BASICS OF SHELL PROGRAMS

How to Run a Shell Script

- Edit and save your program using any one editor
- Add execute permission by *chmod* command
- Run your program by simply typing the name of your program
- If you get an error like ‘command not found’, then understand that your present working directory is not included in the PATH environment variable. Hence run your program using either one of the following way.
  - ./program-name
  - sh program-name
  - Include your path in the PATH environment variable (PATH=$PATH:/root/abc if /root/abc is your path) and run.

Variables

- A shell variable begins with alphabetic or underscore "_" character, and is followed by zero or more alphabetic or underscore characters
- variable=value
- Must not be space before or after "="
- Example: my_bin=/usr/local/bin
- Shell does not have the concept of data types
- If a valid variable name followed the *$*, then the shell takes this as an indication that the value stored inside that variable is to be substituted of that point

- Variables can be assigned to:
  - numbers
  - alphanumeric chracters/strings
  - other variables
  - utilities or programs
  - null values
  - file names

- Null values can be assigned to variables as follows:
  - variable=
  - variable="" (double quote)
  - variable=''(single quote)
- variable=" " with a space between the quotes is not a null variable (space is a character)

QUOTATIONS

There are 4 types of quote characters. They are single quote, double quote, back slash and back quote.

1. Single Quote

- When the shell sees the first single quote, it ignores any otherwise special characters that follow until it sees the closing quote
- Single quotes preserves all white spaces
- Quotes are needed when assigning values containing white space or special characters to shell variables
- Shell does file substitution after variable name substitution even in between single quotes

2. Double Quote

- Three characters are not ignored inside double quotes:
  - Dollar sign
  - Back quotes
  - Backslashes
- File names substitution (*,?) is not done inside double quotes
- Double quotes can be used to hide single quotes from the shell, and vice versa

3. Backslash

- It is used in two ways
  - (i) to remove the special meaning of a character.
  - (ii) To type lengthy command in more than one line (ie; line continuation)

Ex. $ echo \$ *will display $*

$ lines=one\ *it is equivalent to lines = onetwo*
> two

4. Back quote

- Used as command substitution
- Do not leave any space around the assignment operator (=)
- It can be replaced by $(...) construct

Ex. $a=`date` *will assign the output of the command date to variable a*
$a=$(date) *it is equivalent to the above*

ARITHMETIC EXPRESSIONS

- Arithmetic expressions can be written either of the following ways

  (i) Using (( ))

  Ex. ((i+5))

  (ii) Using *expr* command

  Ex. expr $i + 5

- Values of a variable can referred without prefix $ inside ((...)) and the same is the must if we use *expr*.

PASSING ARGUMENTS

- Whenever you execute a shell program, the shell automatically stores the first argument in the special shell variable #1, the second argument in the variable #2, and so on. [$1,$2..$9]
- These special variables are known as *positional parameters*

Shell variable	Meaning

$0	Name of the script
$1, $2, $3, … …,$9	First, Second arguments and so on
${10}, ${11}, …	Argument from tenth onwards
$#	Number of arguments
$*	List of all the arguments
$@	List of all the arguments
$?	Exit status of the last command executed
$$	Process-Id of the current process

Shift Command

- Allows you to left shift your positional parameters. If you execute the command

  shift

  the value of $2 will be assigned to $1, value of $3 will be assigned to $2, and so on.
- The value of $1 will be lost.
- Value of $# will be decremented automatically by one when shift is executed.
- *shift 3* shift 3 parameters left

EXIT STATUS

- Whenever any program completes execution, it returns an *exit* status back to the system
- status is a number
- status 0: program executed successfully
- status of nonzero: program executed unsuccessfully
- The shell variable $? Is automatically set by the shell to the exit status of the last command executed

Ex.

```
$ cp file1 file2
$ echo $?
0                                   ← successful execution
```

```
$ cp file3 file1
cp: cannot access file3
$ echo $?
2                                  ← unsuccessful execution
```

IF STATEMENT

Syntax :

```
if command
then
    command
    command
    ...
else
    command
    command
    ...
fi
```

- If the exit status of the command is zero then the *then* part gets executed, otherwise *else* part will be executed.
- *else* part is optional
- The *command* in the *if* statement should be written in either one of the following ways

  (i) using *test* command

  test expression

  ex. if test “$name” = Judith

  (i) using […] construct

  ex. if [“$name” = Judith]

OPERATORS

1. String Operators

Operator	**Returns TRUE (zero exit status) if**
String	*string* is not null
-n string	*string* is not null (and *string* must be seen by *test*)
-z string	*string* is null (and *string* must be seen by *test*)
string1 = string2	*string1* is identical to *string2*
string1 ! = string2	*string1* is *not* identical to *string2*

