OS Assignment 3

~ Isha Gupta (2018040)

In my Linux distribution:

- ls: /bin/ls
- sort: /usr/bin/sort
- uniq: /usr/bin/uniq

Therefore, the entered command is:

/bin/ls | /usr/bin/sort | /usr/bin/uniq

Flow of code:

```
read_command(0, &cmd, &args);
```

->

```
get_num_pipes(args);
```

->

```
piping(args,num_pipes)
```

->
recursive calls to piping() (till the base case is hit) and at each recursive step,
`execute_command(temp_args)`
is called.

```
read_command(0, &cmd, &args);
```

- Initializes a buffer of a default size: `char *buf = malloc(sizeof(char)*BUFFER_SIZE);`
- read system called issued and command from terminal (ie from the program) read into the buffer and length of command returned `int ret_read = read(fd,buf,BUFFER_SIZE);`
- Allocation space for args list using malloc. `*args = malloc(sizeof(char*)*ret_read);`
- Traversing over the buffer char by char and appending each command by splitting around " " to the args list received in the parameter. `(*args)[arg_count]=arg;`
- Keeping a count of the number of arguments in the args list and finally appending NULL to it.
- cmd has the first value of the args list `*cmd = (*args)[0];`

```
get_num_pipes(args);
```

- Counts the number of pipes in the given args list and returns to num_pipes.`int num_pipes = get_num_pipes(args);`
- Counts by traversing the args list and incrementing a counter if a pipe (|) is encountered.

```
while (args[i]!=NULL){
        if (strcmp(args[i],"|")==0){
             count++;
        }
        i++;
   }
```

```
execute_command(args)
```

- Essentially a big if case that identifies commands and does the required redirections and removes the corresponding arguments from the args list.
- Traverses over the given args list, identifies commands of different types and calls the corresponding functions which manipulate the FDT accordingly and are not added to a newargs list which contains only the commands which have to be passed to the exec system call.
- In the given command, the args received during each call to execute command are:
    - {/bin/ls, NULL}
    - {/usr/bin/sort, NULL}
    - {/usr/bin/uniq, NULL}
- None of the above args list contain any redirection commands (example: >, >> 1>filename etc.) so they are directly added to the newargs list.
- args is updated to newargs.
- exec system call is made

```
execvp(args[0],args);
```

```
piping(args,num_pipes);
```

- A recursive function that recursively executes every portion of a pipe command.
- Initializing a file descriptor:

```
    int fd[2];
    pipe(fd);
```

- Base Case: If the number of pipes is 0 ie the command has to be executed directly. C

```
if (num_pipes==0){
        execute_command(args);
```

```
    }
```

- Recursive Call: when num_pipes!=0:
    - A temporary argument array is created having arguments only for this pipe portion ie if our command is a | b | c | d and we are at the 2nd recursive call, then temp_args will contain the args for command b.
    - Reduce the number of pipes.
    - Update current args list to point to the next command after this pipe ie after b will be executed, the args list should point to c | d:
        - `args = (args+count_args+1);`
    - Fork the current thread, and
        - On the child thread: A
            - Change the file descriptors for the reading end and execute the current command (ie b)

```
if (pid==0){
              close(fd[0]);
              close(1);
              dup(fd[1]);
              close(fd[1]);
              execute_command(temp_args);
         }
```

            - After the execute_command function, which calls the execvp system after which, the child thread doesn't return.
            - The output for command b is written in fd[1].
        - On the parent thread: B
            - The reading end of the pipe's fd is changed and the previous command's output is read.
            - Recursive call is made to the piping function with args from c | d onwards the function executes till it hits the base case.

```
close(fd[1]);
              close(0);
              dup(fd[0]);
              close(fd[0]);
              piping(args,num_pipes);
```

- In the given command, first /bin/ls will be executed in A, then recursive call is made, after which /usr/bin/sort goes in A, recursive call made, which hits the base case at /usr/bin/uniq in C.

We wait for the parent

```
        else if (pid > 0) {
            wait(NULL);
            free(cmd);
            free_args(args);
        }
```

and here the args and cmd are freed via the free_args(args) function which iterates over the args list and frees all the items in it and then frees the array itself

```
void free_args(char **args){
    int i=0;
    while (args[i]!=NULL){
        free(args[i]);
        i++;
    }
    free(args);
}
```