NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
ISLAMABAD

OPERATING SYSTEMS LAB SPRING 2019

# Lab Manual 01
# Introduction and Shell Scripting

## 1 OPERATING SYSTEMS

1. DEFINITION

   An Operating System, or OS, is low-level software that enables a user and higher-level application software to interact with a computer's hardware and the data and other programs stored on the computer.

2. BASIC FUNCTIONS

   (a) Program execution.
   (b) Interrupts.
   (c) Memory management.
   (d) Multitasking

3. GOALS

   (a) Execute user programs and make solving user problems easier.
   (b) Make the computer system convenient to use.

## 2 COMMAND LINE UTILITIES

When Linus Torvalds introduced Linux and for a long time thereafter, Linux did not have a graphical user interface: It ran on character-based terminals only. Command

line utilities are often faster, more powerful, or more complete than their GUI counterparts. Sometimes there is no GUI counterpart to a textbased utility. As it is a text-based interface so it consumes low RAM to run while GUI requires a high specification for running.

## 3 SHELL

A shell is simply a program which is used to start other programs. It takes the commands from the keyboard and gives them to the operating system to perform the particular task.
There are many different shells, but all derive several of their features from the Bourne shell, a standard shell developed at Bell Labs for early versions of Unix. Linux uses an enhanced version of the Bourne shell called bash or the "Bourne-again" shell. The bash shell is the default shell on most Linux distributions, and **/bin/sh** is normally a link to bash on a Linux system.

### THE SHELL WINDOW

After logging in, open a shell window (often referred to as a terminal). The easiest way to do so from a GUI like Ubuntu's Unity is to open a terminal application, which starts a shell inside a new window. The window display a prompt at the top that usually ends with a dollar sign $. If # is the last character it means you are running the command as root user.

## 4 BASIC COMMANDS

In this section we will have an insight of some basic commands. Different commands take multiple arguments and options (where option starts with a dash - sign).

1. **echo**
   The echo command prints its arguments to the standard output.

   $ echo Hello World

   Output *Hello World* on your terminal screen.

2. **ls**
   The ls command lists the contents of a directory. The default is the current directory. Use ls -l for a detailed (long) listing where -l is an option. Output includes the owner of the file (column 3), the group (column 4), the file size (column 5), and the modification date/time (between column 5 and the filename).

3. **cp**
   cp copies files. For example, to copy file1 to file2, enter this:

$ cp file1 file2

where file1 and file2 should be in current working directory.

$cp ⟨*Source*⟩ ⟨*destination*⟩

where source and destination are ful path from the root directory. To copy a number of files to a directory (folder) named dir, try this instead:

$ cp file1 ... fileN dir

4. **cat**
   It simply outputs the contents of one or more files. The general syntax of the cat command is as follows:

   $ cat file1 file2 ...

   where file1 and file2 should be in current working directory or otherwise write down full path starting from root. When you run this command, cat prints the contents of file1, file2, and any other files that you specify (denoted by ...), and then exits. The command is called cat because it performs concatenation when it prints the contents of more than one file.

5. **mv**
   It renames a file. For example, to rename file1 to file2, enter this:

   $ mv file1 file2

   You can also use mv to move a number of files to a different directory:

   $ mv file1 ... fileN dir

6. **rm**
   To delete (remove) a file, use rm. After a file is removed, it's gone from the system and generally cannot be undeleted.

   $ rm file

7. **touch**
   The touch command creates a file. If the file already exists, touch does not change it, but it does update the file's modification time stamp.

   $ touch file

8. **pwd**
   The pwd (print working directory) program simply outputs the name of the current working directory.

9. **date**
   Displays current time and date

10. **clear**
    Clears the terminal screen.

11. **exit**
    Exit the Shell

12. **head and tail**
    To quickly view a portion of a file or stream of data, use the head and tail commands. For example:

    head /etc/passwd

    shows the first 10 lines of the password file.

    tail /etc/passwd

    shows the last 10 lines. To change the number of lines to display, use the -n option, where n is the number of lines you want to see.

13. **sort**
    The sort command quickly puts the lines of a text file in alphanumeric order. The -r option reverses the order of the sort.

14. **grep**
    The grep command prints the lines from a file or input stream that match an expression. For example, to print the lines in the /etc/passwd file that contain the text root, enter this:

    $ grep root /etc/passwd

    The grep command is extraordinarily handy when operating on multiple files at once because it prints the filename in addition to the matching line. For example, if you want to check every file in */etc* that contains the word root, you could use this command:

    $ grep root /etc/*

    Two of the most important grep options are -i (for case-insensitive matches) and -v (which inverts the search, that is, prints all lines that don't match).

15. **mkdir**
    The mkdir command creates a new directory dir:

    $ mkdir dir

16. **rmdir**
    The rmdir command removes the directory dir:

    $ rmdir dir

    If dir isn't empty, this command fails.
    *rm -rf dir* is used to delete a directory and its contents where -r option specifies recursive delete to repeatedly delete everything inside dir, and -f forces the delete operation.

17. **cd**
    The *current working directory* is the directory that a process (such as shell) is currently in. The cd command changes the shell's current working directory:

    $ cd dir

    If you omit dir, the shell returns to your home directory, the directory you started in when you first logged in.

18. **less**
    The less command comes in handy when a file is really big or when a command's output is long and scrolls off the top of the screen. To page through a big file like */usr/share/dict/words*, use the command

    $ less /usr/share/dict/words

    When running less, you'll see the contents of the file one screenful at a time. Press the spacebar to go forward in the file and the b key to skip back one screenful. To quit, type q.

19. **man**
    Linux systems come with a wealth of documentation. For basic commands, the manual pages (or man pages) will tell you what you need to know. For example, to see the manual page for the ls command, run *man* as follows:

    $ man ls

## 5 Output Redirection

To send the output of command to a file instead of the terminal, use the > redirection character:

$ command > file

The shell creates file if it does not already exist. If file exists, the shell erases the original file first. You can append the output to the file instead of overwriting it with the » redirection syntax:

$ command » file

## 6 Navigating Directories

Unix has a directory hierarchy that starts at /, sometimes called the root directory. The directory separator is the slash (/).

### PathName

When you refer to a file or directory, you specify a path or pathname. There are two different ways of writing files path:

1. *Absolute Path*
   When a path starts with root, it is a full or absolute path.

   /home/cslab/Desktop/Lab01

2. *Relative Path*
   A path starts at current directory is called a relative path.

   ./Lab01

### Directory References

1. *Home Directory (~)*
2. *Root Directory(/)*
3. *Current Working Directory(.)*
   One dot (.) refers to the current directory; for example, if you're in */usr/lib*, the path dot(.) is still */usr/lib*, and *./X11* is */usr/lib/X11*.
4. *Parent Directory(..)*
   A path component identified by two dots (..) specifies the parent of a directory. For example, if you're working in */usr/lib*, the path .. would refer to */usr*. Similarly, *../bin* would refer to */usr/bin*.

## 7 SHELL SCRIPTING

If the shell offers the facility to read its commands from a text file ,then its syntax and features may be thought of as a programming language, and such files may be thought of as scripts. So a shell is actually two things:

1. An interface between user and OS.
2. A programming language.

Shell Script is series of commands written in plain text file the shell reads the commands from the file just as it would have typed them into a terminal. The basic advantage of shell scripting includes:

1. Shell script can take input from user, file and output them on screen.
2. Useful to create our own commands.
3. Save lots of time.
4. To automate some task of day today life.
5. System Administration part can be also automated.

**EXAMPLES**

```
# !/bin/bash
#First shell script
clear
echo "Hello World"
```

```
# !/bin/bash
#Script to print user information who currently login, current date and time
clear
echo "Hello $USER"
#echo induce next line at the end
#\c restrict output on same line and used with flag -e
echo -e "Today is \c"; date
#print the line count
echo "Number of lines, total words and characters: "; ls -l | wc
echo "Calendar"
cal
exit 0
```

## 8 WRITE AND EXECUTE SHELL SCRIPT

Use any editor (gedit etc) to write shell script. Then save the shell script to a directory and name it *intro*. Shell scripts don't need a special file extension, so leave the extension blank (or you can add the extension *.sh*).

