“MAKING OF SHELL IN WINDOWS”

PROJECT REPORT

Submitted for the course: CSE2005 OPERATING SYSTEM

By

ANIKET GUPTA (17BCE2018)

Slot: F1

Name of faculty: Dr. K. Jayakumar

(SCHOOL OF COMPUTER SCIENCE AND ENGINEERING)

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CERTIFICATE

This is to certify that the project work entitled “MAKING OF SHELL IN WINDOWS” that is being submitted by “ANIKET GUPTA (17BCE2018), is a record of bonafide work done under my supervision. The contents of this Project work, in full or in parts, have neither been taken from any other source nor have been submitted for any other CAL course.

ACKNOWLEDGEMENTS

I take immense pleasure in thanking Dr. G. Viswanathan, my beloved Chancellor, VIT University and respected Dean, for having permitted me to carry out the project.

I express gratitude to my guide, Dr. K. Jayakumar, for guidance and suggestions that helped me to complete the project on time. Words are inadequate to express my gratitude to the faculty and staff members who encouraged and supported me during the project. Finally, I would like to thank my ever-loving parents for their blessings and my friends for their timely help and support.

ABSTRACT

Shell program (sometimes called a shell script) is a text file that contains standard UNIX and shell commands. Each line in a shell program contains a single UNIX command exactly as present in linux. The difference is that you can execute all the commands in a shell program simply by running the shell program (rather than typing all the commands within) and that also by using C programming. Shell programs are interpreted and not compiled programs. This means when you run a shell program a child shell is started in the compiler window. This child shell reads each line in the shell program and carries out the command on that line. When it has read all the commands the child shell finishes, we have provided the maximum shell program which can be executed in actual Linux shell script.

INTRODUCTION

We started the project by using our literary survey we presented in the first review. All our members worked hard in the research work. Then we found out the library which has to be imported in the C language.

PROBLEM STATEMENT

Our problem statement is to create shell in windows using C. Shell script usage has not reduced instead increased over the years. Most of the middleware tools such as ETL, MB etc. are installed in UNIX servers. To maintain those servers, pre-processing file, log collection and performance improvement shell scripts are used. In this project we are using C coding to implement some of the features present in a typical shell such as arithmetic operation, viewing history, etc.

Technical Specification

Software needed is Code Blocks.

Different Importing of libraries.

Studying of System calls.

Implementation of these in the project.

Debugging of errors.

Etc…

LITERARY SURVEY

Sr.no.	**NAME OF THE RESEARCH PAPER**	**AUTHOR AND YEAR**	**PROPOSED METHODS**	**RESEARCH GAP**

1.	A Packet I/O Architecture for Shell Script-based Packet Processing	Yohei Kuga, Takeshi Matsuya, Hiroaki Hazeyama, Kenjiro Cho\ Rodney Van Meter, Osamu Nakamura	The method proposed is a new scripting model for rapid and easier development of packet processing using shell scripts. In this paper we present EtherPIPE, a character network 110 device, that allows the programmer to access network traffic data as a file through UNIX commands. By setting a UNIX pipe "I" from or to EtherPIPE's output or input with UNIX commands, packets can be easily processed, executing functions such as packet filtering, packet capturing, generating arbitrary packets, and rewriting header information. In order to prove the utilities of our model, we have developed FPGA-based EtherPIPE adapter using a commodity FPGA card and a character device driver featuring new offloading functions.	EtherPIPE is a low-layer network device yet, its data format is simple and easy to handle in commands and scripting languages. Therefore, EtherPIPE can be used not only for simple network scripting but also for more complex packet processing using scripting languages. We believe that EtherPIPE is suitable for SDN where simple packet manipulations are often required..
2.	A Login Shell for Computing Grid	Xiaoning Wang, Jian Lin, Yongqiang Zou, Li Zha	Shell is a most important user interface for HPC users in scientific computing. This paper presents a login shell for computing grid, called GShell, which provides an integrated and uniform environment for constructing, running and managing grid	GShell allows legacy host tools to be integrated in a grid computing environment. At present, GShell has been only implemented over CNGrid

