UNIX Shell Programming (15CS35)
Solution
Internal Assessment- III
November – 2016

1) Explain the mechanism of process creation. Also give the details about process states and zombies.

There are three distinct phases in the creation of a process and uses three important system calls viz., fork, exec, and wait. The three phases are discussed below:

Fork: A process in UNIX is created with the fork system call, which creates a copy of the process that invokes it. The process image is identical to that of the calling process, except for a few parameters like the PID. The child gets a new PID.

Exec: The forked child overwrites its own image with the code and data of the new program. This mechanism is called exec, and the child process is said to exec a new program, using one of the family of exec system calls. The PID and PPID of the exec‟d process remain unchanged.

Wait: The parent then executes the wait system call to wait for the child to complete. It picks up the exit status of the child and continues with its other functions. Note that a parent need not decide to wait for the child to terminate.

To get a better idea of this, let us explain with an example. When you enter ls to look at the contents of your current working directory, UNIX does a series of things to create an environment for ls and the run it: The shell has UNIX perform a fork. This creates a new process that the shell will use to run the ls program.

The shell has UNIX perform an exec of the ls program. This replaces the shell program and data with the program and data for ls and then starts running that new program. The ls program is loaded into the new process context, replacing the text and data of the shell. The ls program performs its task, listing the contents of the current directory. In the meanwhile, the shell executes wait system call for ls to complete. When a process is forked, the child has a different PID and PPID from its parent

Process States:

At any instance of time, a process is in a particular state. A process after creation is in the runnable state. Once it starts running, it is in the running state. When a process requests for a resource (like disk I/O), it may have to wait. The process is said to be in waiting or sleeping state. A process can also be suspended by pressing a key (usually Ctrl-z). When a process terminates, the kernel performs clean-up, assigns any children of the exiting process to be adopted by init, and sends the death of a child signal to the parent process, and converts the process into the zombie state. A process in zombie state is not alive; it does not use any resources nor does any work. But it is not allowed to die until the exit is acknowledged by the parent process.

It is possible for the parent itself to die before the child dies. In such case, the child becomes an orphan and the kernel makes init the parent of the orphan. When this adopted child dies, init waits for its death.

2. Explain the following commands with example

i) Running jobs in the background

& : No Logging Out : The & is the shell's operator used to run a process in the background. The parent in this case doesn't wait for the child's death. Just terminate the command line with an & ,the command will run in the background:

$ sort –o emp.lst emp.lst & 550

The job's PID The shell immediately returns a number –the PID of the invoked command(550) . The prompt is returned and the shell is ready to accept another command even though the previous command has not been terminated yet. The shell, however, remains the parent of the background process Using an & you can run as many jobs in the background as the system load permits. Background execution of a job is a useful feature that you should utilize to transfer time –consuming or low- priority jobs to the background, and run the important ones in the foreground.

Nohup : Logout safely: Background jobs ceases to run, however, when a user logs out (the C shell and Bash excepted). That happens because her shell is killed. And when the parent is killed, its children are also normally killed (subject to certain conditions). The UNIX system permits a variation in this default behaviour . The nohup (or no hangup) command, when prefixed to a command, permits execution of the process even after the user has logged out. You must use the & with it as well:

$ nohup sort emp.lst &

586

Sending output to nohup.out

ii) Execute later

UNIX provides facilities to schedule a job to run at a specified time of day. If the system load varies greatly throughout the day, it makes sense to schedule less important jobs at a time when the system load is low. The at and batch commands make such job scheduling possible

at :To schedule one or more commands for a specified time, use the at command. With this command, you can specify a time, a date, or both.

For example, $ at 14:23 Friday

at> lp /usr/sales/reports/*

at> echo "Files printed, Boss!" | mail -s"Job done" boss

[Ctrl-d]

commands will be executed using /usr/bin/bash job 1041198880.a at Fri Oct 12 14:23:00 2007

The above job prints all files in the directory /usr/sales/reports and sends a user named boss some mail announcing that the print job was done. All at jobs go into a queue known as at queue.at shows the job number, the date and time of scheduled execution. This job

number is derived from the number of seconds elapsed since the Epoch. A user should remember this job number to control the job.

Batch: The batch command lets the operating system decide an appropriate time to run a process. When you schedule a job with batch, UNIX starts and works on the process whenever the system load isn't too great.
To sort a collection of files, print the results, and notify the user named boss that the job is done, enter the following commands:
$ batch
sort /usr/sales/reports/* | lp
echo "Files printed, Boss!" | mailx -s"Job done" boss
The system returns the following response:
job 7789001234.b at Fri Sep 7 11:43:09 2007

The date and time listed are the date and time you pressed to complete the batch command. When the job is complete, check your mail; anything that the commands normally display is mailed to you. Note that any job scheduled with batch command goes into a special at queue

3. a) Using **command line arguments** , write a perl program to find whether a given year is leap year

```
#!/usr/bin/perl
#Initializing a flag called isLeapYear
#A value of 0 indicates that it is not a leap year
#A value of 1 indicates that it is a leap yer
$isLeapYear = 0;
#Accept the user's input
print("\n Enter a year: ");
$year = <STDIN>;
#Remove the newline character at the end of the input
chop($year);
if ($year % 100 == 0)
{
        #If the year is like 2000, 2100, 2300 then
         if ($year % 400 == 0)
        {
        #If the year is divisible by 100 and by 400 then it's a leap year
        #Set the flag's value to 1
        $isLeapYear = 1;
         }
}
elsif ($year % 4 == 0)
{
#If the year is not divisible by 100 but divisible by 4 then it's a leap year
#Set the flag's value to 1
```

```
        $isLeapYear = 1;
}
if ($isLeapYear == 1)
{
        #A value of 1 indicates that it is a leap yer
         printf("\n\n $year is a leap year.");
}
else
{
        #A value of 0 indicates that it is not a leap year
         printf("\n\n $year is not a leap year.");
}
```

b) Write a perl program to convert given decimal number to binary equivalent

```
#!/usr/bin/perl
 foreach $num (@ARGV)
{
$temp = $num; until ($num == 0)
{
$bit = $num % 2;
unshift(@bit_arr, $bit);
$num = int($num/2);
}
$binary_num = join(“”,@bit_arr);
print (“Binary form of $temp is $binary_num\n”);
splice(@bit_arr, 0, $#bit_arr+1);
}
```