### FILE MODES AND PERMISSIONS

Every Linux file has a set of permissions that determine whether you can read, write, or run the file. Running ls -l displays the permissions. The file's mode represents the file's permissions and some extra information. There are four parts to the mode, as illustrated in Figure1 .

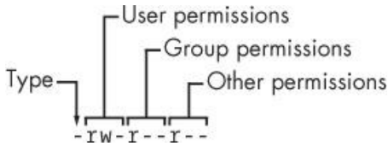



Figure 1: File Mode

The first character of the mode is the file type. A dash (-) in this position, as in the example, denotes a regular file, meaning that there's nothing special about the file. This is by far the most common kind of file. Directories are also common and are indicated by a d in the file type slot. The rest of a file's mode contains the permissions, which break down into three sets: user, group, and other, in that order. For example, the rw- characters in the example are the user permissions, the r– characters that follow are the group permissions, and the final r– characters are the other permissions.

| File Mode | Explanation |
|---|---|
| r | Means that the file is readable. |
| w | Means that the file is writable. |
| x | Means that the file is executable (you can run it as a program). |
| - | Means nothing |

The user permissions (the first set) pertain to the user who owns the file. The second set, group permissions, are for the file's group (somegroup in the example). Any user in that group can take advantage of these permissions. Everyone else on the system has access according to the third set, the other permissions.

### MODIFYING PERMISSIONS

To execute a script first we will make it executable. To change permissions, use the chmod command. Only the owner of a file can change the permissions. There are two ways to write the chmod command:

1. *chmod {a,u,g,o}{+,-}{r,w,x}*

{all, user, group, or other}, {read, write, and execute} '+' means add permission and '-' means remove permission.
For example, to add group (g) and others (o) read (r) permissions to file, you could run these two commands:

$ chmod g+r file

$ chmod o+r file

Or you could do it all in one shot:

$ chmod go+r file

To remove these permissions, use *go-r* instead of *go+r*.

2. *chmod* ⟨*3DigitNumber*⟩

   Every digit sets permission for the owner(user), group and others as shown in Table 2. To set the permission decipher the number first and then execute the command as

   $ chmod 700 file

| User | | | Group | | | Other | | | 3 Digit No. | Description |
|---|---|---|---|---|---|---|---|---|---|---|
| r | w | x | r | w | x | r | w | x | | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 400 | user: read |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 040 | group: read |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 711 | user: read/write/execute; group, other: execute |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 644 | user: read/write; group, other: read |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 600 | user: read/write; group, other: none |

Table 1: Examples of chmod with 3 Digit numbers

### RUNNING SHELL SCRIPTS

Execute your script as

./script-name

## 9 IMPORTANT POINTS

Following are some important points while writing a script:

1. # is used to write comments in your script

2. Whenever a semicolon (;) is placed between two commands, shell will treat them as separate commands. So if you want to write all the commands in your shell script in one line place a semicolon in between them.

3. A script may start with the line *#!/bin/bash* to tell your interactive shell that the program which follows should be executed by bash.

## 10 VARIABLES IN SHELL

In Linux shell there are three types of variables:

1. User defined or shell variables

2. System or Environment variables

3. Parametric variables

### USER DEFINED OR SHELL VARIABLES

The shell can store temporary variables, called shell variables, containing the values of text strings. Shell variables are very useful for keeping track of values in scripts. To assign a value to a shell variable, use the equal sign (=). Here's a simple example:

$ STUFF=blah

The preceding example sets the value of the variable named STUFF to blah. To access this variable, use

*$STUFF* (for example, try running echo *$STUFF*).

Rules for defining variables:

1. Variable name must begin with Alphanumeric character or underscore character, followed by one or more Alphanumeric character.

2. Don't put spaces on either side of the equal sign when assigning value to variable.

3. Variables are case-sensitive, just like filename in Linux.

4. You can define NULL variable as

var=

var=""

5. Do not use ?,* etc, to name your variable names.

```
# !/bin/bash
#
# variables in shell script
#
myvar=Hello
echo $myvar
myvar="Yes dear"
echo $myvar
myvar=7+5
echo $myvar
```

Notice in last assignment, myvar is assigned the string "7+5" and not the result i.e 12.

### System or Environment Variables

An environment variable is like a shell variable, but it's not specific to the shell. All processes on Unix systems have environment variable storage. The main difference between environment and shell variables is that the operating system passes all of your shell's environment variables to programs that the shell runs, whereas shell variables cannot be accessed in the commands that you run. Assign an environment variable with the shell's export command. For example, if you'd like to make the *$STUFF* shell variable into an environment variable, use the following:

$ STUFF=blah

$ export STUFF

Environment variables are useful because many programs read them for configuration and options. Listed below are some common environment variables:

1. *$PATH*
   *PATH* is a special environment variable that contains the command path (or path for short). A command path is a list of system directories that the shell searches when trying to locate a command. For example, when you run *ls*, the shell searches the directories listed in *PATH* for the ls program. If programs with the same name appear in several directories in the path, the shell runs the first matching program. If you run

   $ echo $PATH

   you'll see that the path components are separated by colons (:). For example:

   $ echo $PATH

   will output */usr/local/bin:/usr/bin:/bin*
   To tell the shell to look in more places for programs, change the PATH environment variable. For example, by using this command, you can add a directory dir to the beginning of the path so that the shell looks in dir before looking in any of the other PATH directories.

$ PATH=dir:$PATH

Or you can append a directory name to the end of the PATH variable, causing the shell to look in dir last:

$ PATH=$PATH:dir

2. *$HOME*
   The home directory of the current user.

3. *$IFS*
   An input field separator; a list of characters that are used to separate words when the shell is reading input, usually space, tab and newline characters.

**PARAMETRIC VARIABLES**

These variables keep the values of the command line arguments passed to the Scripts. Most shell scripts understand command-line parameters and interact with the commands that they run. These parametric variable passed to the script make the code more flexible. These variables are like any other shell variable like Environment and Shell Variables, except that you cannot change the values of certain ones. Following are some parametric variables and a set of environment variables used to handle parametric variables.

1. **Individual Arguments: $1, $2, ...**
   $1, $2, and all variables named as positive nonzero integers contain the values of the script parameters, or arguments. For example, say the name of the following script is pshow:

```
# !/bin/bash
echo First argument: $1
echo Third argument: $3
```

   Try running the script as follows to see how it prints the arguments:

$ ./pshow one two three

   Output looks like:
   *First argument: one*
   *Third argument: three*

   The built-in shell command shift can be used with argument variables to remove the first argument ($1) and advance the rest of the arguments forward. Specifically, $2 becomes $1, $3 becomes $2, and so on. For example, assume that the name of the following script is shiftex:

```
# !/bin/bash
echo Argument: $1
shift
echo Argument: $1
shift
echo Argument: $1
```

Run it like this to see it work:
*$ ./shiftex one two three*
*Argument: one*
*Argument: two*
*Argument: three*
As you can see, shiftex prints all three arguments by printing the first, shifting the remaining arguments,and repeating.

The shell maintains a variable called $# that contains the number of items on the command line in addition to the name of the command ($0).

```
# !/bin/bash
if [ $# -gt 0 ]; then
    echo "Your command line contains $# arguments"
    echo "All arguments displayed using \$* positional parameter"
else
    echo "Your command line contains no arguments"
fi
```

2. **Number of Arguments: $#**
   The $# variable holds the number of arguments passed to a script and is especially important when running shift in a loop to pick through arguments. When $# is 0, no arguments remain, so $1 is empty.

3. **Script Name: $0**
   The $0 variable holds the name of the script, and it is useful for generating diagnostic messages. For example, say your script needs to report an invalid argument that is stored in the *$errormsg* variable. You can print the diagnostic message with the following line so that the script name appears in the error message:

   echo $0: $errormsg

4. **Process ID: $$**
   The $$ variable holds the process ID of the shell.