			applications. Besides conventional functionality of shell, GShell provides gridlevel functionality including grid context maintenance, grid application management interfaces, and extended pipe and redirection semantics. Grid contexts avoid repeat information input when a grid application is started. All the grid applications started in GShell are managed in the grid system, and they can be monitored or stopped in a global scope. Extended shell scripts allow constructing new grid applications using existing tools. At present, GShell has been implemented and applied in the CNGrid project, and is proved to be a simple and flexible command-line tool for most scientists to work in the grid.	GOS, and applied in the CNGrid project.
3.	Problem and Project-based Learning in Scripting Lab	Shantala Giraddi, Shilpa Yaligar , Kavitha H.S	Scripting language employ high-level constructs to interpret and execute one command at a time. In general scripting languages are easier to learn and faster to code than structured and compiled languages such as C and C++. Scripting languages have many important advantages over traditional programming languages. In future the usage of these languages is likely to increase. In this paper we discuss and report our experience in teaching scripting	More energy and resources are required to implement this which is not discussed in this research paper.

			languages lab at the undergraduate level, 4th semester. Scripting language is an umbrella term used for languages like unix shell, TCL, perl, java, python and LISP. Out of these, we have chosen UNIX shell programming and python for our curriculum. The authors report various pedagogical activities like multiple assignments, peer assessment within a group, self learning through e-resources and course project that were employed during the course. The course projects were specially designed so as to make students explore the vast number of python packages. The authors found that these activities definitely enhance the learning experience and there was a remarkable change in the learning level of the students as compared to previous years as evident in the grades obtained by the students.	
4.	Clustered Workflow Execution of Retargeted Data Analysis Scripts	Daniel L. Wang, Charles S. Zender, and Stephen F. Jenks	Supercomputing advances have enabled computational science data volumes to grow at ever increasing rates, commonly resulting in more data produced than can be practically analyzed. Whole-dataset download costs have grown to impractical heights, even with multiGbps networks, forcing scientists to rely on server-side subsetting	Limitations on syntax and analysis program choice limit end-user flexibility.

			and limiting the scope of data	

			they can analyze on a workstation. Our system supplements existing scientific data services with lightweight computational capability, providing a means of safely relocating analysis from the desktop to the server where clustered execution can be coordinated, exploiting data locality, reducing unnecessary data transfer, and providing end-users with results several times faster. We show how dataflow and other compiler-inspired analyses of shell scripts of scientists' most common analysis tools enables parallelization and optimizations in disk and network I/O bandwidth. We benchmark using an actual geoscience analysis script, illustrating the crucial performance gains of extracting workflows defined in scripts and optimizing their execution. Current results quantify significant improvements in performance, showing the promise of bringing transparent high-performance analysis to the scientist's desktop.	
5.	An Intelligent Control Shell for CAD Tools	Satoru Fujita, Motohide Otsubo, Masanobu Watanabe	This paper describes an intelligent control shell for CAD tools, which can automatically create a command sequence to control CAD systems using symbolic knowledge of general command flows and nonsymbolic knowledge of	Cooperation among multiple control systems to obtain better solutions than the stand alone system has not been discussed.

			the past execution data. Users define a model of possible control flows, which are transformed into a state transition graph from which executable command sequences are inferred. The control system statistically analyzes nondeterministic branches, where a final result is predicted from a current state of a design object, a command history and the succeeding commands. Then, the most promising command to optimize the design objects is selected and executed. The LSI CAD system controlled by the proposed shell synthesizes about 5% faster circuits than that synthesized with a standard script for delay minimization.	
6.	A Login Shell for Computing Grid	Xiaoning Wang, Jian Lin, Yongqiang Zou, Li Zha	Shell is a most important user interface for HPC users in scientific computing. This paper presents a login shell for computing grid, called GShell, which provides an integrated and uniform environment for constructing, running and managing grid applications. Besides conventional functionality of shell, GShell provides gridlevel functionality including grid context maintenance, grid application management interfaces, and extended pipe and redirection semantics. Grid contexts avoid repeat information input when a grid application is started. All	A wider array of grid systems and enhancing the supports for common workflow management needs to be worked on.

