CSE 344 HW-2 Report
200104004095

In this homework we would implement a terminal emulator. That handles | , < and > operators. Let’s review the what I done:

The basic idea here is that: I take commands in a while loop and then split the command with pipe delimeter and run them in multiple processes and every process’s output is connected to the next processes input with pipe. There is a function named with run_command() that takes input_fd, output_fd and splitted command. There is another function named with redirect_io(). It takes the redirection part of the command (like: < input.txt > output.txt) and redirects the output and input current process. For logs I hold a log file for every command as told in pdf. Lets review the code in detail:

For shell I set up a infinity loop as shown in the lectures. It prompts the user and takes input with fgets() function. There is limit of maximum command which defined by MAX_CMD_LEN 1024.

```
// Shell loop
while (1)
{
  proc_count = 0;

  // Prompt message
  printf("ъ ");

  // Flush the prompt to see the message correctly
  fflush(stdout);


  // Get command from user
  if (fgets(input_str, MAX_CMD_LEN, stdin) == NULL)
    break;
```

After getting the string from prompt it checks if it’s exit command (:q) or not. If the command is ‘:q’ then exits. And also it checks if the command is empty (like full of with space or just enter)

```
// Quit command check
if( strcmp(input_str, ":q\n") == 0 )
    exit(EXIT_SUCCESS);

// Handle free command
if (is_free(input_str))
  continue;
```

is_free(cmd) takes the command and if there is a char outer ‘\n’ and ‘ ’ then returns 0

```
// Check if the command is free (space or nothing) (like : "   \n", "\n" etc.)
int is_free(char* cmd){

    if(cmd[0] == '\n')
      return 1;

    int cmd_len = strlen(cmd), ind;
    for ( ind=0 ; ind < cmd_len ; ind++)
      if( ! isspace(cmd[ind]) )
          return 0;
    return 1;

}
```

Then I open a log file with current timestamp in log directory.

```
// Open a log file with timestamp
int log_fd = open_log_file();
```

```c
// Opens a log file in log directory with current timestamp
int open_log_file(){

    // Check if directory exists
    if (access(LOG_DIRECTORY, F_OK) != 0) {
        // Create directory if it does not exist
        if (mkdir(LOG_DIRECTORY, 0777) == -1) {
            perror("mkdir");
            exit(EXIT_FAILURE);
        }
        // printf("Directory created successfully.\n");
    } else {
        // printf("Directory already exists.\n");
    }

    // Get current time
    char file_name[100];
    time_t t = time(NULL);

    // Get timestamp string
    char *time_stamp  = ctime(&t);
    time_stamp[strlen(time_stamp)-1] = '\0';

    // Format the name of the log file
    sprintf(file_name,"%s%s.log", LOG_DIRECTORY, time_stamp);

    // Open the file
    int fd = open(file_name, O_RDWR | O_CREAT ,0777);
    return fd;

}
```

strtok_r is used to split commands with delimeter ‘|’. At first I used strtok but it was not thread safe so I used strtok_r which is thread safe. After splitting the pipes in the command then I call run_command function in a while loop that creates child process and connect the input and output.

```c
// It process commands after every pipe
while (command != NULL )
{
  proc_count++;
  pipe( pipes[proc_count] );

  // Copy of the command to send to child
  strcpy(command_copy, command);

  pid_t child_pid;

  // Split the command with pipe again
  command = strtok_r(NULL, pip, &context_pip);

  // If the command is last command then we make it's output STDOUT
  // Runs the command with a new child process and gives it's input and outputs
  if(command == NULL)
    child_pid = run_command(command_copy, pipes[proc_count-1][0], STDOUT_FILENO);

  // If it's not las command then we connect it's output to the next child's input with a pipe
  else
    child_pid = run_command(command_copy, pipes[proc_count-1][0], pipes[proc_count][1]);


  // Write log entry for every child processes
  char log_entry[MAX_CMD_LEN+10];
  sprintf(log_entry, "%d : %s \n", child_pid, command_copy);
  write(log_fd, log_entry, sizeof(log_entry));

  // Close the write end of pipe of parent.
  close(pipes[proc_count][1]);

} // End of one line command processing
```