2. Integer Operators

Operator	**Returns TRUE (zero exit status) if**
int1 –eq int2	*int1* is equal to *int2*
int1 –ge int2	*int1* is greater than or equal to *int2*
int1 –gt int2	*int1* is greater than *int2*
int1 –le int2	*int1* is less than or equal *int2*
int1 –lt int2	*int1* is less than *int2*
int1 –ne int2	*int1* is not equal to *int2*

3. Logical Operators

Operator	**Returns TRUE (zero exit status) if**
! expr	*expr is FALSE; otherwise returns TRUE*
expr1 –a expr2	*expr1 is TRUE and int2 is TRUE*
expr1 –o expr2	*expr1 is TRUE or int2 is TRUE*

4. File Operators

Operator	**Returns TRUE (zero exit status) if**
-b file	*file* is a block special file
-c file	*file* is a character special file
-d file	*file* is a directory

-f file	*file* is an ordinary
-g file	*file* has its set group id (SGID) bit set
-k file	*file* has its sticky bit set
-p file	*file* is a named pipe
-r file	*file is readable by the process*
-s file	*file has nonzero length*
-t file	*file is open file descriptor associated with a terminal (1 is default)*
-u file	*file has its set user id (SUID) bit set*
-w file	*file is writable by the process*
-x file	*file is executable*

ELSE IF CONSTRUCT

```
if command$_1$
then
    command
    ...
else
    if command$_2$
    then
        command
        ...
    else
        command
        ...
    fi
fi
```

ELIF CONSTRUCT

```
if command$_l$
```

```
then
        command
        command
        ...
elif command₂
then
        command
        command
        ...
else
        command
        command
        ...
fi
```

EXIT COMMAND

- Exit immediately terminates execution of a shell program
- *exit n*
- *n*: the exit status that you want to be returned
- If *n* is not specified, then the exit status used is that of the last command executed before the exit

CASE STRUCTURE

```
case value in
pat₁)       command
            ...
            command;;
pat₂)       command
            ...
            Command;;
...
```

```
*)          command
            ...
            Command;;
esac
```

- The word *value* is compared against the values pat1, pat2, ... until a match is found
- When a match is found, the commands listed after the matching value, up to the double semicolons, are executed.
- If there is no match found, then the commands followed by option * will be executed.
- The shell lets you use **,?,[]* special characters with shell
- The symbol | has the effect of a logical *OR* when used between two patterns

  pat1 | pat2

The && and || constructs

- The shell has two special constructs that enable you to execute a command based on whether the preceding command succeeds or fails.

```
command1 && command2
```

- If the exit status of command$_1$ is zero (ie.; successfully executed), then command$_2$ will be executed, otherwise command$_2$ gets skipped.

```
command1 || command2
```

- The || construct works similarly, except that the second command gets executed only if the exit status of the first is nonzero.

FOR LOOP

The for command is used to execute a set of commands a specified number of times. Its basic format is as shown :

```
        for var in list-of-values
        do
                command
                command
                ...
        done
```

- First value will be assigned to the variable *var* and body of the loop gets executed, then the second value will be assigned to *var* for the second iteration, and so on.

- There are many ways to pass values to the *var* of for loop.
  (i) List the values that are separated by spaces

      ex.   for i in 2 5 8 4 6

  (ii) Substitution characters may be used

      ex1. for i in *   → name of the files in the present working dir
      ex2. for i in a?   → name of the file starts with *a* and followed by any one character
      ex3. for i in f[1-4]   → name of the files f1, f2, f3, f4

  (iii) Content of a file can be listed as values

      ex.   for i in `cat filename`

  (iv) Command line arguments may be passed

      ex.   for i in $*   → values $1, $2,$3, … …
      ex.   for i in $@   → values "$1", "$2","$3", … …

- If we write for loop without the values(*for i)*, then its is equivalent to *(for i in $@)*

While loop

The format of this command is

```
while command_t
do
    command
    command
    ...
done
```

It works similar to the while loop in c, c++ languages

Until loop

The while command continues execution as long as the command listed after the while returns a zero exit status. The until command is similar to the while, only it continues execution as long as the command that follows the until returns a nonzero exit status. As soon as a zero exit status is returned, the loop is terminated. The general format of the until:

```
until commandt
do
    command
    command
    ...
done
```

- Like the while, the commands between the do and done might never be executed if commandt returns a zero exit status the first time it's executed.
- The until command is useful for writing programs that wait for a particular event to occur.