			the grid applications started in GShell are managed in the grid system, and they can be monitored or stopped in a global scope. Extended shell scripts allow constructing new grid applications using existing tools. At present, GShell has been implemented and applied in the CNGrid project, and is proved to be a simple and flexible command-line tool for most scientists to work in the grid.	
7.	PyORBIT A PYTHON SHELL FOR ORBIT	J.-F. Ostiguy Fermi National Accelerator Laboratory, Batavia, IL J. Holmes, ORNL Oak Ridge, TN	ORBIT is code developed at SNS to simulate beam dynamics in accumulation rings and synchrotrons. The code is structured as a collection of external C++ modules for SuperCode, a high level interpreter shell developed at LLNL in the early 1990s. SuperCode is no longer actively supported and there has for some time been interest in replacing it by a modem scripting language, while preserving the feel of the original ORBIT program. In this paper, we describe a new version of ORBIT where the role of SuperCode is assumed by Python, a free, welldocumented and widely supported object-oriented scripting language. We also compare PyORBIT to ORBIT from the standpoint of features, performance and future expandability.	While it is certainly usable as it stands, some work remains to be done before it can be considered a production tool, mostly in connection with exception handling and error recovery.

8.	Parallelizing the Execution of Sequential Scripts	Zhao Zhang, Justin M. Wozniak, Daniel S. Katz	Scripting is often used in science to create applications via the composition of existing programs. Parallel scripting systems allow the creation of such applications, but each system introduces the need to adopt a somewhat specialized programming model. We present an alternative scripting approach, AMFS Shell, that lets programmers express parallel scripting applications via minor extensions to existing sequential scripting languages, such as Bash, and then execute them inmemory on large-scale computers. We define a small set of commands between the scripts and a parallel scripting runtime system, so that programmers can compose their scripts in a familiar scripting language. The underlying AMFS implements both collective (fast file movement) and functional (transformation based on content) file management. Tasks are handled by AMFS's built-in execution engine. AMFS Shell is expressive enough for a wide range of applications, and the framework can run such applications efficiently on large-scale computers.	Deployment AMFS Shell on different platforms, such as supercomputers with other architectures and commodity clusters still needs to be worked on.

9.	A Web Page Malicious Code Detect Approach Based on Script Execution	Zhi-Yong Li, Ran Tao, ZhenHe Cai ,Hao Zhang	Web page malicious code detection is a crucial aspect of Internet security. Current web page malicious codes detection work by checking for “signatures”, which	Web page malicious code detectors have traditionally relied upon syntactic approaches, typically based
			attempt to capture (syntactic) characteristics of the known malicious codes. This reliance on a syntactic approach makes such detectors vulnerable to code obfuscations, increasingly used by malicious code writers, which alter syntactic prosperities of the malicious code without affecting their execution behavior significantly. This paper takes the position that the key to web page malicious code lies in their execution behavior. It proposes a script execution behavior feature based framework for analyzing propose of malicious codes and proving properties such as soundness and completeness of these malicious codes. Our approach analyses the script and	on signaturematching. While such approaches are simple, they are easily defeated by obfuscations.

			confirms the script which contains malicious code by finding shellcode, overflow behavior and hidden hyper link. As a concrete application of our approach, we show that the script execution behavior based web page malicious code detector can detect many known malicious code but also the newest malicious code.	
10.	A File System Abstraction and Shell Interface for a Wireless Sensor Network Testbed	Andrew R. Dalton, Jason 0. Hallstrom	Despite tremendous research interest and increased adoption, deeply embedded sensor networks are difficult to design, debug, and deploy; ultra-	While the graphical user interface has proven useful especially for students and novice users there are two significant

			dependability remains an elusive goal. To address	

				these difficulties, we have previously presented an interactive, server-centric testbed for wireless sensor networks that targets systems constructed using nesC and TinyOS - the emerging standard in sensor system development. The testbed infrastructure exposes an API suite that enables users to rapidly configure, instrument, compile, install, and profile their systems on one or more remote network deployments. The prototype deployment consists of 80 Tmote Sky devices arranged in a regular grid. The architecture is extensible in both the hardware and software dimensions to foster adoption and specialization. In this paper, we demonstrate the extensibility of the testbed software design, and present a novel file system abstraction and shell interface developed using the original API suite. The design of the new interface is informed by user feedback from client institutions where the standard graphical interface is being used to support research and teaching activities. The new shell interface complements the traditional graphical interface, reducing interaction latency, and enabling programmatic experimentation through an interpreted scripting	challenges to overcome. First, when accessed from outside of its hosting domain (i.e., the Clemson campus), the graphical interface introduces interaction delays that compromise the freshness of profiling data, and reduce the overall usability of the tool. Second, and more important, the tool is ill-suited to performing tasks that are complex, repetitive, and/or involve a large number of devices. An interpreted scripting interface is preferable in such cases.

			facility. We present the design and implementation of the new testbed interface, and present a small, but representative case-study that illustrates its utility.	

