Class 9 Story

Welcome to class 9 . In this class we continue our exploration of how processes, their life cycle, and how they communicate. In order to learn about processes, we are studying the shell - the unix tool designed for users to manage processes.

Last time, we looked at the most popular use of the shell: running programs. We saw how the three system calls fork, exec, and wait are used to run programs by the shell.

This time, we shall look at the shell as a programming language. The shell is much more than a tool that runs programs. It allows you to combine separate programs into one program, the same way that C allows you to combine separate functions into one program. In doing so, it advances Unix from an operating system into a programming environment. This is an important conceptual viewpoint. Once the inventors of Unix realized they could treat individual C programs as functions in larger programs, the operating system continued to evolve into one large programming environment in which the ability to run programs, coordinate their actions, and transfer data from one to another became essential parts of the system.

Tonight we shall look at three main ideas of how the shell creates a programming environment: local variables, global variables, and control flow in shell scripts.

First, Review of Running a Program: fork, exec, wait

First, though, let us review the way a shell runs a program and focus on the details of the connection between the death of a child process and the parent's call to wait().

fork() - create a new process

> The flow is pretty simple: the shell uses fork to create a new process. That new process contains an exact copy of the program that called fork. It determines, by examining the return value from fork() that it is the child process.

exec() - child runs a new program

> The child process then exec()s the new program.

wait() - parent waits for child to end

> Meanwhile, the parent process calls wait(). Wait has two effects. It suspends the calling program until the child ends, and it receives information about the termination of the child process.

an example:

> Here is a sample program that shows how this transfer of termination status works. forkquiz1.c creates a new process. The child process asks the user for input and passes that value, via exit() to the parent. The parent process calls wait() and is suspended until the child exit()s. Let's take it for a spin.
>
> Notice that the parent receives the exit() value from the child in the top 8 bits of the integer the address of which is passed to wait(). Why is the return value stuffed in the top 8 bits? What goes in the low 8 bits?
>
> Well, there is more than one way for a process to end. exit() is the nice way. But, a program could be killed by a signal. For example, if a user presses Ctrl-C or Ctrl- the process will be killed. If an alarm signal arrives and the process has not arranged to catch it, the process dies. There are lots of signals that can kill a process. Let's run our program and send it SIGINT.

conclusion:

A child process can end by calling exit() or by receiving a signal. The parent learns of the cause of death by examining the value returns via wait. If the child called exit(), the exit value appears in the ‘high’ 8 bits. If the child died from

Any questions?

The shell as programming Language: variables

Let us look at the first of two aspects of the shell as a programming language: variables.

Variables: strings stored under names

Assigning variables

In the Bourne shell, we can create variables easily using the assignment operator:

```
$ day=Monday
```

the syntax is simple: varname=value, no spaces!

```
$ NAME=bruce
$ Food=COOKIE
$ year=2000
```

All variables are string variables. The variable ‘year’ for example, contains the string 2000, not the integer 2000.

Retrieving Variables

How do you retrieve the values of these variables? In the shell, the ‘value of’ operator is $. Therefore, we can write:

```
$ echo my name is $NAME
$ echo I like to eat a $Food
$ cal $year
```

You can even use the retrieve operator in an assignment statement:

```
$ now=$year
$ X=3
$ Y=$X+$X
$ echo $Y + $now
```

Inputting Variables

Another way to set a value for a variable is via user input.

```
$ read dogname
fido
$ echo $dogname
```

Here is an example of using variables in a shell script to create a simple rolodex style program:

The file is a flat file with things like

```
fred	111-2222
wilma		222-3333
barney		234-1000
betty	999-4000
```

the command grep fred phonebook will extract all lines that contain that string.

Here is a shell script that uses variables

```
#!/bin/sh
# a rolodex thing

DATAFILE=phonebook
echo "Search phonebook for what name? "
read name
grep $name $DATAFILE
```

This script shows setting a variable using the assignment operator, and using the read command. It shows how one can use the values of those variables in a command. The shell replaces $name and $DATAFILE with the strings stored in those variables and then runs the command. The output is the set of lines that contain the specified search string.

This script is not really satisfactory. What if we search for a string that does not appear in the file. IT might be nice to say something like "no matching entries".

How do we do that? The answer is that the shell provides control flow in the form of if..then statements, while loops and the rest.

Here is a fancier version

```
#!/bin/sh
# rolodex 2

DATAFILE=phonebook.txt
echo "Search phonebook for what name? "
read name
if grep $name $DATAFILE
then
	echo "============"
	echo "are entries matching $name"
else
	echo "No matching entries for $name"
fi
```

The new feature of this script is the use of the if then construction. The shell does the following:

a) runs the command after the word if
b) examines the exit() code from the command
c) if the command exits(0) then the shell
   executes the statements in the then block
d) otherwise it executes the else block

The two questions that lie behind this are:

1) how does the shell get the exit code from grep?
2) What exit code does grep return?

The answers are simple:

1) it uses wait()
2) read the manual on grep.

The general rule for unix tools is:

* if the command ‘succeeds’ exit(0),
* if not, exit() with a non-zero code and
  document the meaning of the codes

Examples: grep, diff, sort, who, du, stty

Therefore, you can use ANY command after the if, and it will run the command, and will do the then part on success and the else part on error. This is purely a convention that unix tools need to follow.

Designing a shell requires creating systems for setting and retrieving shell variables and for managing control flow.