## 11 Input Values

The read command is used to get a line of input into a variable.

```
# !/bin/bash
#
```

```
# Script to read your name from key-board
#
echo "Your first name please:"
read fname
echo "Hello $fname , Lets be friend!"
```

Rules to input variables:

1. Each argument must be a variable name without the leading "$".
2. The built in command reads a line of input and separates the line into individual words using the "IFS" inter field separator.
3. Each word in the line is stored in a variable from left to right.
4. The first word is stored in the first variable, the second word to the second variable and so on.
5. If there are fewer variables than words, then all remaining words are then assigned to the last variable.
6. If you have more variables than words defined, then any excess variables are set to null.

```
# !/bin/bash
#
# Script to read your name from key-board
#
read first middle last
echo "Hello $first $middle $last"
```

## 12 CONDITIONAL STATEMENTS

The Bourne shell has special constructs for conditionals, such as

1. If statement
2. If-else statement
3. If-elif statement
4. Case Statement

### IF STATEMENT

The basic syntax of if statements is:

```
# !/bin/bash
if [ conditional expression ]
then
   statement1
   statement2
fi
```

### TESTING CONDITION

test command or [ expr ]

are used to see if an expression is true, and if it is true it return zero(0) otherwise returns nonzero for false.
The folowing script determine whether given number is equal to 100 or not.

```
#!/bin/bash
#
# Script to see whether argument is positive
count=100
if [ $count -eq 100 ]
then
        echo "Count is 100"
fi
```

The folowing script determine whether given argument number is positive using *test*.

```
#!/bin/bash
#
# Script to see whether argument is positive
if test $1 -gt 0
then
        echo "$1 number is positive"
fi
```

1. **Mathematical Operators**
   Figure 2 shows different conditions that we can use to test our expressions. It's important to recognize that the equal sign (=) looks for string equality, not numeric equality. Therefore, [ 1 = 1 ] returns 0 (true), but [ 01 = 1 ] returns false. When working with numbers, use -eq instead of the equal sign: [ 01 -eq 1 ] returns true.

| Mathematical Operator in Shell Script | Meaning | Normal Arithmetical/ Mathematical Statements | But in Shell | |
|---|---|---|---|---|
| | | | For test statement with if command | For [ expr ] statement with if command |
| -eq | is equal to | 5 == 6 | if test 5 -eq 6 | if [ 5 -eq 6 ] |
| -ne | is not equal to | 5 != 6 | if test 5 -ne 6 | if [ 5 -ne 6 ] |
| -lt | is less than | 5 < 6 | if test 5 -lt 6 | if [ 5 -lt 6 ] |
| -le | is less than or equal to | 5 <= 6 | if test 5 -le 6 | if [ 5 -le 6 ] |
| -gt | is greater than | 5 > 6 | if test 5 -gt 6 | if [ 5 -gt 6 ] |
| -ge | is greater than or equal to | 5 >= 6 | if test 5 -ge 6 | if [ 5 -ge 6 ] |

Figure 2: Conditions for test and expressions

2. **String comparisons**
   Following script checks whether the script's first argument is "hi" where a string comparison is made.

```
#!/bin/bash
#
if [ $1 = hi ]; then
echo "The first argument was hi"
fi
```

   There is a slight problem with the condition in preceding example due to a very common mistake: $1 could be empty, because the user might not enter a parameter. Without a parameter, the test reads *[ = hi ]*, and the command aborts with an error. You can fix this by enclosing the parameter in quotes::

   if [ "$1" = hi ]; then

| Operator | Meaning |
|---|---|
| string1 = string2 | string1 is equal to string2 |
| string1 != string2 | string1 is NOT equal to string2 |
| -n string1 | string1 is NOT NULL and does exist |
| -z string1 | string1 is NULL and does exist |

Table 2: String Comparisons

   *-z* and *-n* work with the string enclosed within double quotes.

```
# !/bin/bash
# String Comparisons

string1=$1
if [ -z "$string1" ] ; then
        echo "NULL STRING"
fi
if [ -n "$string1" ] ; then
        echo "NOT NULL STRING"
fi
```

3. **Test for file and directory**
   Most file tests that are commonly used are unary operations because they require only one argument: the file to test.

```
#!/bin/bash
somefile=a.txt
if [ -r $somefile ]; then
        content=$(cat $somefile)
        echo $content
elif [ -f $somefile ]; then
        echo "The file 'somefile' exists but is not readable to the script."
```

```
else
        echo "The file 'somefile' does not exist."
fi
```

In the above script first we check if the file somefile is readable. If so, we read it into a variable. If not, we check if it actually exists. If that's true, we report that it exists but isn't readable otherwise it moves to the else condition.

| Test | Meaning |
|---|---|
| -s file | Non empty file |
| -f file | Is file exist or normal file and not a directory |
| -d dir | Is directory exist and not a file |
| -w file | Is writable file |
| -r file | Is read-only file |
| -x file | Is executable file |

Table 3: Test for File and Directory

4. **Logical Operators**
You can invert a test by placing the ! operator before the test arguments. For example, *[ ! -f file ]* returns true if file is not a regular file. Furthermore, the -a and -o flags are the logical *and* and *or* operators. For example, *[ -f file1 -a file2 ]*.

| Operator | Meaning |
|---|---|
| ! expression | Logical NOT |
| expression1 -a expression2 | Logical AND |
| expression1 -o expression2 | Logical OR |

Table 4: Logical Operators

### IF ELSE STATEMENT

The basic syntax of if-else statements is:

```
if condition
then
    execute all commands upto else
else
   execute all commands upto fi
fi
```

Examples using if-else

```
# !/bin/bash
echo "Is it morning? Please answer yes or no"
read timeofday
if [ $timeofday = "yes" ]; then
     echo "Good morning"
```

```
else
     echo "good afternoon"
fi
```

```
# !/bin/bash
count=99
if [ $count -eq 100 ]
then
   echo "Count is 100"
else
   echo "Count is not 100"
fi
```

```
# !/bin/bash
#script to see whether argument is positive or negative
if [ $# -eq 0 ]
then
     echo "$0:You must supply one integer"
     exit 1
fi
if test $1 -gt 0
then
     echo "$1 is positive number"
else
     echo "$1 is negative"
fi
```

**IF-ELIF STATEMENT**

The basic syntax of if-elif statements is:

```
if condition
then
     .............
elif condition
then
     .............
else
     .............
fi
```

Example:

```
# !/bin/bash
echo "Is it morning? Please answer yes or no"
read timeofday
if [ $timeofday = "yes" ]; then
     echo "Good morning"
elif [ $timeofday = "no" ]; then
     echo "good afternoon"
else
      echo "Sorry, $timeofday not recognized. Enter yes or no"
      exit 1
fi
exit 0
```

### CASE STATEMENT

For each case value, you can match a single string or multiple strings with | (hi|hello returns true if $1 equals hi or hello in third example), or you can use the * or ? patterns (what*). To make a default case that catches all possible values other than the case values specified, use a single * as shown by the final case in the examples. The syntax of case statement is:

```
case $variable-name in
pattern1) command.....command;;
pattern2) command .....command;;
patternN) command.....command;;
*) command..... command;;
esac
```

Examples

```
# !/bin/bash
echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
yes) echo "Good Morning" ;;
no ) echo "Good Afternoon" ;;
y ) echo "Good Morning" ;;
n ) echo "Good Afternoon" ;;
* ) echo "Sorry, answer not recognized" ;;
esac
```

```
# !/bin/bash
case $1 in
bye)
   echo Fine, bye.
   ;;
hi|hello)
   echo Nice to see you.
   ;;
what*)
   echo Whatever.
   ;;
*)
echo 'Huh?'
;;
esac
```

## 13 ARITHMETIC OPERATIONS

Arithmetic expansion and evaluation is done by placing an integer expression using the following format:

$((expression))

$(( n1+n2 ))

The shell script below includes some arithmetic operations.

```
# !/bin/bash
clear
echo "hello world"
var1=10;
var2=20;
var3=$(( $var1 + $var2 ));
var4=$(( $var1 - $var2 ));
var5=$(( $var1 * $var2 ));
var6=$(( $var1 / $var2 ));
echo $var3;
echo $var4;
echo $var5;
echo $var6;
exit
```

## 14 LOOPS

There are two kinds of loops in the shell: for and while loops.