Lets review run_command() function :

```c
// Takes the splitted command and takes input and output fd
// Process the command for < and >
pid_t run_command(char *command, int input_fd, int output_fd)
{
  int status;
  pid_t childPid;
  char cmd_copy[MAX_CMD_LEN];

  switch (childPid = fork())
  {
    case -1:          /* process creation error */
      return -1;

    case 0:       /* Child */
      printf("PID:%8d PPID:%8d\n",getpid(), getppid());
      fflush(stdout);

      // Change the input and output to the given fd's
      dup2(input_fd, STDIN_FILENO);
      dup2(output_fd, STDOUT_FILENO);


      // !!!! close other end of pipe

      // Divide the command 2 part (basic command and redirection part)
      char *cmd_part1, *cmd_part2, *context2;

      strcpy(cmd_copy,command);

      // Extract command string
      cmd_part1 = strtok_r(command, "<>", &context2);
      printf("%s",cmd_part1);

      // Extract redirection string that starts with < or >
      cmd_part2 = &cmd_copy[ strlen(cmd_part1) ];

      // Redirect IO according to the given redirection string. It parses and redirects the input and output.
      redirect_io(cmd_part2);

      // Run the basic command (without pipe and redirection operators)
      execl("/bin/sh", "sh", "-c", cmd_part1, (char *)NULL);

      // Exec fail
      _exit(127);
```

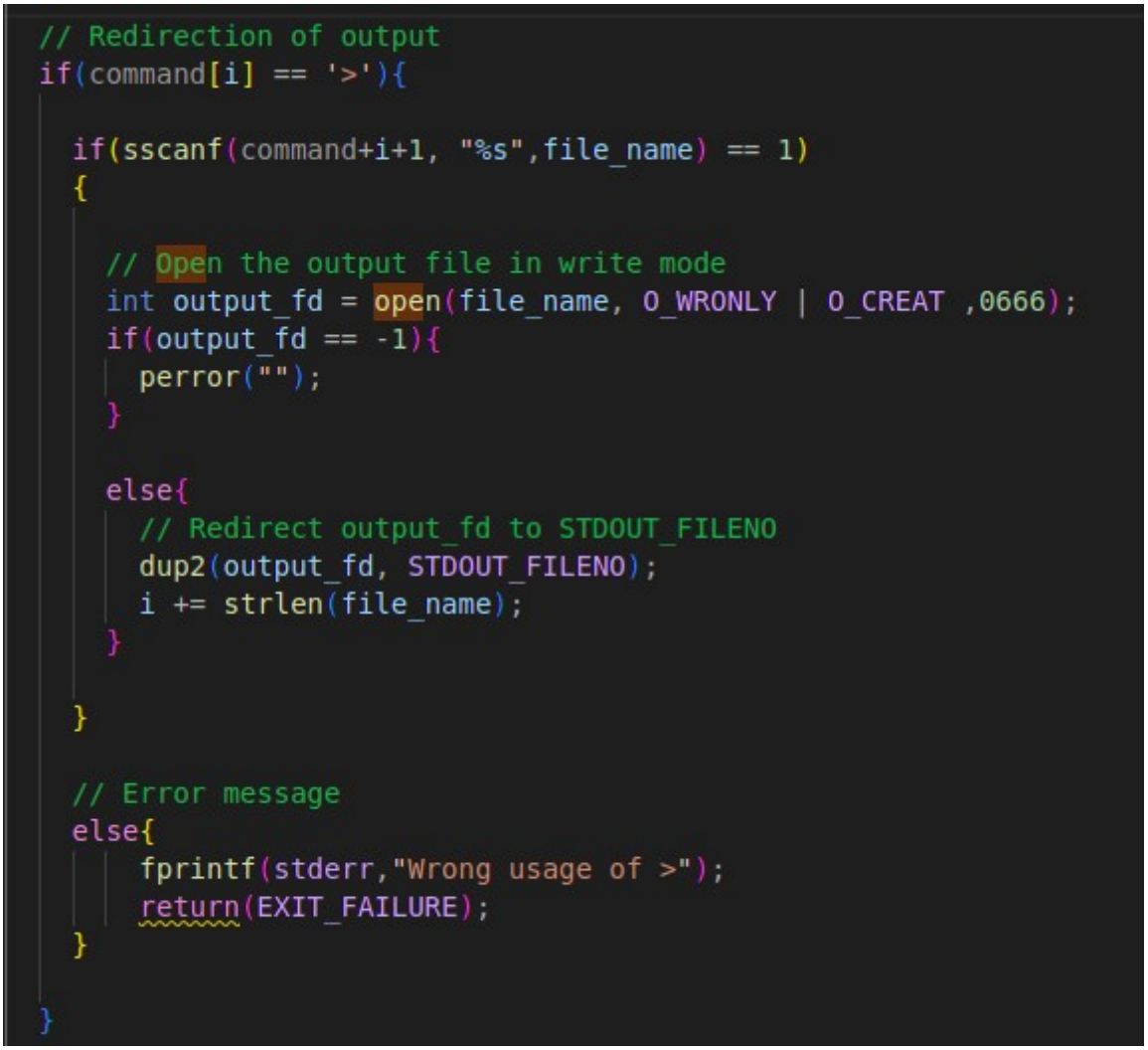
It takes 3 parameters : input_fd, output_fd and command to process. I gave pipes from previous child as input_fd and and a pipe to give next child's input. In the tun_command() function I use dup2 to redirect output and inputs. Then I divide the command into 2 part with '<' or '>' character. One is basic command (cmd_part1) and the other is redirection command part (cmd_part2) then I give the redirection part to a function named redirect_io() which takes redirefction string and redirects the input (and, or) output to file.

