Evolone Layne

To print a word in the terminal after the line has been tokenized, when arguments[0] is "echo", arguments[index] will be printed to the user.

```
if (strcmp(arguments[0], "echo") == 0){
         int index;
         for (index = 1; index < argcount; ++index){
           if(index > 1){
             printf(" ");
           }
           printf("%s", arguments[index]);
         }
         printf("\n");
       }
```

To print the working directory to the user, arguments[0] would need to equal "pwd". Cwd was previously defined with the built-in getcwd() to then display the working directory.

```
       else if (strcmp(arguments[0], "pwd") == 0){
         printf("%s\n", cwd);
       }
```

To change directories,arguments[0] would need to equal "cd". If the line is cd with nothing following it, it will just change to home, otherwise, it'll go to the listed directory.

```
else if (strcmp(arguments[0], "cd") == 0){
  if (arguments[1] == NULL){
    chdir(getenv("HOME"));
  }
  else{
    chdir(arguments[1]);
    }
  }
```

If the user types exit, it will stop running the shell by exiting.

```
else if (strcmp(arguments[0], "exit") == 0){
  exit(0);
}
```

If the user types in “setenv”, they will set variable names to a value after being tokenized.

```
else if (strcmp(arguments[0], "setenv") == 0){
  char name[MAX_ENV_VARIABLE_NAME_LENGTH];
  char value[MAX_ENV_VARIABLE_VALUE_LENGTH];
  char* p = strtok(arguments[1], "=");
  strcpy(name, p);
  while (p != NULL) {
    strcpy(value, p);
    p = strtok(NULL, "=");
  }
  setenv(name, value, 1);
}
```

"Env" will display all the variables if `arguments[0] == "env"`. The loop will iterate while there are variables to be printed.

```
  else if (strcmp(arguments[0], "env") == 0) {
    if (argcount == 1) {
      char ** envs = environ;
      for (;* envs; envs++) {
        printf("%s\n", * envs);
      }
    }

else {
      char * env_value = getenv(arguments[1]);
      if (env_value != NULL) {
        printf("%s\n", env_value);
      }
  }
}
```

Checks if it is not a built-in command, forks a process, loads the program from the file system, passes the arguments, and executes it. The shell must wait for the child process to terminate and report any errors that might have occurred (Lab 2 reference)

```
void cancelHandler(int signum) {
 printf("\n");
}

void handler(int signum)//signal handler
{
 exit(0);
}

else{
         pid_t  pid;
         pid = fork();
         char* cmd = arguments[0];
         int background = 0;
         if (strcmp(arguments[argcount - 1], "&") == 0){
           arguments[argcount - 1] = NULL;
           background = 1;
         }
         if (pid == 0){
           signal(SIGALRM,handler);
           alarm(10);
           if (background){
           int fd = open("/dev/null", O_WRONLY);
           dup2(fd, STDOUT_FILENO);
           dup2(fd, STDERR_FILENO);
         }
         if (execvp(cmd, arguments) == -1){
           printf("execvp() failed: No such file or directory. An
error occurred\n");
         }
         alarm(0);
```

```
          exit(0);
        }
          else {
            if (background == 0){
              signal(SIGINT,cancelHandler);
              waitpid(pid, NULL, WUNTRACED);
            }
          }
    }
```