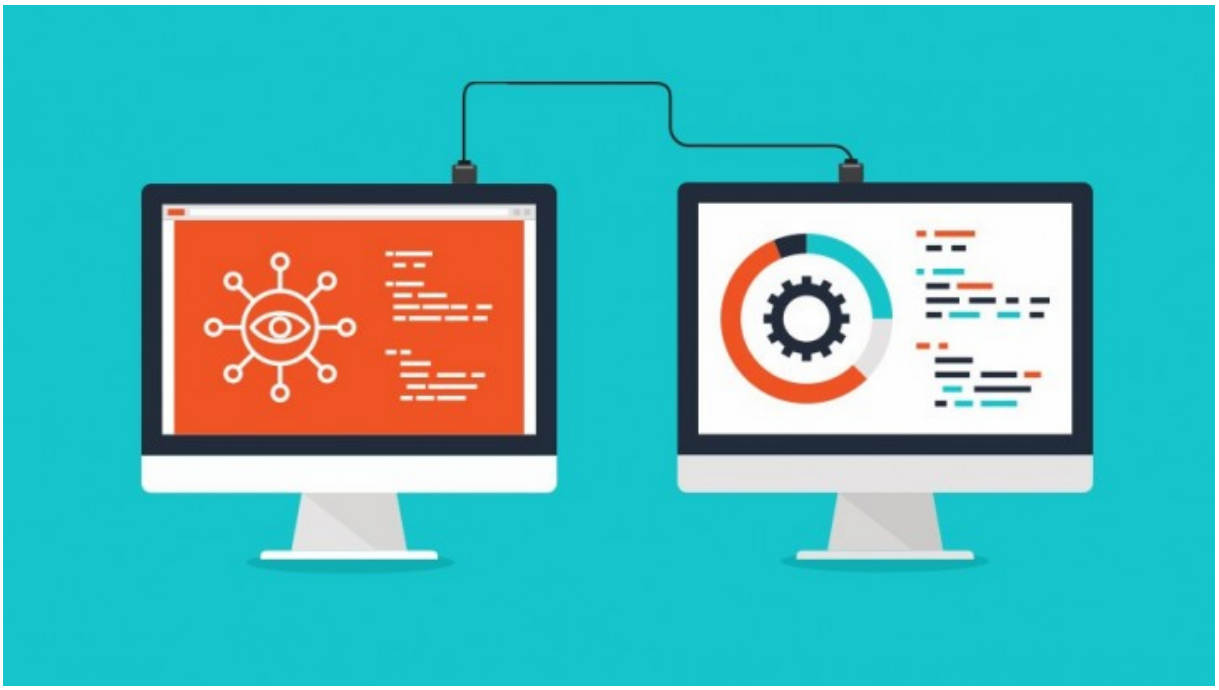
Lab 1

Simple Shell (Multi-Processing)



Name : Pancee Wahid
ID: 18010467

Assumptions

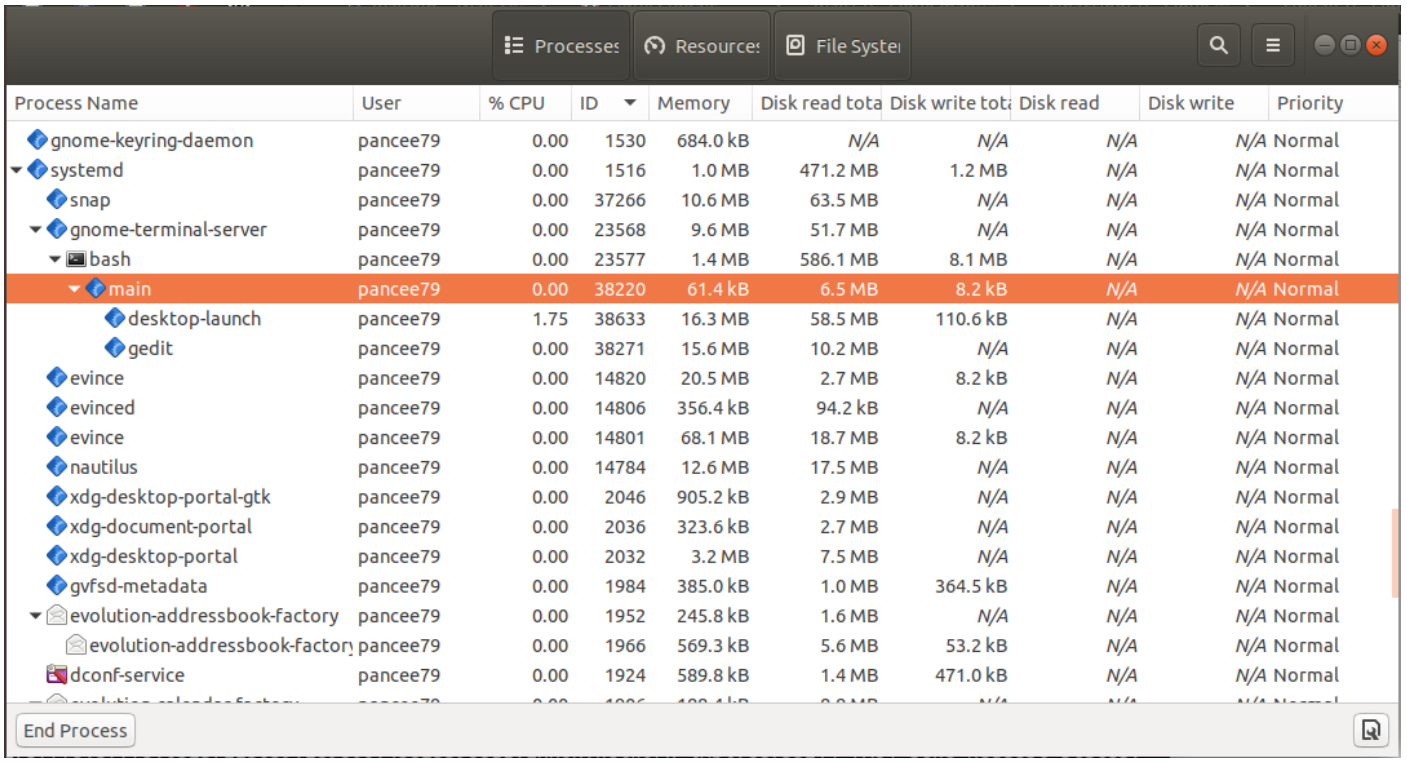
1. All inputs don't contain spaces except between command and arguments and in between arguments.
2. The maximum number of variables can be saved using export is 20
3. Command maximum length = 400 characters
4. Maximum number of words separated by spaces in each command = 10
5. Each word has maximum length of 100 characters