redirect_io() 1: Traverses the command char by char and if finds a redirect operator (<,>) then takes file name and redirects it.

```c
// Take redirection part of command (like: < input.txt > output.txt) and change input and output
void redirect_io(char* command){

  char file_name[MAX_FILE_NAME_LEN];

  int len = strlen(command);

  for (int i = 0; i < len; i++)
  {
    // Redirection of input
    if(command[i] == '<'){

      if(sscanf(command+i+1, "%s",file_name) == 1)
      {

        // Open the input file in read mode
        int input_fd = open(file_name, O_RDONLY);
        if(input_fd == -1){
          perror("");
          exit(EXIT_FAILURE);
        }

        else{
          // Redirect input_fd to STDIN_FILENO
          dup2(input_fd, STDIN_FILENO);
          i += strlen(file_name);
        }

      }

      // Error message
      else{
          fprintf(stderr,"Wrong usage of <");
          exit(EXIT_FAILURE);
      }

    }
```

(continued)

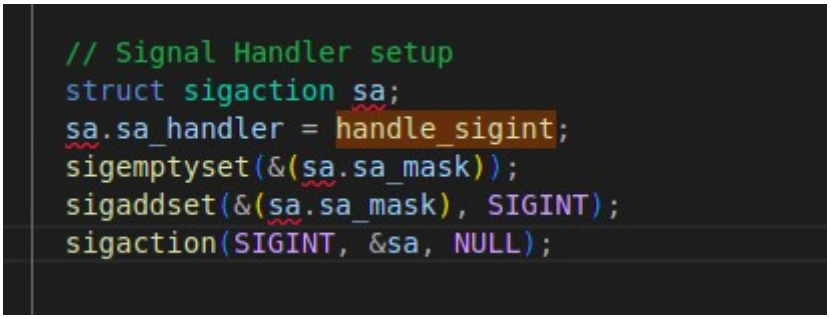
redirect_io() 2: Does the same thing for > operator



```
// Redirection of output
if(command[i] == '>'){

  if(sscanf(command+i+1, "%s",file_name) == 1)
  {

    // Open the output file in write mode
    int output_fd = open(file_name, O_WRONLY | O_CREAT ,0666);
    if(output_fd == -1){
      perror("");
    }

    else{
      // Redirect output_fd to STDOUT_FILENO
      dup2(output_fd, STDOUT_FILENO);
      i += strlen(file_name);
    }

  }

  // Error message
  else{
      fprintf(stderr,"Wrong usage of >");
      return(EXIT_FAILURE);
  }

}
```

In the end of loop I close end of pipes of parent. To make synchronous piping. And I log the child and command with process id. After command process loop, I close log file and wait for processes to end and print them termination signal status.

Signal handling is done. When user press ctrl+c it sends SIGINT to all children.



```c
// Signal Handler setup
struct sigaction sa;
sa.sa_handler = handle_sigint;
sigemptyset(&(sa.sa_mask));
sigaddset(&(sa.sa_mask), SIGINT);
sigaction(SIGINT, &sa, NULL);
```