
      strcpy(tempcmdline, cmdline); // 将原始字符串复制到新的字符串中
      while (fgets(result, MAXLINE, fp) != NULL)
      {
        // 将命令替换的结果插入到新的字符串中
        strcat(tempcmdline, result);
      }
      strcat(tempcmdline, end_replace + 1); // 恢复命令替换之后的部分
      pclose(fp);
      // free(cmdline); // 释放原始字符串的内存
      cmdline = tempcmdline; // 更新指针指向新的字符串
      ptr = cmdline;
    }
    else
    {
      ptr = start_replace + 1; // 继续查找下一个 $ 符号
    }
  }
  return cmdline;
}
/*
* eval - Evaluate the command line that the user has just typed in
*
* If the user has requested a built-in command (quit, jobs, bg or fg)
* then execute it immediately. Otherwise, fork a child process and
* run the job in the context of the child. If the job is running in
* the foreground, wait for it to terminate and then return.  Note:
* each child process must have a unique process group ID so that our
* background children don't receive SIGINT (SIGTSTP) from the kernel
```

```
* when we type ctrl-c (ctrl-z) at the keyboard.
*/
void eval(char *cmdline)
{
  pid_t pid;
  sigset_t mask;
  char *argv[MAXARGS];
  for (int k = 0; cmdline[k] != '\0'; k++)
  {
    if (cmdline[k] == '|')
    {
      evalpipe(cmdline);
      return;
    }
  }
  char *substitution_cmd = command_substitution(cmdline);
  // printf("reach");
  volatile int pathflag = 0;
  if (*cmdline == '\n')
    return;
  int bg = parseline(substitution_cmd, argv); // use this fuc to fill the argv
  char *output_file = NULL;                   // 输出重定向的文件名
  int output_fd;                              // 重定向输出的文件描述符
  int i;
  int saved_stdout; // 保存标准输出的文件描述符
  // 保存标准输出的文件描述符
  saved_stdout = dup(STDOUT_FILENO);
  // 检查命令行参数中是否包含输出重定向符号以及重定向的文件名
  for (i = 0; argv[i] != NULL; i++)
  {
    if (strcmp(argv[i], ">") == 0 || strcmp(argv[i], ">>") == 0)
    {
      output_file = argv[i + 1];
      break;
    }
  }

  // 打开输出重定向的文件
  if (output_file != NULL)
  {
    if (strcmp(argv[i], ">") == 0)
      output_fd = open(output_file, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP
    else // ">>"
```

```
    output_fd = open(output_file, O_WRONLY | O_CREAT | O_APPEND, S_IRUSR | S_IWUSR | S_IRGR
  if (output_fd < 0)
  {
    fprintf(stderr, "Failed to open output file %s\n", output_file);
    return;
  }

  // 将标准输出重定向到文件
  if (dup2(output_fd, STDOUT_FILENO) < 0)
  {
    fprintf(stderr, "Failed to redirect standard output\n");
    close(output_fd);
    return;
  }

  // 关闭重定向文件描述符的副本
  close(output_fd);
  for (int j = i; argv[j] != NULL; j++)
  {
    argv[j] = argv[j + 2];
  }
}

if (!builtin_cmd(argv))
{ // forking and execing a child fuction...
  sigemptyset(&mask);
  sigaddset(&mask, SIGCHLD);
  sigprocmask(SIG_BLOCK, &mask, NULL);
  char *executable;
  // 在 PATH 中搜索用户输入的命令对应的可执行文件
  if (access(argv[0], X_OK) != 0)
  {
    pathflag = 1;
    executable = search_executable(argv[0]);
    if (executable == NULL)
    {
      printf("%s: command not found\n", argv[0]);
      return;
    }
  }
  else
  {
    executable = argv[0];
```

```
    }

    if ((pid = fork()) == 0)
    { // in child now
      sigprocmask(SIG_UNBLOCK, &mask, NULL);
      setpgid(0, 0);
      if (execv(executable, argv) < 0)
      {
        printf("Command not found!\n");
        exit(0);
      }
    }
    if (pathflag == 1)
      free(executable);
    addjob(jobs, pid, bg ? BG : FG, cmdline);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    if (!bg)
    {
      waitfg(pid);
    }
    else
    {
      printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
    }
  }
  dup2(saved_stdout, STDOUT_FILENO);
  close(saved_stdout);
  return;
}
```