Running gnome-calculator (process name = desktop-launch) and text editor (process name = gedit) using the simple shell by entering the following commands

gnome-calculator

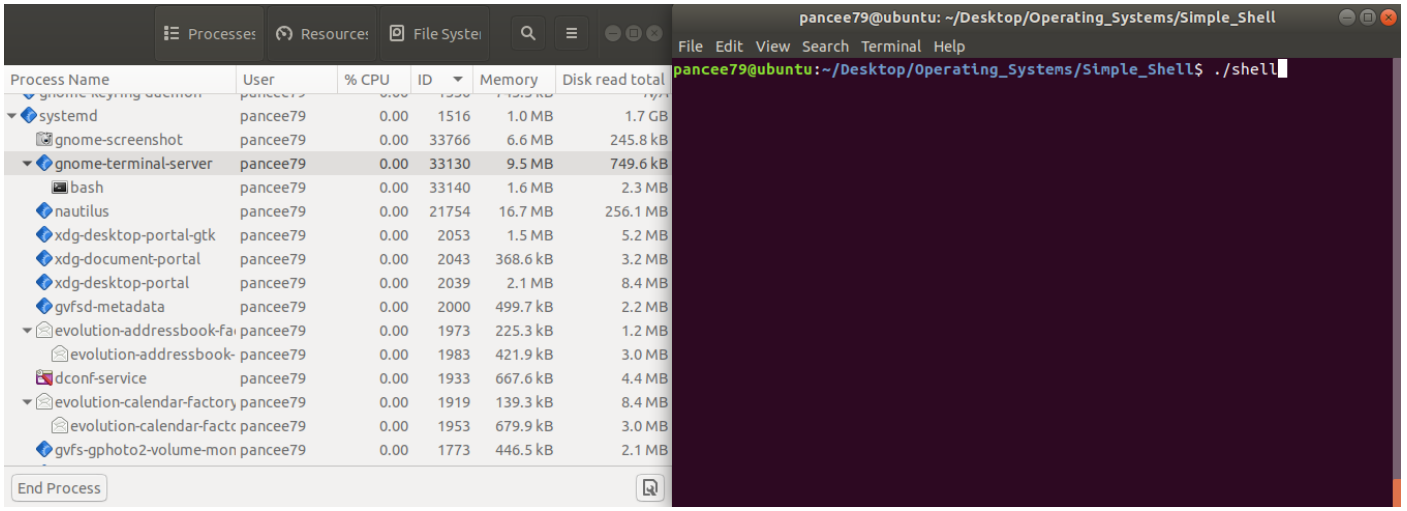
gedit

Both are running as child processes of the main (simple shell) process.

