**Unix Shell and History Feature**

**SJSU Fall -2016**

**Project Report**

**By**

**Kanti Bhat (011488386)**

**Introduction:**

“Unix shell and History feature” program provides a UNIX shell type interface that allows the user to execute basic Unix/Linux commands. It redirects command results to default output device (screen / STDOUT). This program is implemented using parent and multiple child process concept supported by UNIX/Linux type operating systems. parent process forks a child process to execute user commands until user exits the shell.

This program supports few other features like running processes in the background where parent process and its children progress concurrently. Here user command is executed as an independent child process. Parent process on the other hand, checks on regular intervals for completion of these independent child processes and notifies the user once complete.

This Program do support history feature where it displays last 10 commands entered by the user. Two special pattern commands enables the user to execute the most recent command as well as any command from the list of commands saved in the history.

**Background:**

This program is a part of academic Individual studies on operating systems and aims at exploring and utilizing various features and services provided by the operating system.

This program utilizes many of the services provided by Operating systems. However, below are some features being utilized extensively than the rest:

* Program Execution – This is one of the major OS service used by all the user programs. Operating system loads the program into memory. Handles program execution. It provides a mechanism for process synchronization, process communication and manages deadlock handling. This program makes use of many aspects of process handling offered by OS .
* I/O operations – Different standards and peculiarities of various I/O devices are handled smoothly by the operating system such that I/O handling stays transparent to the user. operating systems carry out the communication between users and the I/O device drivers . I/O operation is any read /write operation with any file or I/O device. This program makes use of default input device (keyboard/STDIN) to get the commands from the user and default output device (screen / STDOUT) to display the results of the user commands.
* Memory Management – Every program needs to be loaded into the memory to run as a process. Along with instruction memory a process operates on data memory. This program makes extensive use of both the memories. Data memory of this program for example comprise of an array of structures to maintain process Id and names of processes running in the background, another list of dynamically allocated linked list of nodes to save command names for the history feature and many others.
* Error Handling – Errors and interrupts can occur anytime and anywhere. For example in this program an attempt to create a child process using fork() system call may fail. Like wise it may occur in I/O devices or in memory. Operating system take care of errors by means like triggering the exceptions and providing a means to catch these exceptions in the user program.
* Resource Management – This program is a multi tasking program. User can run any (total of 100 in this program) number of commands in the background. Such multi tasking and multi user environment needs resources like memory, CPU cycles, files storage to be shared. We can see how efficiently OS schedules processes and manages to share these resources among the processes from this program.

**Approach:**

This project is implemented in two small parts. shell type interface is the goal of this project.

**Part I :** **execution of basic shell commands entered by the user.**

1. Declare and initialize the required variables.

2. do the following in loop:

2.1 Display user prompt. Wait for user command.

2.2 on command input use fork() system call to create a child process.

2.3 In the child process replace the child’s address space with the new binary of the command

2.4 In the parent process:

2.4.1 Save the child process Id into an array if user has requested a background service and loop back to step 2.

2.4.2 wait for the child process to complete.

2.4.3 print the the return status of the child on the stdout.

2.4.4 check the background processes array to see if there is any processes running in the background. If processes exists:

a. do the following in the loop

b. check for next exited child process from the background processes array.

c. wait for the exited child process.

d. print the return status of that child.

**Part II : history command support to show upto 10 commands recently input user commands.**

This is implemented as a First in First out Linked list in this project.

steps:

1. when user enters a command this command (name) is saved as a new node at the beginning of this linked list.

2. If there are more than 10 nodes after adding a node at the beginning, then remove the oldest node (which is at the rare end of the list) from the list.

3. Another detail saved along with the command name in each node is the sequence number. If a total of 15 commands entered till the moment in the current session, then most recent 10 commands are saved in the linked list.

Ex: If below is the order of the commands entered by the user in that respective session:

1.ps → (least recent)

2.ls -l

3 cal

4. top

5. who

6. date

7. ls

8. pwd

9. chmod

10. mv

11. cp

12. rmdir

13. ps

14. more

15. history → (most recent)

k\_sh > history

10 . history

9 . more abc.txt

8 . ps

7 . rmdir

6 . cp

5 . mv

4 . chmod

3 . pwd

2 . ls

1 . date

4 Entering the command !8 executes the command numbered 8th from the commands stored in the history. Here the program recognizes the special pattern with “!” character.

k\_sh > !8 ==> execute the ps command in the above example session.