DESCRIPTION

The following functions have been implemented in this project:

1) Basic arithmetic operations like addition, subtraction, multiplication, division and taking exponential power.

2) A basic text editor to create and edit text files.

3) Printing the current directory and the files in the directory.

4) It allows us to see a maximum of 10 items as the history of the previous commands entered.

5) See the current date and time.

6) Call the inbuilt compiler to compile files (C/C++ files) and display their output.

7) Start any other process using the compiler.

CODE

```
#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<ctype.h>

#include<unistd.h>

#include<math.h>

#include<dirent.h>

#include<time.h>


int i,j,ec,fg,ec2; int last = 0; char fn[20],e,c;
```

```
FILE *fp1,*fp2,*fp;

void Create();
void Append();
void Delete();
void Display();
void TextEditor();
void cd(); void
listHistory();
void newCmd(char* cmd);
void ls(); void
disp_time();

typedef struct{ char
history[30];
}list;

list history_list[100];

void basic_loop(void){
char input[50] = ""; char
file_name[30];

char process_name[30];

while(strcmp("exit",input)!=0)
{
printf("\n>>"); scanf("%s",input);
```

```
newCmd(input);

if(strcmp("help",input) == 0)
{ printf("Shell Features are
\n");

printf("1)  addition,subtraction,multiplication \n\t,power & division can be
performed for 2 integers with no spacing.\n"); printf("2) editor -To use
text editor\n"); printf("3) cd   - Prints the current directory.\n"); printf("4)
history - History Command\n"); printf("5) ls   - Shows all files in current
directory!\n"); printf("6) time  - Displays current date & time!\n");
printf("7) gcc   - Runs a compiler to compile C/C++ files in the current
directory!\n"); printf("8) rm  - Removes a file.\n"); printf("9) start   -
Shows the output of the compiled file!\n"); printf("10) run  - to run
application!\n"); printf("11) exit  - To exit!");
}

else if(strcmp("exit",input)==0)
{     printf("Goodbye
!");
    exit(0);
}

else if(strcmp("ls",input)==0)
{
    ls();
}

else if(strcmp("gcc",input) == 0)
```

```
{     char
system_call[100];
scanf("%s",file_name);

   sprintf(system_call, "%s %s", input, file_name);
int error = system(system_call);     if(error == 0)
   {
       printf("Compilation successfull!\n");
   }

}

else if(strcmp("start",input) == 0)
{     char
system_call[100];
scanf("%s",file_name);

   sprintf(system_call, "%s %s", input, file_name);
int error = system(system_call);     if(error == 0)

   {
       printf("Compilation successfull!\n");
   }

} else if(strcmp("run",input) ==
0)
{     char system_call[100];
scanf("%s",process_name);
sprintf(system_call,"%s",process_name);
```

```
printf("Wait\n");     system(system_call);
printf("Application closed\n");
}

else if(strcmp("time",input) == 0)
    {
        disp_time();
    }

else if(isdigit(input[0]))
{ float first,
second;

if(strchr(input,'+')) {
sscanf(input,"%f+%f",&first,&second); printf("%f",first+second);

}

else if(strchr(input,'-'))
    {
    sscanf(input,"%f-%f",&first,&second);     printf("%f",first-
second);

}

else if(strchr(input,'*'))
    {
        sscanf(input,"%f*%f",&first,&second);
printf("%f",first*second);
}
```

```
else if(strchr(input,'/'))
    {
        sscanf(input,"%f/%f",&first,&second);
printf("%f",first/second);

}

else if(strchr(input,'^'))
    {
        sscanf(input,"%f^%f",&first,&second);
printf("%f",pow(first,second));

}

} else
if(strcmp("rm",input)==0)
{
            Delete();
} else if(strcmp("editor",input) ==
0)
    {
        printf("\nWelcome!");
        TextEditor();
}

else if(strcmp("cd",input)==0)
{ cd();
}
```

```
else if(strcmp("history",input)==0)
{ listHistory();
}

else { printf("Unknown command! Please enter 'help' to get the list of
commands");
}
}

}

void TextEditor()
{ do { printf("\n\nSelect
functions:");