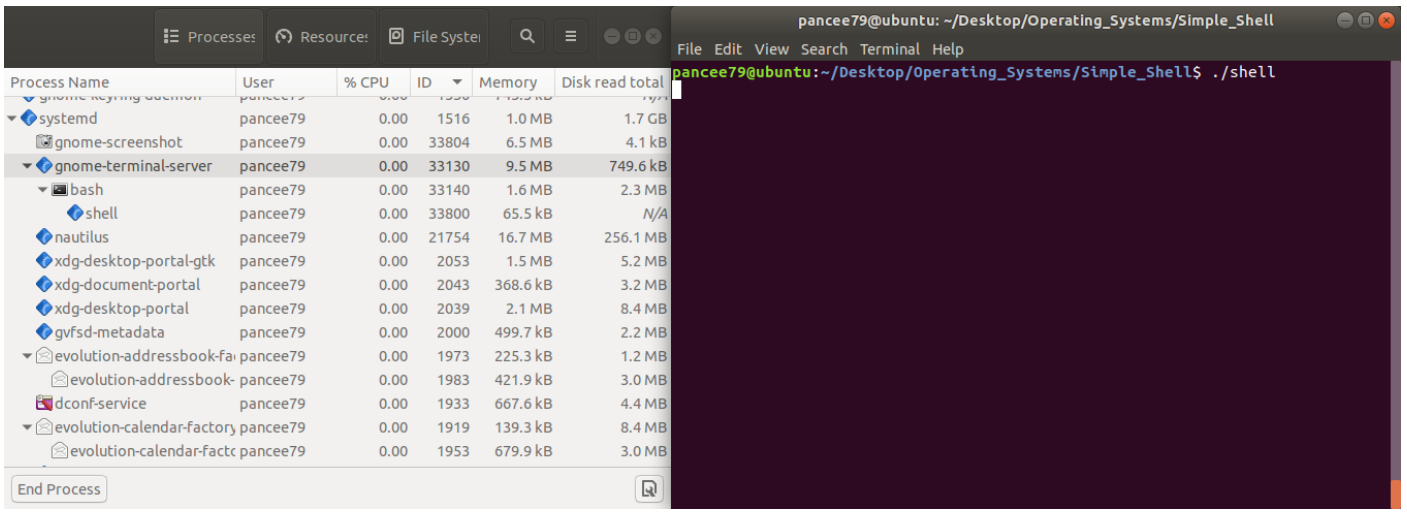


Sample Run

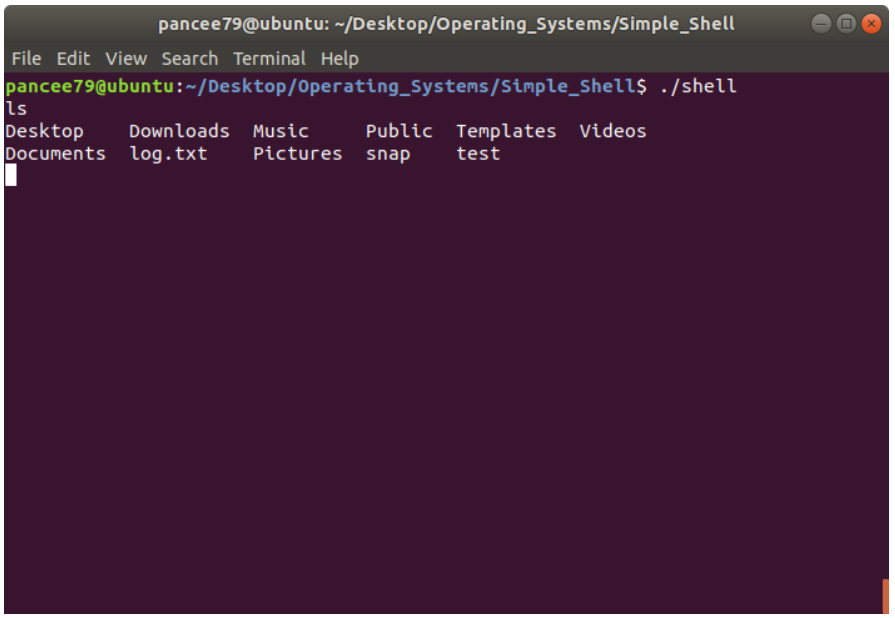
1. Before running the simple shell



2. After running simple shell

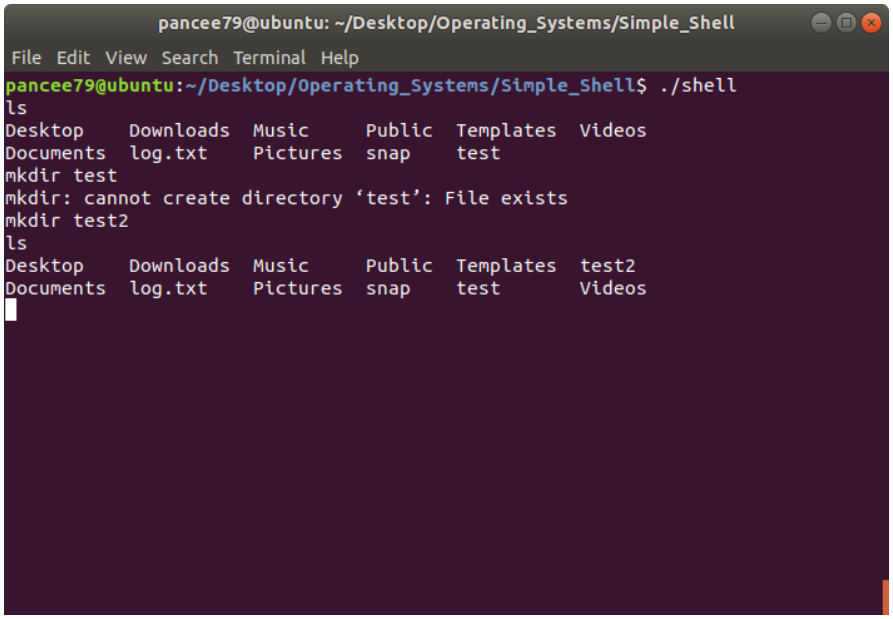


3. ls



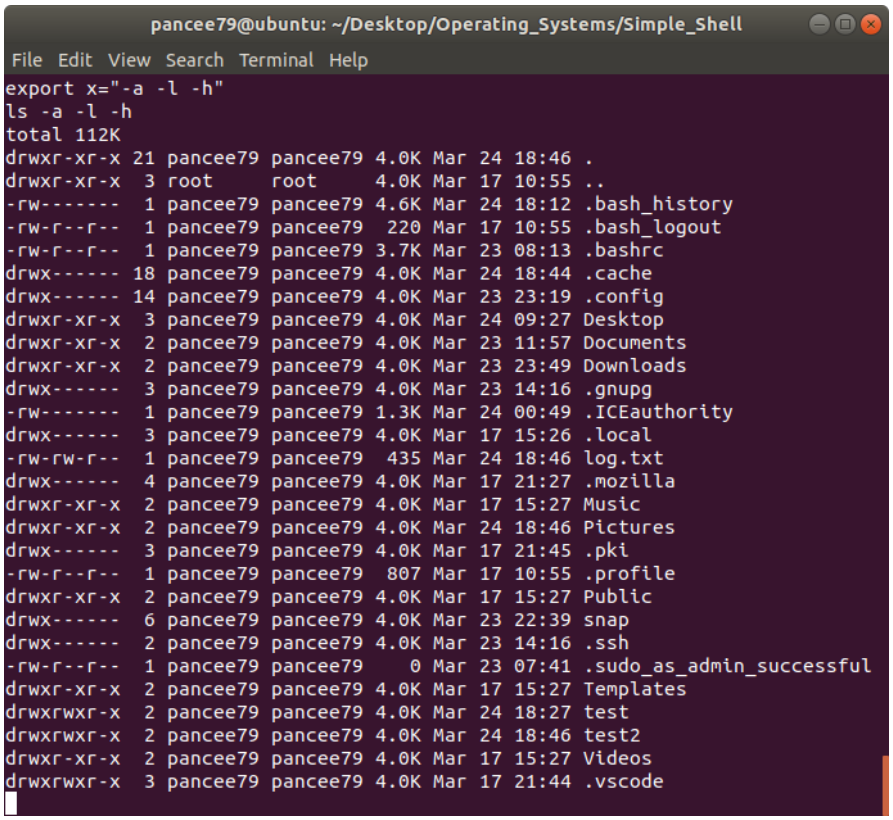