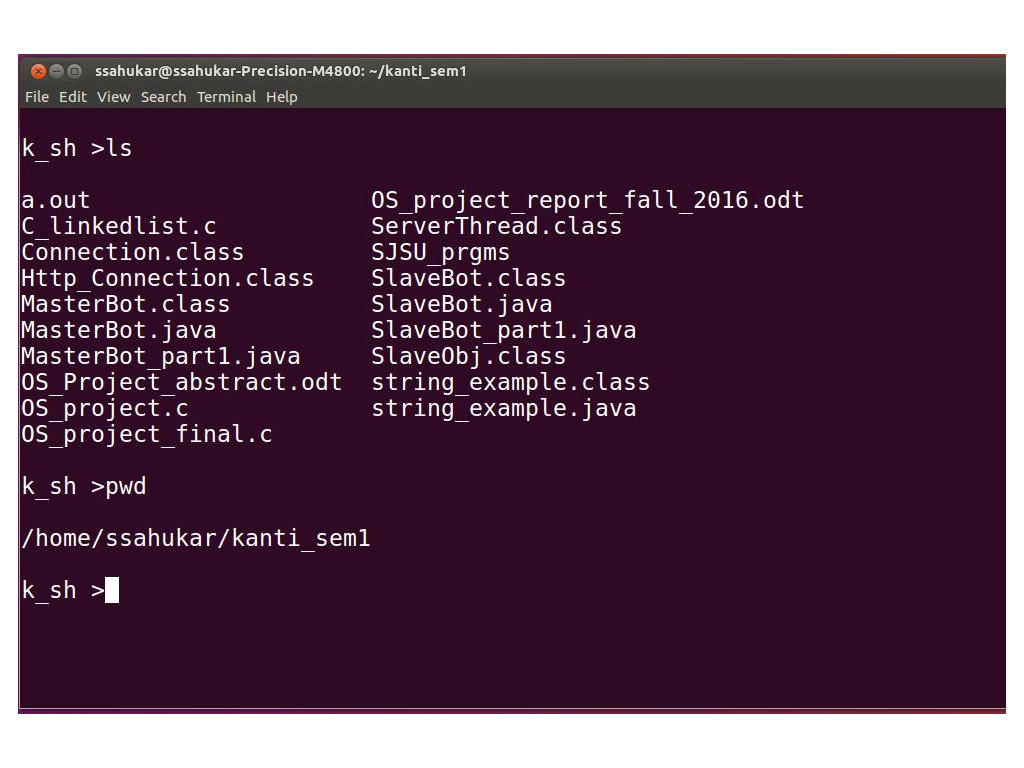
4.1 Entering the command !! executes the most recent command from the commands stored in the history.

k\_sh > !! ==> executes history command in the above example session.