printf("\n1.Create a file.\n2.Display the contents.\n3.Append in a file.\n4.Delete a
file.\n5.Exit the editor.\n"); printf("Enter your choice: "); scanf("%d",&ec);
switch(ec) { case 1: Create(); break;

case 2: Display(); break;

case 3: Append(); break;

case 4: Delete(); break;

case 5: basic_loop(); break;
} }
while(1);
```

```
} void
Create()
{
fp1=fopen("temp.txt","w");

printf("\nEnter the text and press '.' to save!\n\n");
while(1) { c=getchar();


fputc(c,fp1);
if(c == '.') { fclose(fp1);
printf("\nEnter then new filename:
"); scanf("%s",fn);
fp1=fopen("temp.txt","r");
fp2=fopen(fn,"w"); while(!feof(fp1))
{


c=getc(fp1);
putc(c,fp2);
}
fclose(fp2);
break; }
}


}


void Display()
{
```

```
printf("\nEnter file name: ");
scanf("%s",fn);
fp1=fopen(fn,"r");
if(fp1==NULL)
{ printf("\n\tFile not
found!"); goto end1; }
while(!feof(fp1)) {
c=getc(fp1);

printf("%c",c); } end1: fclose(fp1);
printf("\n\nPress any key to
continue!");
}

void Delete()
{

        printf("\nEnter file name: ");
        scanf("%s",fn);
  fp1=fopen(fn,"r");
if(fp1==NULL)
  {
     printf("\nFile not found!");
              goto end2;
        }
        fclose(fp1);
        if(remove(fn)==0)
  {
              printf("\n\nFile deleted successfully!");
```

```
goto end2;
    }
    else printf("\nError!\n");
    end2:
    printf("\n\nPress any key to continue!");
}

void Append()

{

printf("\nEnter the file name: ");
scanf("%s",fn);
fp1=fopen(fn,"r");
if(fp1==NULL)
{ printf("\nFile not
found!"); goto end3; }
while(!feof(fp1))
{
c=getc(fp1); printf("%c",c); } fclose(fp1);
printf("\nType the text and press 'Ctrl+S' to append.\n");
fp1=fopen(fn,"a"); while(1)
{

if(c==19)
goto end3;

if(c==13)
{ c='\n';
```

```
printf("\n\t");
fputc(c,fp1);
} else
   {
        printf("%c",c);
fputc(c,fp1);
   }
} end3:
fclose(fp1);
}


void cd()
{ char
*buf;


buf=(char *)malloc(100*sizeof(char));
getcwd(buf,100); printf("The current
directory is : %s",buf);


}


void ls() {
DIR *d;


struct dirent *dir; d = opendir(".");
printf("\nThe files in the directory are: \n");
if (d) { while ((dir = readdir(d)) !=
NULL)
```

```
{     printf("%s\n", dir-
>d_name);
} closedir(d);
}
}

void newCmd(char* cmd)
{
strcpy(history_list[last].history,cmd);
last++;
}



void listHistory() { printf("\nThe commands
last entered are:\n");
int z;
for (z = 0;z < last;z++)
{ printf("%s\n",history_list[z].history);
}
}

void disp_time() {     time_t t;     time(&t);
printf("Today's date and time : %s",ctime(&t));
}

int main()
{
```

```
printf("Welcome user to this basic UNIX shell! This has been implemented using
C code.\n"); printf("\n Made by : \n");
printf("\n DEBPROTIM CHAKRABARTI (17BCE0408) \n");
printf("\n AAYUSH SAHAY (17BCE2141) \n"); printf("\n
ANIKET GUPTA (17BCE2018) \n");
printf("\n\n"); printf("-----command
prompt-----\n");
printf("\n");

basic_loop();
}
```

CONCLUSION

We can presume that executing shell utilizing C language was a decent ordeal as we became acquainted with to does each order work and how to actualize them on any framework which does not keep running on Linux OS utilizing C language.

By deduction, the different favorable circumstances and impediments of shell usage is investigated and comprehended.

Points of interest include:

- To computerize the much of the time performed tasks
- To run arrangement of directions as a solitary order
- Easy to utilize
- Can be utilized for non-Linux based machines
- Repetitive errands can be effortlessly performed

Drawbacks include:

- Slightly moderate execution speed
- A new process propelled for pretty much every shell direction executed
- Difficult investigating
- Certain process confinements like complex activities that are hard to perform
- Difficult to actualize applications that are not cross stage perfect
- Difficult to actualize applications that bargain with security of framework.