1. **For Loop**
   The basic syntax of for loop is:

   ```
   for variable_name in { list of values }
   do
       execute one for each item in the list until list is not finished
       repeat all statement between do and done
   done
   ```

   Examples:

   ```
   #!/bin/bash
   for i in 1 2 3 4 5
   do
      echo "Welcome $i times"
   done
   ```

   ```
   #!/bin/bash
   i=1
   for day in Mon Tues Wed Thu Fri
   do
       echo "Weekday $((i++)) : $day"
   done
   ```

   Sample Output:
   *Weekday 1 : Mon*
   *Weekday 2 : Tue*
   *Weekday 3 : Wed*
   *Weekday 4 : Thu*

*Weekday 5 : Fri*

```
#!/bin/bash
#define interval in for loop
#the loop execute from 1 to 5
for i in {1..5}
do
   echo "Welcome $i times"
done
```

```
#!/bin/bash
#the loop execute from 0 to 10
#with a jump of 2 in each interation
for i in {0..10..2}
do
   echo "Welcome $i times"
done
```

2. **While Loop**
   The basic syntax of while loop is:

```
#!/bin/bash
while [ condition ]
do
   command_1
   command_2
   ..............
   command_N
done
```

Examples:

```
#!/bin/bash
echo "Enter password"
read trythis

while [ "$trythis" != "secret" ]; do
    echo "Sorry, try again"
    read trythis
done
```

```
#!/bin/bash
#set n to 1
n=1

#continue until n equals 5
while [ $n -le 5 ]
do
    echo "Welcome $n times."
    n=$(( n+1 ))
done
```

Sample Output:
*Welcome 1 times.*
*Welcome 2 times.*
*Welcome 3 times.*
*Welcome 4 times.*
*Welcome 5 times.*

```
#!/bin/bash
counter=$1
factorial=1
while [ $counter -gt 0 ]
do
   factorial=$(( $factorial * $counter ))
   counter=$(( $counter - 1 ))
done
echo $factorial
```

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
ISLAMABAD

OPERATING SYSTEMS LAB SPRING 2019

# Lab Manual 02
# Process Creation
*fork, wait, exit, getpid and getppid Unix System Calls*

## 15 COMPILE AND EXECUTE A C PROGRAM

Write down the C code in a text editor of your choice and save the file using **.c** extension i.e. *filename.c*. Now through your terminal move to the directory where *filename.c* resides.

```
gcc firstprogram.c
```

It will compile the code and by default an executable named *a.out* is created.

```
gcc -o firstprogram firstprogram.c
```

If your file is named firstprogram.c then type *-o firstprogram* as the parameter to gcc. It would create an executable by the name *firstprogram* for the source code named *firstprogram.c* .
To execute the program simply write the following:

```
./a.out OR ./firstprogram
```

In case you have written the code in C++ saved using .cpp extension, compile the code using g++ as:

```
g++ filename.cpp OR g++ -o exec_name filename.cpp
```

And execute as ./a.out or ./exec_name

## 16 Process Creation

When is a new process created?

1. System startup
2. Submit a new batch job/Start program
3. Execution of a system call by process
4. A user request to create a process

On the creation of a process following actions are performed:

1. Assign a unique process identifier. Every process has a unique process ID, a non-negative integer. Because the process ID is the only well-known identifier of a process that is always unique, it is used to guarantee uniqueness.
2. Allocate space for the process.
3. Initialize process control block.
4. Set up appropriate linkage to the scheduling queue.

## 17 Fork System Call

An existing process can create a new process by calling the fork function.

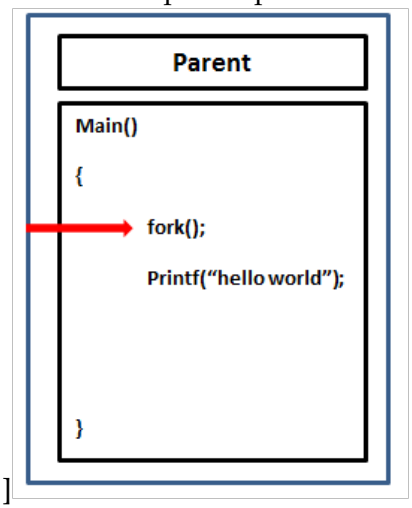
```
#include <unistd.h>
pid_t fork(void);
// Returns: 0 in child, process ID of child in parent, −1 on error
```

The definition of the pid_t is given in <sys/types> include file and <unistd.h> contain the declaration of *fork* system call.

### Important Points

1. The new process created by fork is called the child process.
2. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
3. Both the child and the parent continue executing with the instruction that follows the call to fork.
4. The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack. Note that this is a copy for the child; the parent and the child do not share these portions of memory.

5. In general, we never know whether the child starts executing before the parent, or vice versa. The order depends on the scheduling algorithm used by the kernel.

21 [The process will execute sequentially until it arrives at fork system call. This process will be named as parent process and the one created using fork() is termed as child process.] 

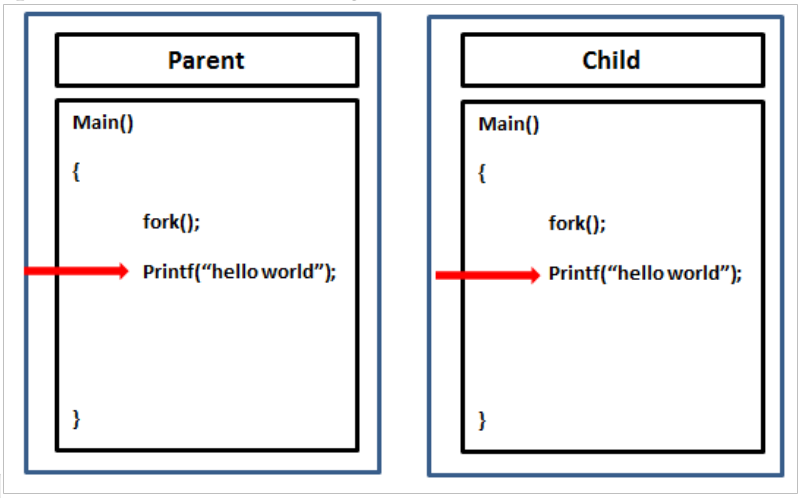
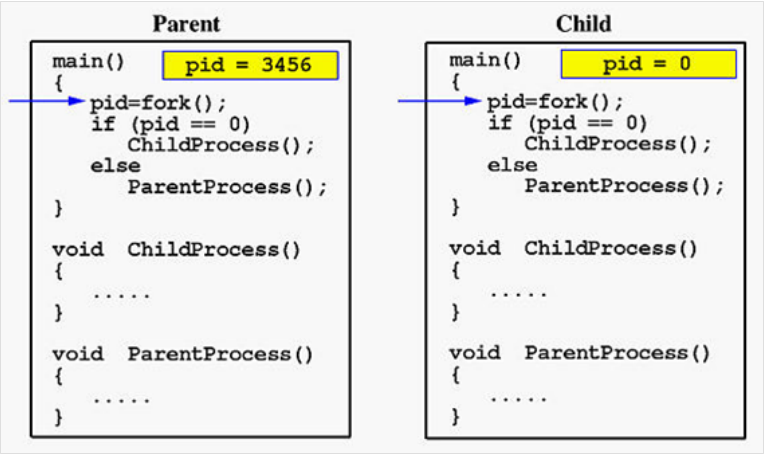
[After fork system call both the child and the parent process continue executing with the instruction that follows the call to fork.] 