read command

Syntax : read variables

- The shell reads a line from standard input and assigns the first word to the first variable, the second word to the second variable, and so on.
- If there are more words on the line than there are variables listed, the excess words get assigned to the last variable. So for example, the command

1. Program to find the sum of first ‘n’ natural numbers

```
echo Enter value for n
read n
sum=0
i=1
while [ $i –le $n ]
do
        sum=$((sum+i))
        i=$((i+1))
done
echo Sum is $sum
```

2. Program to list only the name of sub directories in the present working directory.

```
for i in *
do
 if [ -d $i ]
 then
  echo $i
 fi
done
```

3. Program to check all the files in the present working directory for a pattern (passed through command line) and display the name of the file followed by a message stating that the pattern is available or not available.

```
echo Enter pattern to search
read pat
for i in *
do
 if [ -f $i ]
 then
  grep $pat $i
  if [ $? -eq 0 ]
then
    echo $i :Found
  else
    echo $i :NOT Found
  fi
 fi
done
```

READER WRITER PROBLEM

What is The Readers-Writers Problem?

The Readers-Writers Problem is a classic synchronization issue in operating systems that involves managing access to shared data by multiple threads or processes. The problem addresses the scenario where:

- **Readers**: Multiple readers can access the shared data simultaneously without causing any issues because they are only reading and not modifying the data.
- **Writers**: Only one writer can access the shared data at a time to ensure data integrity, as writers modify the data, and concurrent modifications could lead to data corruption or inconsistencies.

Challenges of the Reader-Writer Problem

The challenge now becomes how to create a synchronization scheme such that the following is supported:

- **Multiple Readers**: A number of readers may access simultaneously if no writer is presently writing.
- **Exclusion for Writers**: If one writer is writing, no other reader or writer may access the common resource.

Solution of the Reader-Writer Problem

There are two fundamental solutions to the Readers-Writers problem:

- **Readers Preference:** In this solution, readers are given preference over writers. That means that till readers are reading, writers will have to wait. The Writers can access the resource only when no reader is accessing it.
- **Writer's Preference:** Preference is given to the writers. It simply means that, after arrival, the writers can go ahead with their operations; though perhaps there are readers currently accessing the resource.

Focus on the solution Readers Preference in this paper. The purpose of the Readers Preference solution is to give a higher priority to the readers to decrease the waiting time of the readers and to make the access of resource more effective for readers.

Problem Parameters

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read

Here, we use one **mutex** m and a **semaphore** w. An integer variable read_count is used to maintain the number of readers currently accessing the resource. The variable read_count is initialized to 0. A value of 1 is given initially to m and w.

Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the read_count variable.
The code for the **writer** process looks like this:

```
while(TRUE)
{
   wait(w);
   /* perform the write operation */

   signal(w);
}
```

The code for the reader process looks like this:

```
while(TRUE)
{
    //acquire lock
    wait(m);
    read_count++;
    if(read_count == 1)
        wait(w);

    //release lock
    signal(m);

    /* perform the reading operation */

    // acquire lock
    wait(m);
    read_count--;
    if(read_count == 0)
        signal(w);

    // release lock
    signal(m);
}
```

Code Explanation

- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.

- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.
- When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

Bounded Buffer Problem

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

Now let's see the solutions to the above problem.

One solution of this problem is to use semaphores. The semaphores which will be used here are:

- m, a **binary semaphore** which is used to acquire and release the lock.
- empty, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- full, a **counting semaphore** whose initial value is 0.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

The Producer Operation

The pseudocode of the producer function looks like this:

```
do
{
   // wait until empty > 0 and then decrement 'empty'
   wait(empty);
   // acquire lock
   wait(mutex);

   /* perform the insert operation in a slot */

   // release lock
   signal(mutex);
   // increment 'full'
   signal(full);
}
while(TRUE)
```

- Looking at the above code for a producer, we can see that a producer first waits until there is atleast one empty slot.

- Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

The Consumer Operation

The pseudocode for the consumer function looks like this:

```
do
{
   // wait until full > 0 and then decrement 'full'
   wait(full);
   // acquire the lock
   wait(mutex);

   /* perform the remove operation in a slot */

   // release the lock
   signal(mutex);
   // increment 'empty'
   signal(empty);
}
while(TRUE);
```

- The consumer waits until there is atleast one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.

- Then, the consumer releases the lock.
- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

Monitors

Monitors are a higher-level synchronization construct that simplifies process synchronization by providing a high-level abstraction for data access and synchronization. Monitors are implemented as programming language constructs, typically in object-oriented languages, and provide mutual exclusion, condition variables, and data encapsulation in a single construct.