4. mkdir test2

ls

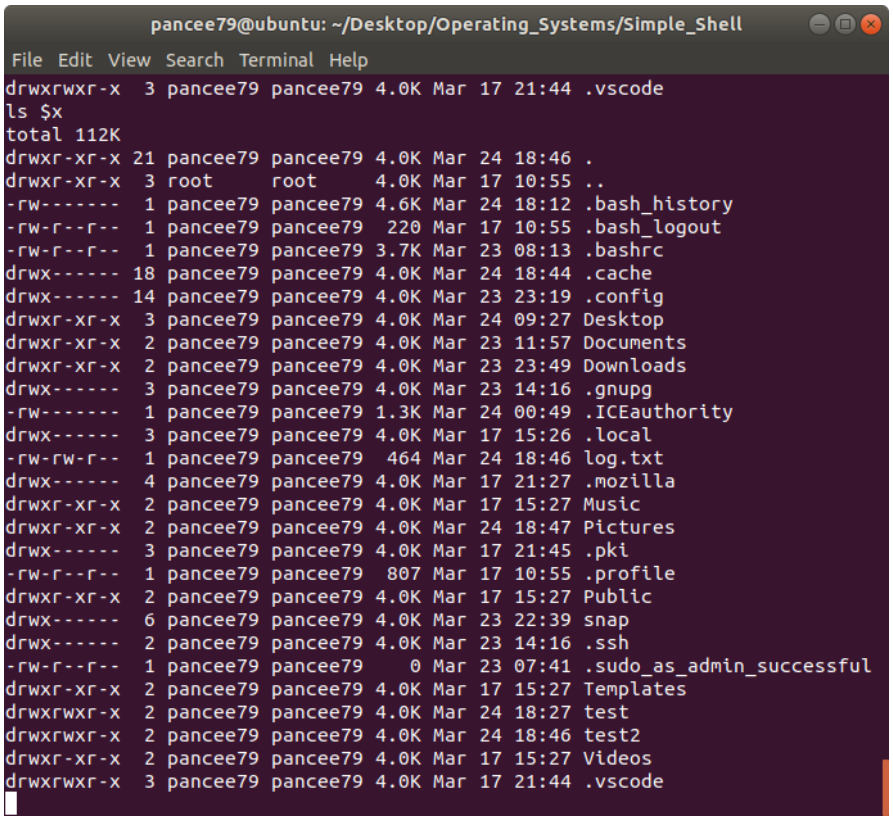


5. export x=”-a -l -h”

ls -a -l -h



6. ls $x

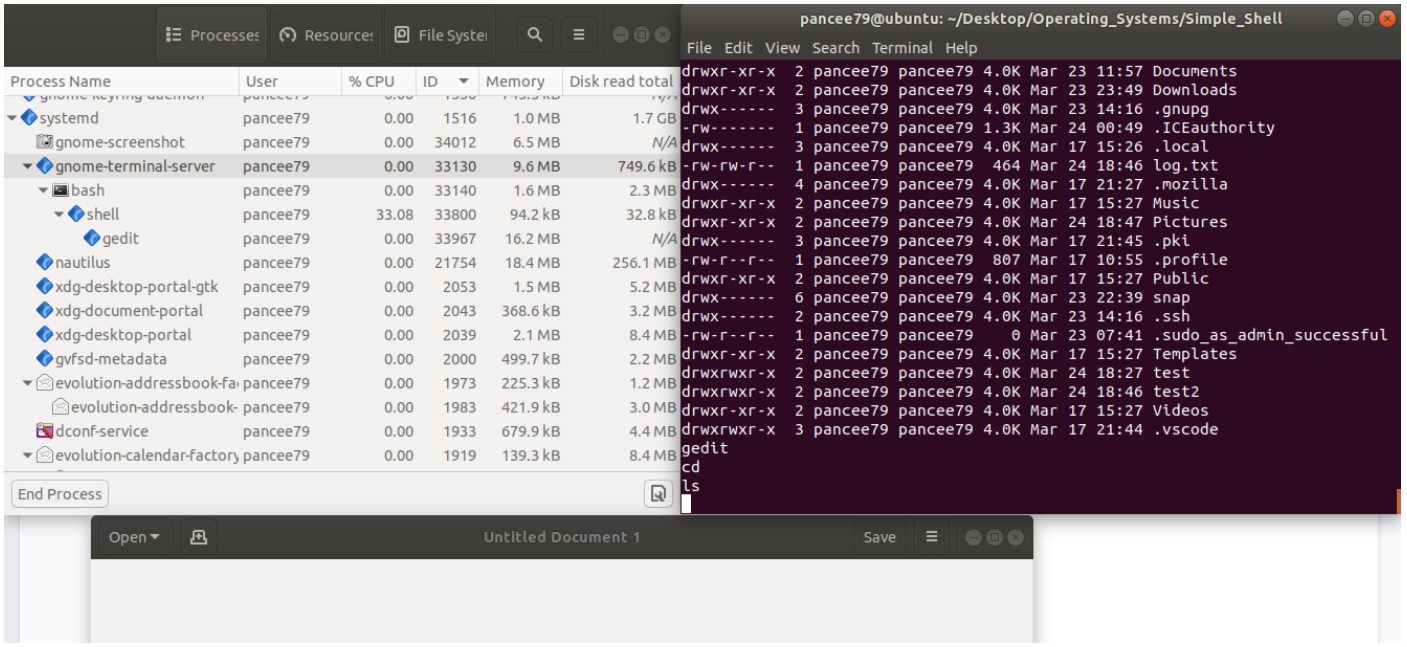


7. gedit → (foreground)

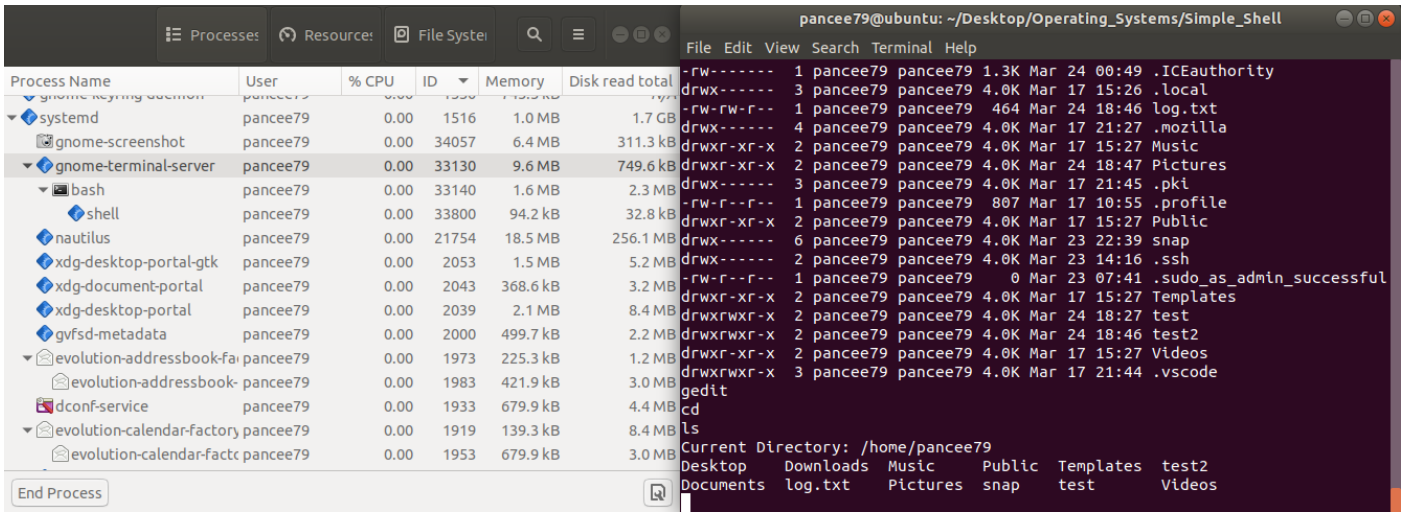
Showing shell is stuck trying:

cd

ls

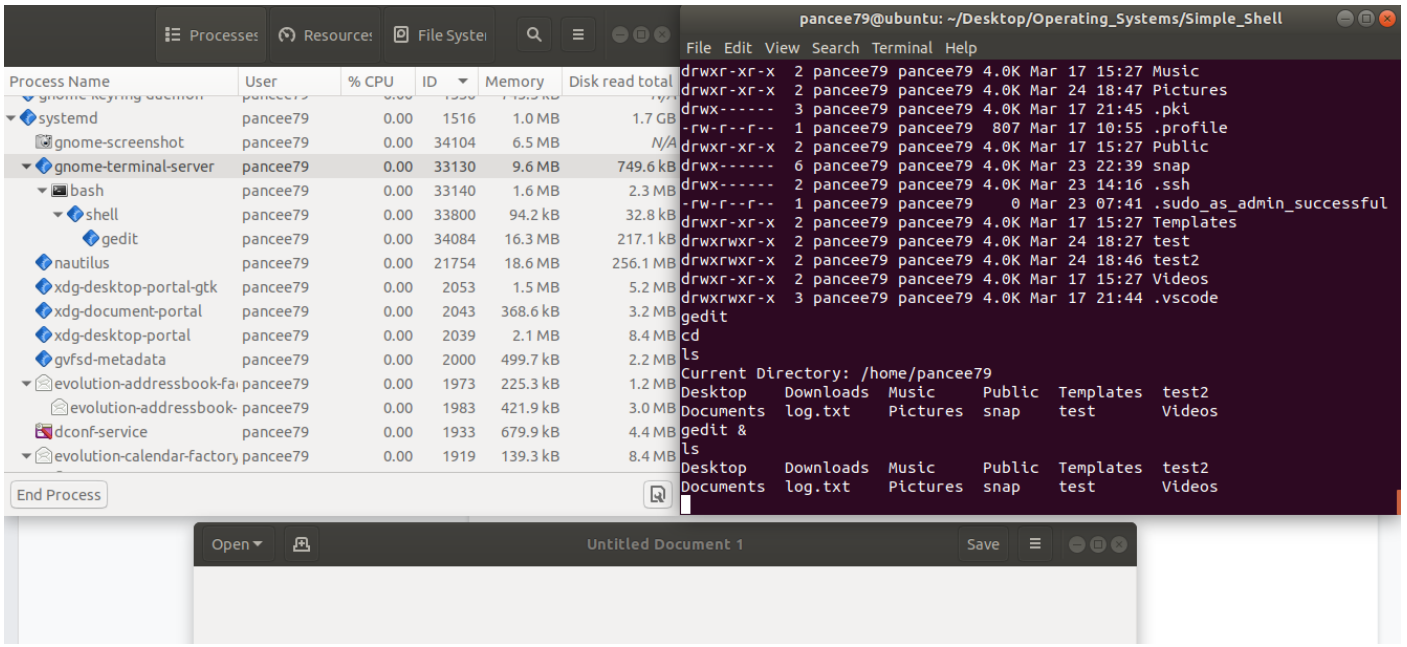


8. After closing the text editor, commands are executed.

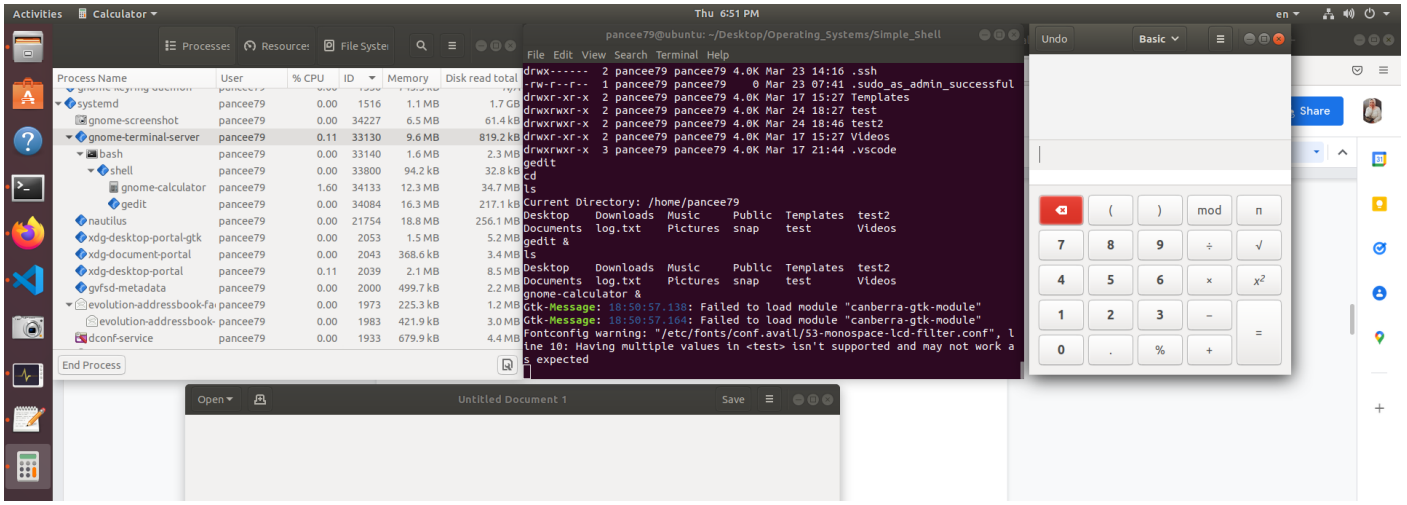


9. gedit & → (background)