Figure 3: Parent and Child Process

21 [Fork system call returns child process ID (created using fork) in parent process and zero in child process] 

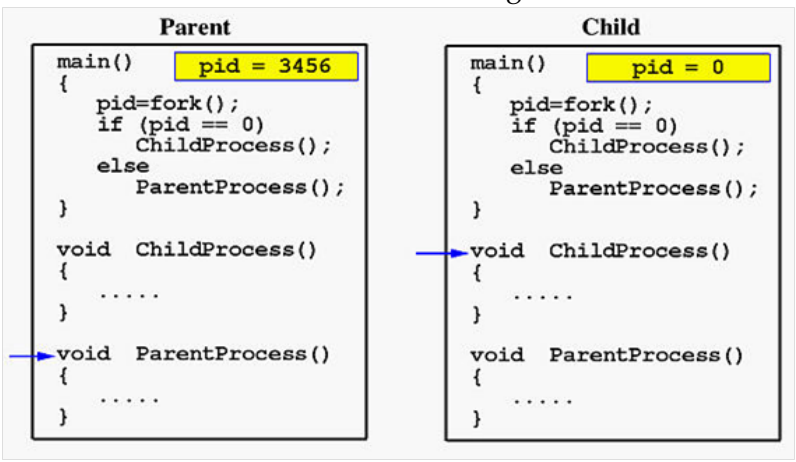
[We can execute different portions of code in parent and child process based on the return value of fork that is either zero or greater than zero.] 


Figure 4: Return Value of fork system call

## 18 Examples Of Fork()

1. Fork()'s Return Value

```
#include  <stdio.h>
#include  <unistd.h>
#include  <sys/types.h>
#include <stdlib.h>

void main(void){
        pid_t  pid=fork();
        if(pid == 0){ //This check will pass only for child process
                printf("I am Child process \n");
        }
```

```
        else if (pid > 0){ // THIS Check will pass only for parent
                printf("I am Parent process \n");
        }

        else if (pid < 0){ // if fork() fails
                printf("Error in Fork");
        }
}
```

2. Manipulating Local and Global Variables

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>

int global=0;

int main()
{
        int status;
        pid_t child_pid;
        int local = 0;
        /* now create new process */
        child_pid = fork();

        if (child_pid >= 0){ /* fork succeeded */
                if (child_pid == 0){
                        /* fork() returns 0 for the child process */
                        printf("child process!\n");
                        // Increment the local and global variables
                        local++;
                        global++;
                        printf("child PID =  %d, parent pid = %d\n",
                                        getpid(), getppid());
                        printf("\n child's local = %d, child's
                                        global = %d\n",local,global);
        }

                else{ /* parent process */
                        printf("parent process!\n");
                        printf("parent PID =  %d, child pid = %d\n",
                                        getpid(), child_pid);
                        int w=wait(&status);
                        //The change in local and global variable
                        //in child process should not reflect
                        //here in parent process.
                        printf("\n Parent'z local = %d, parent's  global
                                        = %d\n",local,global);
                        printf("Parent says bye!\n");
                        exit(0);  /* parent exits */
        }
```

```
}

        else{ /* failure */
                perror("fork");
                exit(0);
        }
}
```

## 19 WAIT SYSTEM CALL

This function blocks the calling process until one of its child processes exits. wait() takes the address of an integer variable and returns the process ID of the completed process.

```
pid_t wait(int *status)
```

Include <sys/wait.h> and <stdlib.h> for definition of wait().
The execution of wait() could have two possible situations

1. If there are at least one child processes running when the call to wait() is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution
2. If there is no child process running when the call to wait() is made, then this wait() has no effect at all. That is, it is as if no wait() is there.

### EXAMPLE

```
#include  <stdio.h>
#include  <unistd.h>
#include  <sys/types.h>
#include <stdlib.h>

void  main(void){
        int status=0;
        pid_t  pid=fork();
        if(pid == 0){ //This check will pass only for child process
                printf("I am Child process with pid %d and i am not
                        waiting\n",getpid());
                exit(status);
        }

        else if (pid > 0){ // THIS Check will pass only for parent
                printf("I am Parent process with pid %d and
                        i am waiting\n",getpid());
                pid_t exitedChildId=wait(&status);
                printf("I am Parent process and the child with pid %d
                        is exited\n",exitedChildId);
        }

        else if (pid < 0){ // if fork() fails
                printf("Error in Fork");
        }
```

```
}
```

## 20 EXIT SYSTEM CALL

A computer process terminates its execution by making an exit system call.

```
#include <stdlib.h>
int main(){
        exit(0);
}
```

## 21 COMMAND LINE ARGUMENT TO C PROGRAM

```
void main(int argc, char *argv[]){
        /* argc -- number of arguments */
        int i;
        for(i=0;i<argc;i++){
                printf("The argument at %d index is %s\n",i,argv[i]);
        }
}
```

To run:
./commandlineargument.out abc def 1 2

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
ISLAMABAD

OPERATING SYSTEMS LAB SPRING 2019

# Lab Manual 03
# Exec System Call

*A family of six functions*

## 22 EXEC() SYSTEM CALL

Forking provides a way for an existing process to start a new one, but what about the case where the new process is not part of the same program as parent process? This is the case in the shell; when a user starts a command it needs to run in a new process, but it is unrelated to the shell. This is where the exec system call comes into play. exec will replace the contents of the currently running process with the information from a program binary.

1. The *exec* function is actually a family of six functions each with slightly different calling conventions and use.
2. Like *fork*, *exec* is declared in <unistd.h>.
3. *exec* completely replaces the calling process's image with that of the program being executed.
4. *fork* creates a new process, and so generates a new PID. *exec* initiates a new program, replacing the original process. Therefore, the PID of an *execed* process doesn't change.
5. If successful, the *exec* calls do not return as the calling image is lost.

## 23 PROTOTYPES OF EXEC()

Prototypes of exec() are:

```
int execl (const char *path, const char *arg, ...);

int execlp(const char *file, const char *arg, ...);

int execv(const char *path, char *const argv[]);

int execvp(const char *file, char *const argv[]);

int execle(const char *path, const char *arg, char *const envp[]);

int execve(const char *path, const char *argv[], char *const envp[]);
```

## 24 NAMING CONVENTION OF EXEC()

The naming convention for the "exec" system calls reflects their functionality:

1. The function starts with *exec* indicating that one of the function from *exec* family is called.
2. The next letter in the call name indicates if the call takes its arguments in a "list format" (i.e. literally specified as a series of arguments) or as a pointer to an "array of arguments".
   (a) The functions execl, execlp, and execle require each of the command-line arguments to the new program to be specified as separate arguments, terminated by a NULL pointer.
   (b) For the other functions (execv, execvp, execve), we have to build an array of pointers to the arguments.
3. The presence of a "p" indicates the current PATH string should be used when the system searches for executable files. (If the executable file is script file, the shell is invoked to execute the script. The shell is then passed the specified argument information).
4. The functions whose name end with an *e* allow to pass array to set new environment. Otherwise the calling process copy the existing environment for the new program.

| Letters | Explanation |
|---|---|
| l | argv is specified as a list of arguments. |
| v | argv is specified as a vector (array of character pointers). |
| e | environment is specified as an array of character pointers. |
| p | PATH is searched for command, and command can be a shell program |

## 25 EXECL()

```
int execl(const char *path, const char *arg, ...)
```

1. Takes the path name of an executable program (binary file) as its first argument.
2. The rest of the arguments are a list of command line arguments to the new program.
3. The list is terminated with a null pointer.
4. Examples:

```
execl("/bin/ls", "ls", "-l", "/usr", NULL);
execl("/home/lab/Desktop/a.out","./a.out","arg 1","arg 2","arg n",NULL)
```

### EXAMPLE 01

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(){
        pid_t childpid = fork();
        if(childpid==0){
                printf("I am a child proce with pid = %d \n",getpid());
                printf("The next statement is execl and ls will run \n");
                execl("/bin/ls","ls","-l","/usr",NULL);
                printf("Execl failed");
        }
        else if(childpid>0){
                wait(NULL);
                printf("\n I am parent process with pid = %d
                                        and finished waiting \n",getpid());
        }
```

```
        return 0;
}
```

**EXAMPLE 02**

Apart from executing the shell commands we can run other executables as well. In this example we will write a code in *C* and create an executable named as *execExam*. Given below is a simple code in *C* that prints the arguments passed to it through terminal.

```
#include <stdio.h>
void main( int argc , char *argv[] ) {
   int i ;
   for( i=0 ; i < argc ; i++ ) {
      printf( "Argument %d : %s\n" , i , argv [ i ] ) ;
   }
}
```