```c
int builtin_cmd(char **argv)
{
  if (strcmp("quit", argv[0]) == 0)
  {
    exit(0);
  }
  else if (strcmp("jobs", argv[0]) == 0)
  {
    listjobs(jobs);
    return 1;
  }
  else if ((strcmp("fg", argv[0]) == 0) || (strcmp("bg", argv[0]) == 0))
  {
    do_bgfg(argv);
    return 1;
  }
  return 0;
}

/*
* do_bgfg - Execute the builtin bg and fg commands
*/
void do_bgfg(char **argv)
{
  struct job_t *job;
  int jid;
  pid_t pid;
  char *temptr;

  temptr = argv[1];
  // NULL:bg command requires PID or %jobid argument
  if (!temptr)
  {
    printf("%s command requires PID or %%jobid argument\n", argv[0]);
    return;
    //%% means %
  }
  else if (temptr[0] == '%')
  {
    jid = atoi(&temptr[1]);
    job = getjobjid(jobs, jid);
    if (!job)
    {
```

```
      printf("%%%d: No such job\n", jid);
      return;
    }
  }
  else if (isdigit(temptr[0]))
  {
    pid = atoi(temptr);
    job = getjobpid(jobs, pid);
    if (!job)
    {
      printf("(%d): No such process\n", pid);
      return;
    }
  }
  else
  {
    printf("%s: argument must be a PID or %%jobid\n", argv[0]);
    return;
  }
  if (kill(-(job->pid), SIGCONT) < 0)
  {
    unix_error("kill error");
  }

  if (strcmp("fg", argv[0]) == 0)
  {
    job->state = FG;
    waitfg(job->pid);
  }
  else
  {
    job->state = BG;
    printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
  }
  return;
}

/*
* waitfg - Block until process pid is no longer the foreground process
*/
void waitfg(pid_t pid)
{
  struct job_t *job;
```

```
  job = getjobpid(jobs, pid);
  if (!job)
  {
    return;
  }
  // no judgement 'cause the dobgfg has already confirm the validity of job and pid.
  while (pid == fgpid(jobs))
  {
    sleep(1);
  }
  return;
}

/*****************
 * Signal handlers
 *****************/

/*
* sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
*     a child job terminates (becomes a zombie), or stops because it
*     received a SIGSTOP or SIGTSTP signal. The handler reaps all
*     available zombie children, but doesn't wait for any other
*     currently running children to terminate.
*/
void sigchld_handler(int sig)
{
  int status;
  pid_t pid;
  struct job_t *job;
  while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)
  {
    job = getjobpid(jobs, pid);
    if (WIFEXITED(status))
    {
      deletejob(jobs, pid);
    }
    else if (WIFSIGNALED(status))
    {
      printf("Job [%d] (%d) terminated by signal %d\n", job->jid, job->pid, WTERMSIG(status));
      deletejob(jobs, pid);
    }
    else if (WIFSTOPPED(status))
    {
```

```
      job->state = ST;
      printf("Job [%d] (%d) stopped by signal %d\n", job->jid, job->pid, WSTOPSIG(status));
    }
  }
  return;
}

/*
* sigint_handler - The kernel sends a SIGINT to the shell whenver the
*    user types ctrl-c at the keyboard.  Catch it and send it along
*    to the foreground job.
*/
void sigint_handler(int sig)
{
  pid_t pid = fgpid(jobs);
  if (pid) // not 0
  {
    if (kill(-pid, SIGINT) < 0)
    {
      unix_error("kill error");
    }
  }
  return;
}

/*
* sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
*     the user types ctrl-z at the keyboard. Catch it and suspend the
*     foreground job by sending it a SIGTSTP.
*/
void sigtstp_handler(int sig)
{
  pid_t pid = fgpid(jobs);
  if (pid)
  {
    if (kill(-pid, SIGTSTP) < 0)
    {
      unix_error("kill error");
    }
  }
  return;
}
```