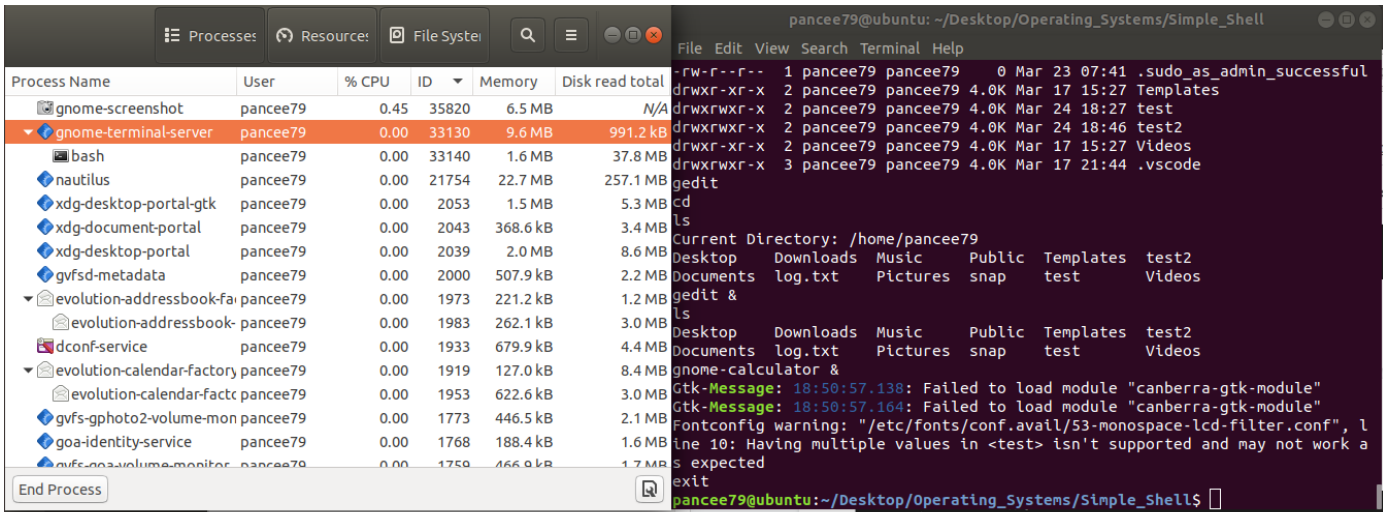
Showing that shell is not stuck.



10. gnome-calculator &



11. exit


Processes
Resources
File System
Process Name User % CPU ID Memory Disk read total
gnome-screenshot pancee79 0.45 35820 6.5 MB N/A
gnome-terminal-server pancee79 0.00 33130 9.6 MB 991.2 kB
bash pancee79 0.00 33140 1.6 MB 37.8 MB
nautilus pancee79 0.00 21754 22.7 MB 257.1 MB
xdg-desktop-portal-gtk pancee79 0.00 2053 1.5 MB 5.3 MB
xdg-document-portal pancee79 0.00 2043 368.6 kB 3.4 MB
xdg-desktop-portal pancee79 0.00 2039 2.0 MB 8.6 MB
gvfsd-metadata pancee79 0.00 2000 507.9 kB 2.2 MB
evolution-addressbook-fa pancee79 0.00 1973 221.2 kB 1.2 MB
evolution-addressbook- pancee79 0.00 1983 262.1 kB 3.0 MB
dconf-service pancee79 0.00 1933 679.9 kB 4.4 MB
evolution-calendar-factory pancee79 0.00 1919 127.0 kB 8.4 MB
evolution-calendar-factc pancee79 0.00 1953 622.6 kB 3.0 MB
gvfs-gphoto2-volume-mon pancee79 0.00 1773 446.5 kB 2.1 MB
goa-identity-service pancee79 0.00 1768 188.4 kB 1.6 MB
End Process
pancee79@ubuntu: ~/Desktop/Operating_Systems/Simple_Shell
File Edit View Search Terminal Help
-rw-r--r-- 1 pancee79 pancee79 0 Mar 23 07:41 .sudo_as_admin_successful
drwxr-xr-x 2 pancee79 pancee79 4.0K Mar 17 15:27 Templates
drwxrwxr-x 2 pancee79 pancee79 4.0K Mar 24 18:27 test
drwxrwxr-x 2 pancee79 pancee79 4.0K Mar 24 18:46 test2
drwxr-xr-x 2 pancee79 pancee79 4.0K Mar 17 15:27 Videos
drwxrwxr-x 3 pancee79 pancee79 4.0K Mar 17 21:44 .vscode
gedit
cd
ls
Current Directory: /home/pancee79
Desktop Downloads Music Public Templates test2
Documents log.txt Pictures snap test Videos
gedit &
ls
Desktop Downloads Music Public Templates test2
Documents log.txt Pictures snap test Videos
gnome-calculator &
Gtk-Message: 18:50:57.138: Failed to load module "canberra-gtk-module"
Gtk-Message: 18:50:57.164: Failed to load module "canberra-gtk-module"
Fontconfig warning: "/etc/fonts/conf.avail/53-monospace-lcd-filter.conf", l
ine 10: Having multiple values in <test> isn't supported and may not work a
s expected
exit
pancee79@ubuntu:~/Desktop/Operating_Systems/Simple_Shell$