Now compile the code and create an object file *execExam* :

```
gcc -o execExam filename.c
```

Following program will now create a new process using fork system call. Then the child process will run the executable of the above program using *execl()*.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(){
        pid_t childpid = fork();
        if(childpid==0){
                printf("I am a child proce with pid = %d \n",getpid());
                printf("The next statement is execl and execExam executable
                                        will run \n");
                execl("/home/sarosh/Desktop/execExam","ls","-l","/usr",NULL);
                printf("Execl failed");
        }
        else if(childpid>0){
                wait(NULL);
                printf("\n I am parent process with pid = %d and
                                        finished waiting \n",getpid());
        }
        return 0;
}
```

## 26 EXECLP()

```
int execlp(const char *file , const char *arg, ...)
```

1. Same as execl(), except that the program name doesn't have to be a full path name and it can be a shell program instead of an executable module. The program directory should be in your PATH variable.
2. The rest of the arguments are a list of command line arguments to the new program.
3. The list is terminated with a null pointer.
4. Example:

```
execlp("ls", "ls", "-l", "/usr", NULL);
execlp("cat", "cat", "filename", NULL);
execlp("./a.out", "a.out", "arg 1", NULL);
```

**EXAMPLE 01**

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(){
        pid_t childpid = fork();
        if(childpid==0){
                printf("I am a child proce with pid = %d \n",getpid());
                printf("The next statement is execlp and ls will run \n");
                execlp("ls","ls","-l","/usr",NULL);
                printf("Execl failed");
        }
        else if(childpid>0){
                wait(NULL);
                printf("\n I am parent process with pid = %d and
                                finished waiting \n",getpid());
}
return 0;
}
```

## 27 EXECV()

```
int execv(const char *path, char *const argv[]);
```

1. Takes the path name of an executable program (binary file) as it first argument.
2. The second argument is a pointer to a list of character pointers (like argv[]) that is passed as command line arguments to the new program.
3. The list is terminated with a null pointer.

4. Example:

```
char *args[] = { "cat", "filename1", NULL };
execv("/bin/cat", args);

char *args[] = { "./a.out", "arg1", "arg2", NULL };
execv("/home/lab/Desktop/a.out",args);
```

**EXAMPLE 01**

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(){
        pid_t childpid = fork();
        if(childpid==0){
                printf("I am a child proce with pid = %d \n",getpid());
                printf("The next statement is execv and ls will run \n");
                char* argv[]={"ls","-l","/usr",NULL};
                execv("/bin/ls",argv);
                printf("Execl failed");
        }
        else if(childpid>0){
                wait(NULL);
                printf("\n I am parent process with pid = %d and
                                finished waiting \n",getpid());
        }
        return 0;
}
```

## 28 EXECVP()

```
int execvp(const char *file, char *const argv[]);
```

1. Same as execv(), except that the program name doesn't have to be a full path name, and it can be a shell program instead of an executable module.
2. The second argument is a pointer to a list of character pointers (like argv[]) that is passed as command line arguments to the new program.
3. The list of character pointer should be terminated with a null pointer.
4. Example:

```
char *args[] = { "cat", "f1", "f2", NULL };
execvp("cat", args);
```

**EXAMPLE 01**

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(){
        pid_t childpid = fork();
        if(childpid==0){
                printf("I am a child proce with pid = %d \n",getpid());
                printf("The next statement is execv and ls will run \n");
                char* argv[]={"ls","-l","/usr",NULL};
                execvp("ls",argv);
                printf("Execl failed");
        }
        else if(childpid>0){
                wait(NULL);
                printf("\n I am parent process with pid = %d and
                                finished waiting \n",getpid());
}
return 0;
}
```

## 29 EXECLE()

```
int execle(const char *path, const char *arg, char *const envp[])
```

1. Same as execl(), except that the end of the argument list is followed by a pointer to a null-terminated list of character pointers that is passed as the environment of the new program.

2. Example:

```
char *env[] = { "export TERM=vt100", "PATH=/bin:/usr/bin", NULL };
execle("/bin/cat", "cat", "f1", NULL, env);
```

## 30 EXECVE()

```
int execve(const char*path, const char *argv[], char *const envp[]);
```

1. Same as execv(), except that a third argument is given as a pointer to a list of character pointers (like argv[]) that is passed as the environment of the new program.

2. Example:

```
char *env[]={ "export TERM=vt100", "PATH=/bin:/usr/bin", NULL};
char *args[]={ "cat", "f1", NULL };
execve("/bin/cat", args, env);
```

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
ISLAMABAD

OPERATING SYSTEMS LAB SPRING 2019

# Lab Manual 04
# Inter Process Communication

## 31 FILE DESCRIPTOR

To the kernel, all open files are referred to by file descriptors. A file descriptor is a non-negative integer. Every time a file is opened, one of the free file pointers is used to point to the new file. Linux processes expect three file descriptors to be open when they start. These are known as

1. standard input(0)
2. standard output(1)
3. standard error(2)

The program treat them all as files. All accesses to files are via standard system calls which pass or return file descriptors.

## 32 READ() SYSTEM CALL

**read** - read from a file descriptor

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count)
```

read() attempts to reads up to count bytes of data from the file associated with the file descriptor *fd* and places them into the buffer starting at *buf*.
**RETURN VALUE**

It returns the number of data bytes actually read, which may be less than the number requested. If a read call returns 0, it had nothing to read; it reached the end of the file. An error on the call will cause it to return -1.

## 33 WRITE() SYSTEM CALL

**write** - write to a file descriptor

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count)
```

write() writes up to *count* bytes from the buffer pointed *buf* to the file referred to by the file descriptor *fd*.

### RETURN VALUE

It returns the number of bytes actually written. This may be less than count if there has been an error in the file descriptor or if the underlying device driver is sensitive to block size. If the function returns 0, it means no data was written. If it returns -1, there has been an error in the write call.

## 34 FILE DESCRIPTORS EXAMPLE

```
char buffer[10];
//Read from standard input (by default it is keyboard)
read(0,buffer,10);
//Write to standard output (by default it is monitor))
write(1,buffer,10);
//By changing the file descriptors we can write to files
```

## 35 READING AND WRITING FROM STANDARD INPUT/OUTPUT

```
#include <iostream>
#include <stdlib.h>
#include <unistd.h>
using namespace std;
int main(){
        char buffer[10];
        cout<<"Enter string"<<endl;
        read(0,buffer,10);
        write(1,buffer,10);
        return 0;
}
```

## 36 PIPES

Pipelines are the oldest form of UNIX inters process communication. They are used to establish one-way communication between processes that share a common ancestor. The pipe system call is used to create a pipeline:

```
int pipe(int pipefd[2])
```

The array *pipefd* is used to return two file descriptors referring to the ends of the pipe:

1. pipefd[0] is open for reading.
2. pipefd[1] is open for writing.

Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

### EXPLANATION

1. Pipe system call opens a pipe, which is an area of main memory that is treated as a *virtual file*. The pipe can be used by the creating process, as well as all its child processes, for reading and writing.
2. One process can write to this *virtual file* or pipe and another related process can read from it.
3. If a process tries to read before something is written to the pipe, the process is suspended until something is written.
4. The pipe system call finds the first two available positions in the process's open file table and allocates them for the read and write ends of the pipe.
5. **Implementation**: A pipe can be implemented as a 10k buffer in main memory with 2 pointers, one for the FROM process and one for TO process.

## 37 PIPES AFTER FORK

Typically, a process creates the pipeline, then uses "fork" to create a child process. Each process now has a copy of the file descriptor array; one process writes data into pipeline, while the other reads from it.

