# Shell for Minix and Linux Kernels

## Purpose

The purpose of this project is to provide a command line interface to the operating systems like Minix and Linux where users can type commands themselves and execute them.

## Available Resources

The project utilizes the underlining System Call APIs to execute commands entered on the shell prompt. GNU readline library was used for saving command history to provide the facility to users for iterating through the previous commands, besides it was also employed to take user input from CLI. Standard C library was used for its services.

## Design

The shell program is divided into several sections for performing various functionalities. The shell starts with invoking a 'init_sh' procedure for initializing the shell and environment variables from the profile file. After initialization the next step is to reload the command history from '.history' file present in the home directory followed by registering and setting up signal handlers.
After the initial setup, the main shell code is executed. The main shell code is responsible for taking up the user input and parsing it to get executable commands in proper formats which are then operated upon. The main shell code resides inside an infinite loop to provide continuous prompt to end user until and unless user chooses to exit using the 'exit' command. The commands executed on the shell should be present in the path environment variable setup during initially. The environment can be modified by simply making changes to the profile file.
There are separate procedures for handling signals given on CLI and after signal handling is done the control is again redirected to the main shell prompt.

### Important Global variables

```
pid_t pid_arr[50]; // Array to hold all running(foreground) pids

pid_t pid_arr_bg[50]; // Array to hold all running(background) pids

int pid_arr_index=0; // index of last entry in pid_arr array

pid_arr_bg_index=0; // index of last entry in pid_arr_bg array

sigjmp_buf ctrlc_buf; // control jump point after handling events
```

### Design of each function

```
void init_sh()
  change directory to $HOME
  Set PATH variable
  Read previous history from file
```

```
bool error_check(char *)
  Iterates over the string and checks for syntax errors like & |.

void freeHistory()
  Access the command history list
  Release the dynamically allocated memory used for storing command history

void sigint_handler(int signum)
  Handles the SIGINT signal sent on pressing the 'Ctrl-C' key combination.
  Send SIGTERM to all running processes
  Jump to set point in main

void sigtstp_handler(int signum)
  Handles the SIGTSTP signal sent on pressing the 'Ctrl-Z' key combination.
  Move running process to background array
  Reset foreground array index
```

Following the function containing the main shell code:

```
int main(int argc,char * argv[],char **envp)
  Initialize the shell and environment variables
  Setup and register signal handler
  Run the loop for shell prompt
  Take command as input from users
  Do necessary error handling
  Execute the command involving pipes, redirections, background processes and
  optional arguments.
  Free dynamically allocated memory and flush buffers
```

Implementation of pipes and redirections:

```
  The shell implements multiple pipes in a command by forking and executing
  all commands. Almost simultaneously and then changing their input and
  output file descriptors to the pipe. File redirections are special case
  where we redirect the output or input to the file instead of the pipe.
```

Implementation of background processes and signal handlers:

```
  The entered command must contain the '&' symbol at the end of the command
  to indicate that it's a background process. We maintain a flag for
  background processes and change the pid's group ID so that the signals
  sent to the background processes are not forwarded to the child processes.
  The shell defines its own signal handlers for SIGINT
  and SIGTSTP and hence the shell does not quit or pause when they occur.
```

## Testing

The test strategy for this design will be to try out various combinations of input commands involving pipes, redirections, background processes and optional arguments.

Another test we may want to run is to send various input signals to check if signal handling is done properly. We may even try executing an invalid command to check error handling implementation.