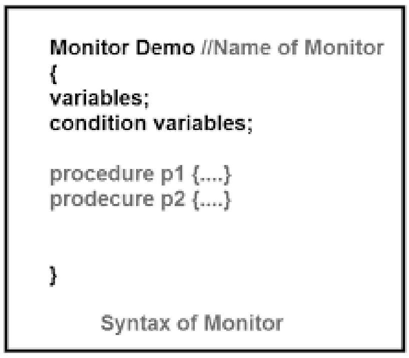
1. A monitor is essentially a module that encapsulates a shared resource and provides access to that resource through a set of procedures. The procedures provided by a monitor ensure that only one process can access the shared resource at any given time, and that processes waiting for the resource are suspended until it becomes available.
2. Monitors are used to simplify the implementation of concurrent programs by providing a higher-level abstraction that hides the details of synchronization. Monitors provide a structured way of sharing data and synchronization information, and eliminate the need for complex synchronization primitives such as semaphores and locks.
3. The key advantage of using monitors for process synchronization is that they provide a simple, high-level abstraction that can be used to implement complex concurrent systems. Monitors also ensure that synchronization is encapsulated within the module, making it easier to reason about the correctness of the system.

However, monitors have some limitations. For example, they can be less efficient than lower-level synchronization primitives such as semaphores and locks, as they may involve additional overhead due to their higher-level abstraction. Additionally, monitors may not be suitable for all types of synchronization problems, and in some cases, lower-level primitives may be required for optimal performance.

The monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.
2. The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.
3. Only one process at a time can execute code inside monitors.

Syntax:



```
Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {....}
prodecure p2 {....}


}
```

Syntax of Monitor

Condition Variables: Two different operations are performed on the condition variables of the monitor. They are wait() and signal()

Let say we have 2 condition variables **condition x, y; // Declaring variable Wait operation** x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable. **Note:** Each condition variable has its unique block queue. **Signal operation** x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

```
If (x block queue empty)
  // Ignore signal
else
```

```
        // Resume a process from block queue.
```

Advantages of Monitor: Monitors have the advantage of making parallel programming easier and less error prone than using techniques such as semaphore.

Disadvantages of Monitor: Monitors have to be implemented as part of the programming language . The compiler must generate code for them. This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes. Some languages that do support monitors are Java,C#,Visual Basic,Ada and concurrent Euclid.

Dining Philosopher Problem

The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.
Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Solutions to the above problem.
From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, stick[5], for each of the five chopsticks.
The code for each philosopher looks like:

```
while(TRUE)
{
    wait(stick[i]);
    /*
        mod is used because if i=5, next
        chopstick is 1 (dining table is circular)
    */
    wait(stick[(i+1) % 5]);

    /* eat */
    signal(stick[i]);

    signal(stick[(i+1) % 5]);
    /* think */
}
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever.

The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

Overlay Concepts

Exec() System Call – Overlay Calling process and run new Program

The exec() system call **replaces (overwrites) the current process with the new process image.** The PID of the new process remains the same however code, data, heap and stack of the process are replaced by the new program.

There are 6 system calls in the family of exec().All of these functions mentioned below are layered on top of execve(), and they differ from one another and from execve() only in the way in which the program name, argument list, and environment of the new program are specified.

Syntax

```
int execl(const char* path, const char* arg, …)
int execlp(const char* file, const char* arg, …)
int execle(const char* path, const char* arg, …, char* const envp[])
int execv(const char* path, const char* argv[])
int execvp(const char* file, const char* argv[])
int execvpe(const char* file, const char* argv[], char *const envp[])
```

- The names of the first five of above functions are of the form **exec*XY***.
- X is either l or v depending upon whether arguments are given in the list format (arg0, arg1, ..., NULL) or arguments are passed in an array (vector).
- Y is either absent or is either a p or an e. In case Y is p, the PATH environment variable is used to search for the program. If Y is e, then the environment passed in *envp* array is used.
- In case of execvpe, X is v and Y is e. The execvpe function is a GNU extension. It is named so as to differentiate it from the execve system call.

Example

```
#include <stdio.h>
#include<unistd.h>
int main()
```

```
{
  printf("Transfer to execlp function \n");
  execlp(“head”, “head”,”-2”,”f1”,NULL);
  printf("This line will not execute \n”);
  return 0;
}
```

Conclusion

This week we learned about the Basics of Shell Programs, Shell Program Examples, Classical Problems of Synchronization like Reader writer Problem, Bounded Buffer Problem, Monitors, Dining Philosophers Problem and Overlay Concepts.