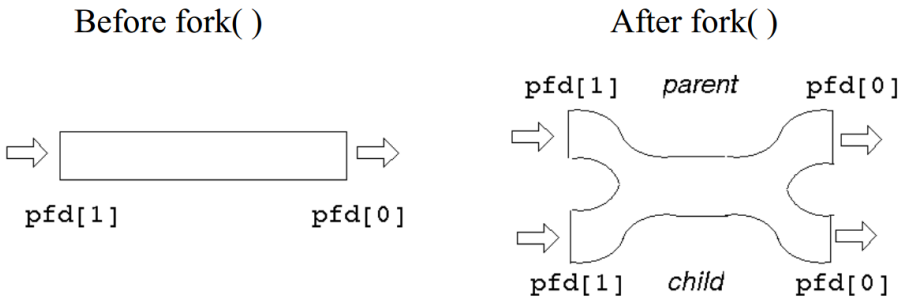



Figure 5: Structure of pipe after fork()

This gives two read ends and two write ends. Either process can write into the pipe, or either can read from it. Which process will get what is not known? For predictable behaviour, one of the processes must close its read end, and the other must close its write end. Then it will become a simple pipeline again.

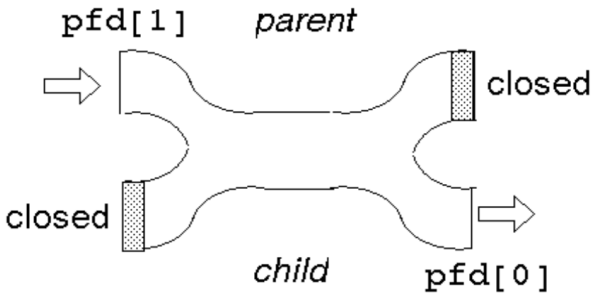



Figure 6: Closing ends of pipe

## 38 EXAMPLES OF PIPE

### EXAMPLE 01

```
#include <iostream>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
int main(){
        int n;
        int fd[2];
        char buf[1025];
        char const* data = "Hello this is written to pipe";
        pipe(fd);
        write(fd[1],data,strlen(data));
        if((n=read(fd[0],buf,1024))>=0){
                buf[n] = 0;
                printf("Read %d bytes from pipe %s \n\n",n,buf);
```

```
        }
        else{
                perror("read");
                exit(0);
        }
        return 0;
}
```

**EXAMPLE 02**

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include<iostream>
#include<sys/wait.h>
using namespace std;
int main(){
        int fd[2];
        pid_t childpid;
        char string[] = "Hello world\n";
        char readbuffer[80];
        int result = pipe (fd);
        if(result <0){
                cout<<"Error while creating file";
                exit(1);
        }

        childpid = fork();
        if(childpid == -1){
                cout<<"Error in fork"<<endl;
                exit(1);
        }

        if(childpid == 0){
                close(fd[0]);
                cout<<"Child writing to the pipe"<<endl;
                write(fd[1],string, sizeof(string));
                cout<<"Written to a file"<<endl;
                exit(0);
        }else{
                close(fd[1]);
                wait(NULL);
                cout<<"Parent reading from the pipe"<<endl;
                read(fd[0], readbuffer, sizeof(readbuffer));
                cout<<"Received string: "<<readbuffer<<endl;
                exit(0);
        }
return 0;
}
```

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
ISLAMABAD

OPERATING SYSTEMS LAB SPRING 2019

# Lab Manual 05
# Inter Process Communication
*Named Pipes FIFO*

## 39 NAMED PIPES FIFO

A named pipe works much like a regular pipe, but does have some noticeable differences.

1. Named pipes exist as a device special file in the file system.
2. Processes of different ancestry can share data through a named pipe.
3. When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

## 40 CREATING FIFO

There are several ways of creating a named pipe.

1. It can be done directly from the shell.
```
mkfifo a=rw MYFIFO
```
2. To create a FIFO in C, we can make use of the mkfifo() system call.

FIFO files can be quickly identified in a physical file system by the "**p**" indicator seen here in a long directory listing.
$ ls -l

```
prw-r--r-- 1 root root 23 FEB 14 22:15 MYFIFO|
```

Also notice the vertical bar ("pipe sign") located directly after the file name.

## 41 USING FIFO THROUGH SHELL

As mentioned before pipe can be created directly from the shell using:

```
mkfifo a
```

Now open 2 different terminals and run the following commands.

```
Terminal 01: ls -l > a
Terminal 02: cat < a
```

where *ls -l* is writing in pipe **a** while *cat* is reading from the pipe and displaying the content.

## 42 CREATING FIFO IN C++

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include<iostream>
#include<sys/wait.h>
 #include <sys/stat.h>
using namespace std;
int main(){
        int f1, f2;
        //create a pipe with name "pipe_one" and set permissions to 0666
        f1 = mkfifo("pipe_one",0666);
        //check if mkfifo call was successful
        if(f1<0)
                cout<<"Pipe one not created"<<endl;
        else
                cout<<"Pipe one created"<<endl;

        f2 = mkfifo("pipe_two",0666);
        if(f2<0)
                cout<<"Pipe two not created"<<endl;
        else
                cout<<"Pipe two created"<<endl;
        return 0;
}
```

## 43 OPEN() SYSTEM CALL

An "open" system call or library function should be used to physically open up a channel to the pipe.

```
int open(const char *pathname, int flags);
```

Given a pathname for a file, open() returns a file descriptor.
The argument flags must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file read- only, write-only, or read/write, respectively.

## 44 CLOSE() SYSTEM CALL

close() closes a file descriptor, so that it no longer refers to any file and may be reused.

```
int close(int fd)
```

close() returns zero on success. On error, -1 is returned. Not checking the return value of close() is a common but nevertheless serious programming error.

## 45 COMMUNICATION BETWEEN DIFFERENT ANCESTRY PROCESSES

### PROGRAM 01

Write the code in a file and run it in Terminal 01.

```
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <iostream>
using namespace std;
int main(){
        char str[256] = "hello world";
        int fifo_write;
        //open "pipe_one" with WRITE only mode
        // and return its file descriptor
        fifo_write = open ("pipe_one",O_WRONLY);
        //check if open call was successful
        if(fifo_write < 0){
                cout<<"Error opening file"
        }
        else{
                while(strcmp(str, "abort")!=0){
                        cout<<"Enter text: "<<endl;
                        cin>>str;
                        write(fifo_write, str, sizeof(str));
                        cout<<"*"<<str<<"*"<<endl;
                }
                close (fifo_write);
        }
        return 0;
}
```

### PROGRAM 02

Write the code in a file and run it in Terminal 02.

```
#include <unistd.h>
#include <string.h>
```

```
#include <fcntl.h>
#include <iostream>
using namespace std;
int main(){
        char str[256] = "hello world";
        //open "pipe_one" with READ only mode
        // and return its file descriptor
        int fifo_read = open ("pipe_one", O_RDONLY);
        //check if open call was successful
        if(fifo_read < 0){
                cout<<"Error opening file"
        }
        else{
                while(strcmp(str,"abort")!=0){
                        read(fifo_read, str, sizeof(str));
                        cout<<"Text: "<<str<<endl;
                }
                close (fifo_read);
        }
        return 0;
}
```

## 46 UNLINK()

unlink() deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse. If the name was the last link to a file but any processes still have the file open, the file will remain in existence until the last file descriptor referring to it is closed. If the name referred to a socket, FIFO, or device, the name for it is removed but processes which have the object open may continue to use it.

### EXAMPLE

```
// creating and unlinking pipes in one file
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <iostream>
#include<sys/wait.h>
 #include <sys/stat.h>
using namespace std;
int main(){
        char str[256] = "hello world";
        int f1, f2;
        f1 = mkfifo("pipe_one",0666);
        if(f1<0)
                cout<<"Pipe one not created"<<endl;
        else
                cout<<"Pipe one created"<<endl;

        f2 = mkfifo("pipe_two",0666);
```

```
        if(f2<0)
                cout<<"Pipe two not created"<<endl;
        else
                cout<<"Pipe two created"<<endl;

        //removing pipes
        unlink("pipe_one");    //remove pipe_one
        unlink("pipe_two");    //removing pipe_two
        return 0;
}
```

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
ISLAMABAD

OPERATING SYSTEMS LAB SPRING 2019

# Lab Manual 06
# Signals

**Q:** How does the OS communicate to an application process?
**A:** Signals.

## 47 WHAT IS SIGNAL

A notification that intimates a receiving process about occurrence of an event. With pipes we communicated data, with signal we can communicate control command.