Code Overview

```c
C main.c > ...
  1  #include <stdio.h>
  2  #include <stdlib.h>
  3  #include <unistd.h>
  4  #include <sys/wait.h>
  5  #include <signal.h>
  6  #include <sys/resource.h>
  7  #include <string.h>
  8
  9  #define MAX_LINE_LENGTH 400
 10  #define MAX_WORD_LENGTH 100
 11  #define MAX_WORDS_NO 10
 12  #define MAX_VARIABLES_NO 20
 13
 14  // for export
 15  int available_space = MAX_VARIABLES_NO;
 16  char* var_name[MAX_VARIABLES_NO] = {};
 17  char* var_value[MAX_VARIABLES_NO] = {};
 18
 19  // for running in backgroud or foreground
 20  int background = 0;
 21
 22  /* Functions Declarations */
 23  void on_child_exit();
 24  void setup_environment();
 25  void shell();
 26  int parse_execute(char cmd[], char* cmd_args[]);
 27  void change_directory(char arg[]);
 28  void export(char arg[]);
 29  void echo(char arg[]);
 30  void execute_command(char cmd_type[], char* cmd_args[]);
 31  int evaluate_args(char arg[], char evaluated[]);
 32
 33  /* parent_main function */
 34 > int main(){ ⋯
 52
 53  /* Interrupt handler to remove zombie process */
 54 > void on_child_exit(){ ⋯
 82
```

```c
 83 > void shell(){ ⋯
 96
 97 > int parse_execute(char cmd[], char* cmd_args[]){ ⋯
140
141 > void change_directory(char arg[]) ⋯
169
170 > void export(char arg[]){ ⋯
199
200 > void echo(char arg[]){ ⋯
213
214 > int evaluate_args(char arg[], char evaluated[]){ // returns 0 in case of
257
258 > void execute_command(char cmd_type[], char* cmd_args[]){ ⋯
```

Functions

Main Function & on_child_exit() interrupt handler

Allocates memory for arrays storing exported variables and their values.

Registers child signal, when SIGCHLD is received, the interrupt handler on_child_exit() function is called.

Changes directory to home directory.

Calls shell() function.

```
33  /* parent_main function */
34  int main(){
35      /* initialization */
36      for (int i = 0; i < MAX_VARIABLES_NO; i++) {
37          var_name[i] = (char*) malloc(MAX_WORD_LENGTH*sizeof(char));
38          var_value[i] = (char*) malloc(MAX_WORD_LENGTH*sizeof(char));
39      }
40
41      /* register_child_signal(on_child_exit()) */
42      signal(SIGCHLD, on_child_exit);
43
44      /* environment setup */
45      if (chdir(getenv("HOME")) != 0)
46              printf("ERROR: Could't change directory to home directory\n"
47
48      /* shell() */
49      shell();
50
51      return 0;
52  }
```

```
54  /* Interrupt handler to remove zombie process */
55  void on_child_exit(){
56      /* reap zombie process */
57      int wstat;
58      pid_t pid;
59
60      while (1) {
61          pid = wait3(&wstat, WNOHANG, (struct rusage *)NULL);
62          if (pid == 0 || pid == -1)
63              break;
64      }
65      // printf("Child terminated");
66      /* write to log file  */
67      // open the file for appending (or creation if it doesn't exist)
68      FILE* log_file;
69      log_file = fopen("log.txt", "a");
70
71      // print error msg in case of error while creating the file
72      if (log_file == NULL){
73          printf("Error in creating log file!");
74      }
75
76      // write the termination msg to the log file and close it
77      // char msg[] = "Child process was terminated\n";
78      // fwrite(msg , 1 , sizeof(msg) , log_file);
79      fprintf(log_file, "Child process was terminated\n");
80      fclose(log_file); // close the log file
81      return;
82  }
```

shell()

```
84  void shell(){
85      int is_exit = 0;
86      do{
87          char cmd[MAX_LINE_LENGTH];
88          char* cmd_args[MAX_WORDS_NO];
89          for (int i = 0; i < MAX_WORDS_NO; i++) { // try to delete
90              cmd_args[i] = (char*) malloc(MAX_WORD_LENGTH*sizeof(char));
91          }
92          fgets(cmd, MAX_LINE_LENGTH, stdin);
93          is_exit = parse_execute(cmd, cmd_args);
94      } while (!is_exit);
95      exit(0);
96  }
```

Takes an input line, parses it then if it's exit command, the parse_execute() function will return 1 which will terminate the loop. Then exit() is called to terminate the main process.

parse_execute()

```
98  int parse_execute(char cmd[], char* cmd_args[]){
99      char* context = NULL;
100     int i = 0;
101     for (i = 0; cmd[i] != '\n'; i++) {;}
102     cmd[i] = '\0';
103
104     // get command type
105     char *cmd_type = strtok_r(cmd, " ", &context);
106
107     if (cmd_type == NULL || strcmp(cmd_type, "") == 0){ // no command
108         printf("No Command is entered!\n");
109         return 0;
110     }
111     else if (strcmp(cmd_type, "exit") == 0){ // exit command
112         return 1;
113     }
114
115     // built-in commands
116     if (strcmp(cmd_type, "cd") == 0 ){
117         change_directory(context);
118     }
119     else if (strcmp(cmd_type, "echo") == 0){
120         echo(context);
121     }
122     else if(strcmp(cmd_type, "export") == 0 ){
123         export(context);
124     }
125     else
126     {
127     //non-built-in commands
128         cmd_args[0] = cmd_type;
129         char* context_args = NULL;
130         char* token = strtok_r(context, " ", &context_args);
131         int k = 1;
132         while (token != NULL){
133             cmd_args[k++] = token;
134             token = strtok_r(NULL, " ", &context_args);
135         }
136         cmd_args[k] = NULL;
137         execute_command(cmd_type, cmd_args);
138     }
139     return 0;
140 }
```

Parses the command then checks the command type. If it's cd, echo or export, calls the corresponding function. If it's an empty line, shows an appropriate message. If it's exit, returns 1. Otherwise, passes the command and the arguments to execvp() to execute it.

change_directory()

```
142  void change_directory(char arg[])
143  {
144      char s[MAX_WORD_LENGTH];
145      /* cd(Current_Working_Directory) */
146      if (arg == NULL || strcmp(arg, "") == 0 || strcmp(arg, "~") == 0){ // cd
147          if (chdir(getenv("HOME")) != 0)
148              printf("ERROR: Could't change directory to home directory\n");
149          printf("Current Directory: %s\n", getcwd(s, MAX_WORD_LENGTH));
150      }
151      else if (strcmp(arg, "..") == 0){ // cd to parent dir
152          if (chdir("..") != 0)
153              printf("ERROR: Could't change directory to parent directory\n");
154          printf("Current Directory: %s\n", getcwd(s, MAX_WORD_LENGTH));
155      }
156      else{  // relative or absolute path
157          char dir[250];
158          char edited_arg[MAX_WORD_LENGTH+1] = "/";
159          if (arg[0] != '/')
160              strcpy(edited_arg, strcat(edited_arg, arg));
161          else
162              strcpy(edited_arg, arg);
163          strcpy(dir, strcat(getcwd(s,MAX_WORD_LENGTH), edited_arg));
164          if (chdir(dir) != 0)
165              printf("ERROR: Could't change directory to %s\n", arg);
166          printf("Current Directory: %s\n", getcwd(s, MAX_WORD_LENGTH));
167      }
168      return;
169  }
```