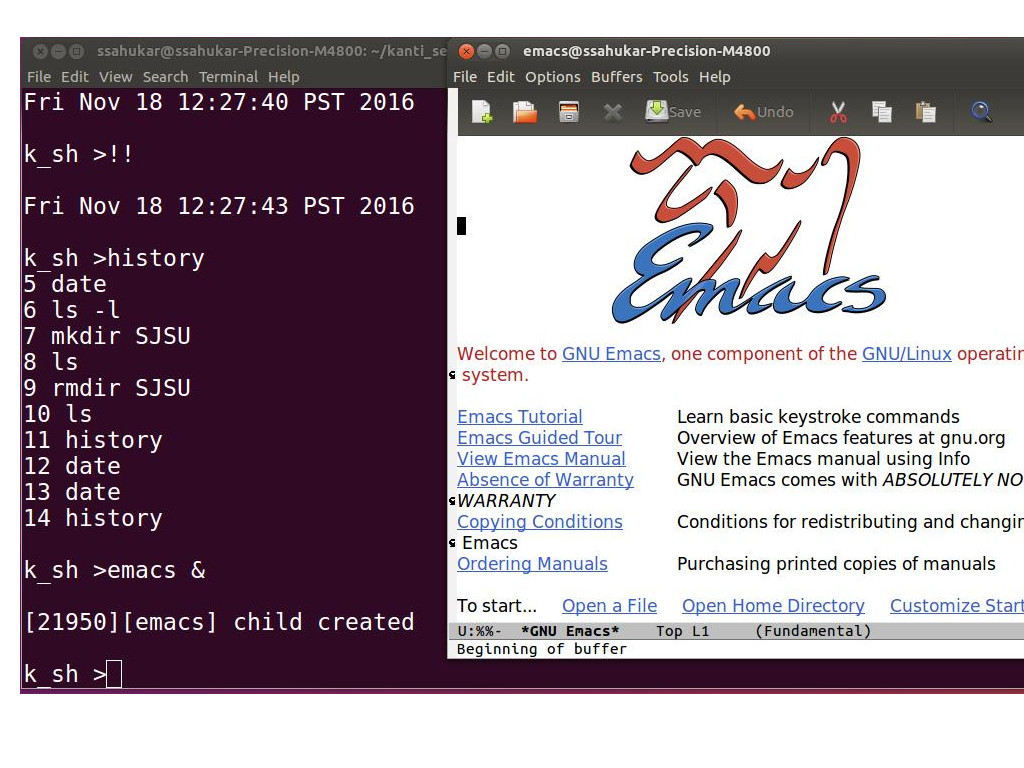
**Results:**

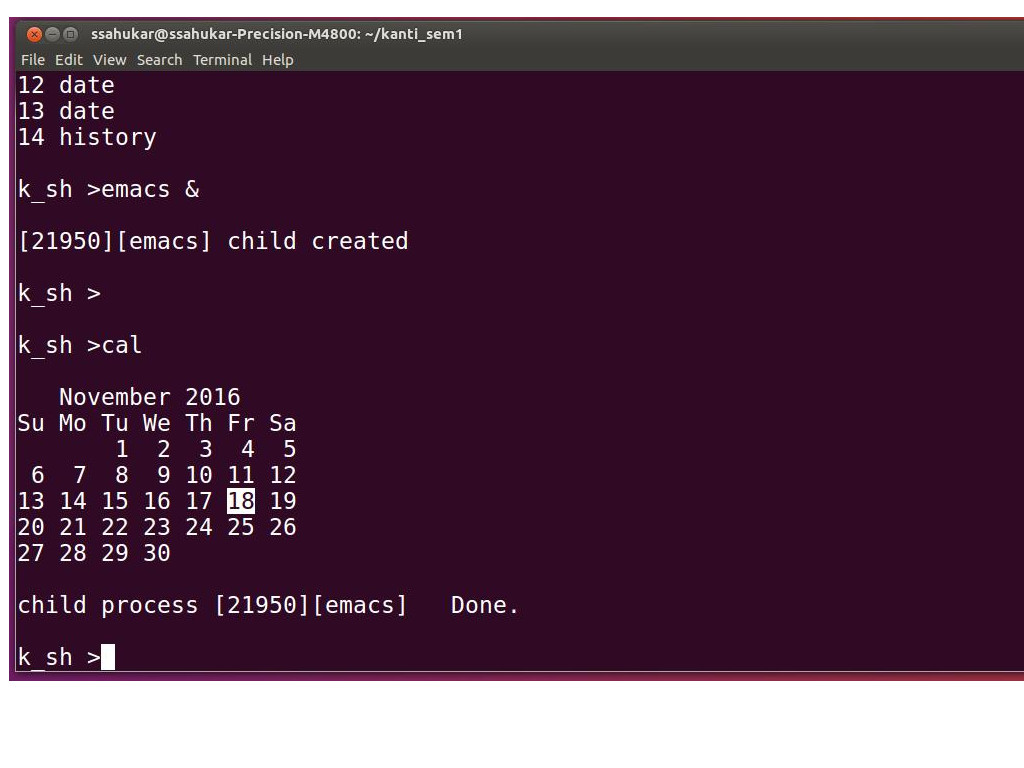
Below are some of the screen shots of program output:

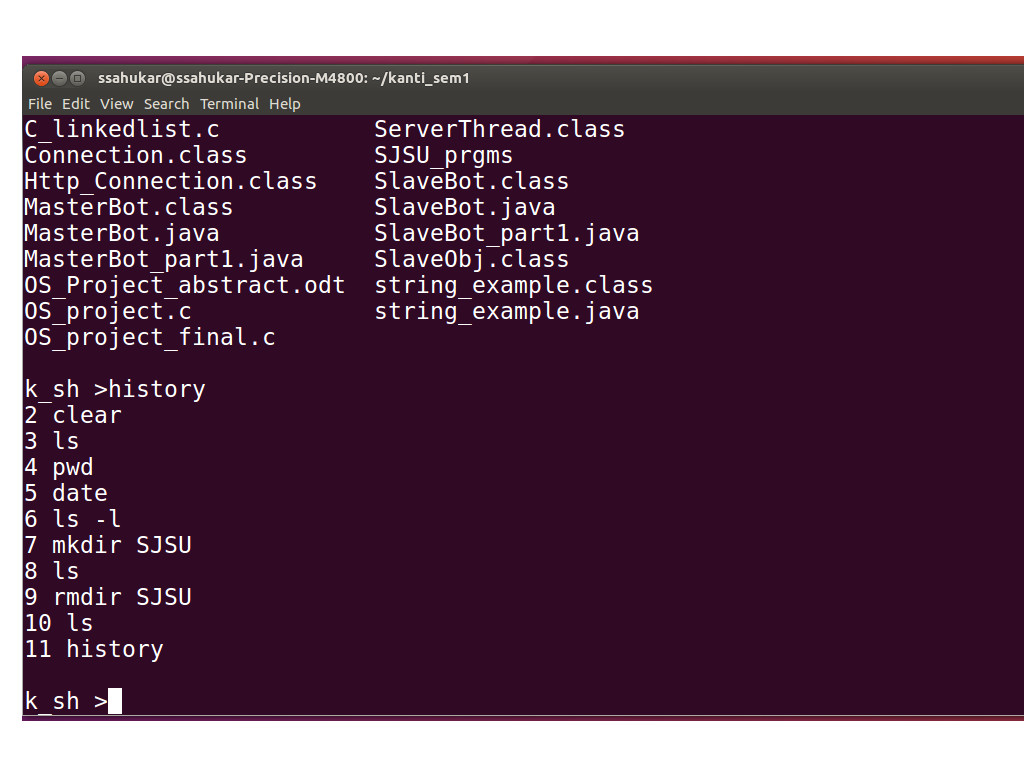
1. User command execution



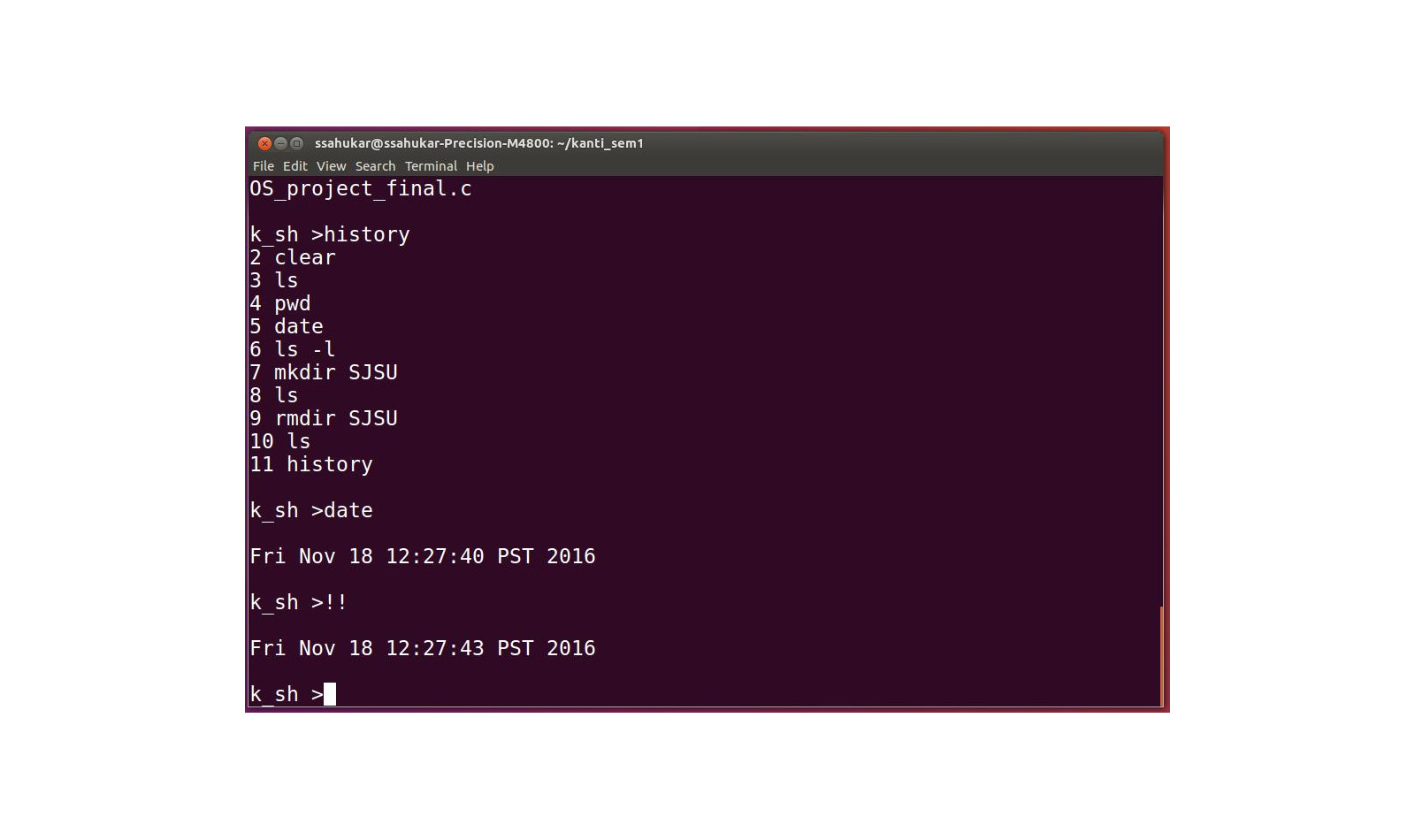
2. User command execution in the background