4) Explain the following in PERL with examples

i) for each loop

The syntax for foreach loop is:

```
foreach $var (@arr)
{
Statements
}
```

Example :

```
#!/usr/bin/perl
@list = ("This", "is", "a", "list", "of", "words");
print("Here are the words in the list: \n");
foreach $temp (@list)
{
```

```
        print("$temp ");
}
print("\n");
```

ii) **split: Splitting into a List or Array**

split breaks up a line or expression into fields. These fields are assigned either to variables or an array.

Syntax:

```
($var1, $var2, $var3 ……. ) = split(/sep/, str);

@arr = split(/sep/, str);
```

It splits the string **str** on the pattern **sep**. Here **sep** can be a regular expression or a literal string. **str** is optional, and if absent, **$_** is used as default. The fields resulting from the split are assigned to a set of variables , or to an array.

iii) **join: Joining a List**

It acts in an opposite manner to split. It combines all array elements in to a single string. It uses the delimiter as the first argument. The remaining arguments could be either an array name or a list of variables or strings to be joined.

```
$x = join(" ", "this", "is", "a", "sentence"); # $x becomes "this is a sentence".
@x = ("words","separated","by");
$y = join("::",@x,"colons");   #$y becomes "words::separated::by::colons".
```

To undo the effects of join(), call the function split():

```
$y = "words::separated::by::colons";
@x = split(/::/, $y);
```

6. Explain the following commands

nice:The nice command is used to control background process dispatch priority
nice values are system dependent and typically range from 1 to 19
Example: $ nice wc –l hugefile.txt OR $ nice wc –l hugefile.txt &
We can specify the nice value explicitly with –n number option where number is an offset to the default. If the –n number argument is present, the priority is incremented by that amount up to a limit of 20. Example: $ nice –n 5 wc –l hugefile.txt &

kill: Issuing the kill command sends a signal to a process. The default signal is SIGTERM signal (15). UNIX programs can send or receive more than 20 signals, each of which is represented by a number.

To use kill, use either of these forms:
kill PID(s) OR kill –s NUMBER PID(s)
To kill a process whose PID is 123 use,
$ kill 123
To kill several processes whose PIDs are 123, 342, and 73
use, $ kill 123 342 73

cron: cron program is a daemon which is responsible for running repetitive tasks on a regular schedule. It is a perfect tool for running system administration tasks such as backup and system logfile maintenance. It can also be useful for ordinary users to schedule regular tasks including calendar reminders and report generation .
The cron system is managed by the cron daemon. It gets information about which programs and when they should run from the system's and users' crontab entries. The crontab files are stored in the file /var/spool/cron/crontabs/ where is the login-id of the user. Only the root user has access to the system crontabs, while each user should only have access to his own crontabs.
A typical entry in crontab file A typical entry in the crontab file of a user will have the following format. minute hour day-of-month month-of-year day-of-week command where, Time-Field Options are as follows:

Minute	00 through 59 Number of minutes after the hour
Hour	00 through 23 (midnight is 00)
day-of-month	01 through 31
month-of-year	01 through 12
day-of-week	01 through 07 (Monday is 01, Sunday is 07)

Example: 00-10 17 * 3.6.9.12 5 find / -newer .last_time –print >backuplist
In the above entry, the find command will be executed every minute in the first 10 minutes after 5 p.m. every Friday of the months March, June, September and December of every year.

find : locating files It recursively examines a directory tree to look for files matching some criteria, and then takes some action on the selected files. It has a difficult command line, and if you have ever wondered why UNIX is hated by many, then you should look up the cryptic find documentation. How ever, find is easily tamed if you break up its arguments into three components:
find path_list selecton_criteria action
where,
Recursively examines all files specified in path_list
It then matches each file for one or more selection-criteria
It takes some action on those selected files
The path_list comprises one or more subdirectories separated by white space. There can also be a host of selection_criteria that you use to match a file, and multiple actions to

dispose of the file. This makes the command difficult to use initially, but it is a program that every user must master since it lets him make file selection under practically any condition.

6. Explain the following string handling functions in PERL with examples

i) **length:** determines the length of its argument.

ii) **substr(str,m,n)** : extracts a substring from a string str, m represents the starting point of extraction and n indicates the number of characters to be extracted.

iii) **splice** : The splice function can do everything that shift, pop, unshift and push can do. It uses upto four arguments to add or remove elements at any location in the array. The second argument is the offset from where the insertion or removal should begin. The third argument represents the number of elements to be removed. If it is 0, elements have to be added. The new replaced list is specified by the fourth argument (if present). splice(@list, 5, 0, 6..8); # Adds at 6th location, list becomes 1 2 3 4 5 6 7 8 9 splice(@list, 0, 2); # Removes from beginning, list becomes 3 4 5 6 7 8 9

iv) **push**: The push function add elements to an array.
For example if the elements of array list are 1 2 3 4 5 then
push(@list,9); # Pushes 9 at end –- 1 2 3 4 5 9

v) **pop**: For deleting elements at the end of an array, perl uses the pop function.
Example: @list = (3..5, 9);
pop(@list); # Removes last element, becomes 4 5