It's called when the command 'cd' is entered. If there's no argument or "~", it changes the current directory to the home directory. If argument is "..", it changes the current directory to the parent directory. Otherwise, it changes the current directory to the specified path.

echo()

```
201  void echo(char arg[]){
202     char to_print[MAX_LINE_LENGTH];
203     memset(to_print, '\0', MAX_LINE_LENGTH*sizeof(char));
204
205     int stat = evaluate_args(arg, to_print);
206     if (stat == 1)
207        printf("%s\n", to_print);
208     else if (stat == 0)
209        printf("Undefined variable!\n");
210     else // -1
211        printf("No arguments passed!");
212     return;
213  }
214
```

Evaluates arguments and replaces variables with their values then prints the result.

export()

```
171  void export(char arg[]){
172      char* context = NULL;
173      char* context_2 = NULL;
174      char* context_3 = NULL;
175
176      char* token = strtok_r(arg, "=", &context); // context has what's after th
177      // remove spaces before the equal if exist
178      token = strtok_r(token, " ", &context_2); // token has the LHS without lea
179      int index = MAX_VARIABLES_NO - available_space;
180      if (token != NULL)
181      {
182          strcpy(var_name[index], token);
183          // if (context[0] == ' ') {  // remove spaces after the equal if exist
184          //     token = strtok_r(context, " ", &context_3);
185          //     token = strtok_r(token, "\"", &context_3); // remove double quote
186          // } else {
187          token = strtok_r(context, "\"", &context_3);
188          // }
189          if (token == NULL) { // no value entered
190              available_space++;
191              printf("Invalid argument!\n");
192
193          } else {
194              strcpy(var_value[index], token);
195              available_space--;
196          }
197      }
198      return;
199  }
```

Separates the variables and their values and stores them to be used in the same session later.

evaluate_args()

```
215  int evaluate_args(char arg[], char evaluated[]){ // returns 0 in case of unde
216      int found = 0;
217      char* token = arg;
218
219      if (arg == NULL){
220          return -1;
221      }
222      else if (arg[0] == '\"'){ // double quoted
223          char* context = NULL;
224          token = strtok_r(arg, "\"", &context);
225      }
226
227      for (int i = 0; token[i] != '\0'; i++){
228          if (token[i] == '$'){
229              i++;
230              char v[MAX_WORD_LENGTH];
231              memset(v, '\0', MAX_WORD_LENGTH*sizeof(char));
232
233              int j = 0;
234              while (token[i] != '\0' && token[i] != ' ') {
235                  v[j++] = token[i++];
236              }
237              i--; // as i will be incremented again by the main loop
238
239              for (int k = 0; k < MAX_VARIABLES_NO - available_space; k++){
240                  if (strcmp(var_name[k], v) == 0){ // found the variable
241                      strcat(evaluated, var_value[k]);
242                      found = 1;
243                      break;
244                  }
245              }
246              if (found)
247                  found = 0;
248              else
249                  return 0;
250          }
251          else
252              strncat(evaluated, &token[i], 1);
253      }
254      return 1;
255  }
```

Evaluates the arguments by checking the entry after the command. Splits them into an array of arguments.

Replaces the used variables by their values.

Returns 0 in case of using a variable without defining it.

Returns 1 in case of successfully evaluating the arguments.

Returns -1 in case of passing null argument.

execute_command()

```
257  void execute_command(char cmd_type[], char* cmd_args[]){
258      if(cmd_args[1] != NULL && strcmp(cmd_args[1], "&") == 0){
259          background = 1;
260      }
261      int cstatus;
262      int cpid = fork();
263
264      if (cpid == 0) { // in child
265          // evaluate arguments
266          char* evaluated_args[MAX_WORDS_NO];
267          int evaluation_state = 0;
268          strcpy(evaluated_args[0], cmd_args[0]);
269          int i, k;
270          for (i = (background == 1)? 2 : 1, k = 1; cmd_args[i] != NULL; i++, k++)
271          {
272              evaluated_args[k] = (char*) malloc(MAX_WORD_LENGTH*sizeof(char));
273              evaluation_state = evaluate_args(cmd_args[i], evaluated_args[k]);
274              if (evaluation_state == 0) {
275                  printf("Undefined variable!\n");
276                  return;
277              }
278              else if (evaluation_state == -1) {
279                  printf("No arguments passed!");
280                  return;
281              }
282              if (evaluated_args[k] != NULL){
283                  char* con = NULL;
284                  char* to_split = (char*) malloc(MAX_WORD_LENGTH*sizeof(char));
285                  strcpy(to_split, evaluated_args[k]);
286                  char* tok = strtok_r(to_split, " ", &con);
287                  while (tok != NULL) {
288                      evaluated_args[k] = (char*) malloc(MAX_WORD_LENGTH*sizeof(char));
289                      strcpy(evaluated_args[k], tok);
290                      tok = strtok_r(NULL, " ", &con);
291                      k++;
292                  }
293                  k--;
294              }
295          }
296          evaluated_args[k] = NULL;
297
298          // execute command
299          int status = execvp(cmd_type, evaluated_args);
300          if (status < 0){ // error occured within execvp()
301              // terminate the childe process
302              printf("ERROR: error occured on executing %s\n", cmd_type);
303              exit(1);
304          } else {
305              exit(0);
306          }
307      } else if (cpid > 0  && !background) { // parent and foreground run mode
308          do {
309              cpid = waitpid(cpid, &cstatus, WNOHANG);
310          } while (cpid == 0);
311      }
312      background = 0;
313      return;
314  }
```

Spawns a new process then checks if the parent process is executing and command is to be executed in foreground, waits for the child process to terminate then continues executing. If a child process is executing, executes the command passed to the function using execvp() then exits the child process.