3. execution of history command:

4. Execution of special !! (most recent command in the history) command:



**Analysis/Comparison:**

This program is implemented in a short time span for an academic requirements and hence does not support an extensible range of features like the one provided by Bourne again shell (bash), Korn Shell (ksh), POSIX shell(sh) and many more.

Most of the linux commands are supported along with the history command to provide some flexibility to the user. How ever some of the advance features like → piping the output of one command as input of another command , search file / directory feature using wild card characters like \* and other regular expressions is not supported. However these are in the improvement scope for this project.

**Summary:**

Implementation of Unix shell and History feature was a good experience of exploring and witnessing different aspects and services offered by the operating system. This program partially mimics the standard Linux shells and accomplishes command execution given by the user. Implementing an application which mimics the system utility allows us to see, analyze, and understand how processes are created and managed, how external I/O devices are handled and many more, from an operating systems point of view. This program gives a glimpse of operating systems process management and I/O management and has scope for further improvement and chance to explore and understand file system management and other aspects.

**Appendix:**

Program Code:

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <errno.h>

#define true 1

#define false 0

#define SIZE 100

#define LIMIT 20

struct proc{

int proc\_id;

char proc\_name[20];

};

struct node{

int cmd\_cnt;

char pname[100];

struct node \*next;

};

/\* Function to print the list\*/

void print\_list(struct node \*\*root)

{

struct node \*temp;

temp = \*root;

while(temp !=0)

{

printf("%d %s \n",temp->cmd\_cnt,temp->pname);

temp = temp->next;

}

}

/\*Function to add a node at the end of the list\*/

void add\_node(struct node\*\* root,int cmd\_count, char f\_pname[])

{

struct node \*temp;

struct node \*new\_node;

temp = \*root;

if (\*root != NULL) {

while(temp->next != 0) {

temp = temp->next;

}

new\_node = (struct node \*)malloc(sizeof(struct node));

new\_node->cmd\_cnt = cmd\_count;

strcpy(new\_node->pname, f\_pname);

new\_node->next = 0;

temp->next = new\_node;

} else {

\*root = (struct node \*)malloc(sizeof(struct node));

(\*root)->cmd\_cnt = cmd\_count;

strcpy((\*root)->pname, f\_pname);

(\*root)->next = NULL;

}

}

int count\_nodes(struct node \*\*root)

{

int count =0;

struct node \*temp;

if(\*root != NULL){

temp = \*root;

while(temp != NULL){

count++;

temp = temp->next;

}

}

return count;

}

/\*Function to delete a node with pid = f\_pid \*/

int delete\_node(struct node \*\*root, int nodes)

{

struct node \*temp;

struct node \*temp1;

int flag = false;

for(int i = nodes; i>10;i--)

{

temp = \*root;

\*root = (\*root)->next;

free(temp);

}

}

char\* find\_node(struct node \*\*root, int count)

{

struct node \*temp;

temp = \*root;

while(true) {

if(temp->cmd\_cnt == count)

break;

else

temp = temp->next;

}

return temp->pname;

}

void parse(char line[], int len, char \*\*arr, int \*cnt)

{

int i=0;

int j=0;

char my\_str[50][50];

for(int p=0;p<len+1;p++)

{

if(line[p] == ' ' || line[p] == '\r' || line[p] == '\t' || line[p] == '\0' ){

my\_str[i][j] = '\0';

arr[i] = my\_str[i];

i++;j=0;

++(\*cnt);

continue;

}

my\_str[i][j] = line[p];

j++;

}

printf("\n");

}

/\*function returns true it there is any some

background job running.false otherwise.\*/

int check\_bckgrnd\_process(struct proc proc\_arr[])

{

int ret\_value = false;

for(int i=0;i<SIZE;i++){

if(proc\_arr[i].proc\_id != 0){

ret\_value = true;

break;

}

}

return ret\_value;

}

void clear\_bakgrnd\_procs(struct proc cp\_arr[])

{

int ret;

int return\_status;

for(int k=0;k<SIZE;k++)

{

if(cp\_arr[k].proc\_id != 0)

{

return\_status = 0;

ret = waitpid(cp\_arr[k].proc\_id, &return\_status,WNOHANG);

if(ret && (WIFEXITED(return\_status) || WIFSIGNALED(return\_status)))

{

if (return\_status == 0){

printf("child process [%d][%s] Done. \n",cp\_arr[k].proc\_id, cp\_arr[k].proc\_name);

}

cp\_arr[k].proc\_id = 0;

cp\_arr[k].proc\_name[0] = '\0';

}

}

}

}

void main()

{

pid\_t cpid1;

int s, return\_status,status,len = 0;

int words\_count = 0,back\_flag = 0;

int nodes = 0;

int count = 0;

char \*cmd[50];

char name[20];

char c;

char line[SIZE];

struct proc cp\_arr[SIZE];

struct node \*root = NULL;

for (int j = 0;j < SIZE; j++){

cp\_arr[j].proc\_id = 0;

strcpy(cp\_arr[j].proc\_name,"");

}

signal(SIGINT, SIG\_IGN);

while(true)

{

memset(line,'\0',SIZE);

len =0;

back\_flag = 0;

words\_count = 0;

for(int i=0;i<50;i++)

cmd[i] = NULL;

printf("\nk\_sh >");

if(fgets(line,sizeof(line),stdin) != NULL)

{

if (line[0] == '\n')

continue;

len = strlen(line);

line[len-1] = '\0';

len = strlen(line);

if(line[len-1] == '&')

{

int i;

back\_flag = true;

line[len-1] = '\0';

len = strlen(line);

for(i=len-1;i>=0;i--){

if (line[i] == ' ')

line[i] = '\0';

else

break;

}

line[i+1] = '\0';

len = strlen(line);

}

if(0 == strcmp(line, "!!")){

if(root != NULL){

strcpy(line,find\_node(&root, count));

len = strlen(line);

} else {

printf("No commands in the history\n");

continue;

}

}

add\_node(&root,++count,line); //add command name to the list to maintain the history

nodes = count\_nodes(&root);

if (nodes > 10) {

delete\_node(&root, nodes);

}

if(0 == strcmp(line,"exit"))

exit(0);

if(0 == strcmp(line, "history"))

{

//display the list

print\_list(&root);

continue;

}

//parse user input and save them as tokens into cmd array

parse(line,len,cmd,&words\_count);

//strcpy(name, cmd[0]);

cpid1 = fork();

if(cpid1 < 0) {printf("ERROR in fork.\n");continue;}

else if(cpid1 == 0)

{

/\*This is child process\*/

if (0 > execvp(cmd[0], (char \*\*)cmd))

{

printf("ERROR: %s: %s\n", cmd[0], strerror(errno));

exit(-1);

}

}

else if(cpid1 > 0)

{

if(back\_flag != 1)

{

waitpid(cpid1,&return\_status,0);

//if (WIFEXITED(return\_status) || WIFSIGNALED(return\_status))

//{

// printf("\nchild process %d[%s] Done. \n",cpid1, name);

//}

/\*check if any pending background processes exist\*/

status = check\_bckgrnd\_process(cp\_arr);

if(status) {

clear\_bakgrnd\_procs(cp\_arr);

}

} else if(back\_flag == 1)

{

for( s=0;s <SIZE;s++)

{

if(cp\_arr[s].proc\_id == 0){

break;}

}

if (s == SIZE)

{

printf("Maximum limit of background processes running already..\n");

printf("Try again after some time..\n");

continue;

}

cp\_arr[s].proc\_id = cpid1;

strcpy(cp\_arr[s].proc\_name,cmd[0]);

printf("[%d][%s] child created\n",cpid1, cmd[0]);

}

}

} else

{

continue;

}

}

}