Signals can be sent

1. By a process to another.
2. By kernel to other processes e.g. when a child process call exit() SIGCHLD is sent to the parent process by the kernel by default it is ignored.

The only information is:

- The number identifying the signal.

Signal interrupts a process and forces it to handle the event immediately.

## 48 UNDERSTANDING SIGNAL

Signal: A notification of an event

1. Event gains attention of the OS.
2. OS stops the application process immediately, sending it a signal.

3. **Signal handler** executes to completion.
4. Application process resumes where it left off.

**EXAMPLE**
User types Ctrl-c

1. Event gains attention of OS
2. OS stops the application process immediately, sending it 2/SIGINT signal
3. Signal handler for 2/SIGINT signal executes to completion
4. Default signal handler for 2/SIGINT signal exits process

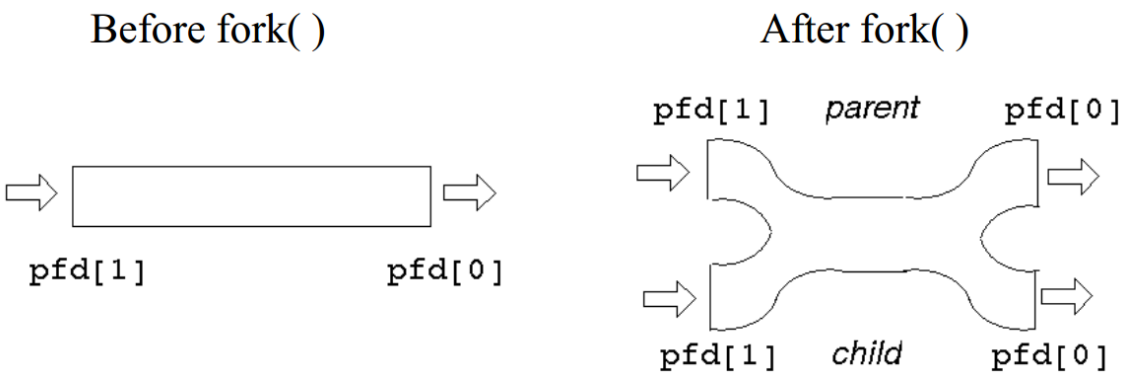



Figure 7: Signal

## 49 Actions Performed upon Receiving a Signal

There are three ways in which a process can respond to a signal:

1. Explicitly ignore the signal.
2. Execute the default action associated with the signal.
3. Catch the signal by invoking a corresponding signal-handler function.

## 50 Sending Signals via Keystrokes

Three signals can be sent from keyboard:

1. Ctrl-c – 2/SIGINT signal
   Default handler exits process.
2. Ctrl-z – 20/SIGTSTP signal
   Default handler suspends process.

3. Ctrl- – 3/SIGQUIT signal
   Default handler exits process.

## 51 List of Signals and default action

kill -l can be used to view all the signals supported by your system.

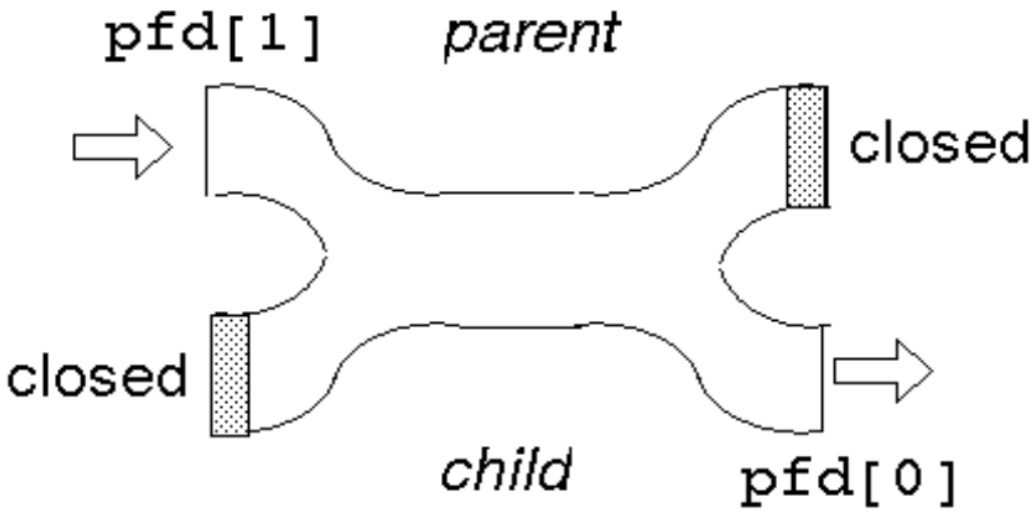



Figure 8: List of Signals and default action

## 52 The signal() System Call

signal() system call is used to set signal handler for a signal type.

```
#include <signal.h>
sighandler_t signal(int signum, sighandler_t handler);
```

1. signal() sets the disposition of the signal signum to handler, which is one of the following:
   (a) SIG_IGN
   (b) SIG_DFL
   (c) The address of a programmer-defined function (a "signal handler")
2. signal() returns the previous value of the signal handler, or SIG_ERR on error.

## 53 Signal Handler

- Corresponding to each signal is a signal handler.

- Called when a process receives a signal.
- The function is called "asynchronously".
- When the signal handler returns the process continues, as if it was never interrupted.

## 54 Pre-defined Signal Handlers

Pre-defined Signal Handlers are:

1. **SIG_IGN:**
   Causes the process to ignore the specified signal.
   *Example:* To ignore Ctrl-C completely – signal(SIGINT, SIG_IGN);
2. **SIG_DFL:**
   Causes the system to set the default signal handler for the given signal
   *Example:* signal(SIGTSTP, SIG_DFL);

**EXAMPLE**

```
#include <iostream>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
using namespace std;

int main(){
        signal(SIGQUIT, SIG_IGN);  //ignoring CTRL backlash
        signal(SIGTSTP, SIG_IGN);  //ignoring CTRL Z
        signal(SIGINT, SIG_IGN);  //ignoring CTRL C

        for(int i=0;;i++){
                cout<<i<<"  My pid is: "<<getpid()<<endl;
                cout<<"You can't close me using conventional commands"<<endl;
                sleep(2);
        }
return 0;

}
```

## 55 User Defined Signal Handlers

1. User can Define a functions to be executed on receiving different signals.
2. Most of the signals can be caught or handled except SIGKILL and SIGSTOP.
3. Signal Handlers can not be called at any point except for the signal generation.

**EXAMPLE**

```
#include <iostream>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
using namespace std;

void myhandler(int);

int main(){
        signal(SIGINT, myhandler);  //adding custom handler to CTRL-C

        for(int i=0;;i++){
                cout<<i<<endl;
                //just slowing down execution so can observe better
                sleep(2);
        }
return 0;
}

void myhandler (int signal_number){
        signal (SIGINT, myhandler); //resetting the signal
        cout<<"Inside my custom handler and received signal # "
                <<signal_number<<endl;
}
```

## 56 THE KILL() SYSTEM CALL

The kill() system call can be used to send any signal to any process.
If pid is positive, then signal sig is sent to pid.

```
int kill(pid_t pid, int sig);
```

To send signal using terminal the syntax for kill command is:

```
kill [signal or option] PID(s)
$ kill -SIGKILL Pid
```

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.
To check if a process exists or not use kill() by sending signal#0 to that process.

```
If(kill(pid,0)==0)
        process exist
else
        do not exist
```

**EXAMPLE 01**

```
#include <iostream>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>
using namespace std;

int main(){
```

```
        pid_t pid;
        pid = fork();
        if(pid<0){
                cout<<"Child process not created"<<endl;
        }

        else if(pid>0){
                cout<<"I am parent with pid "<<getpid()
                        <<" and child pid is "<<pid<<endl;
                wait(NULL);
        }

        else{
                cout<<"I am child with pid "<<getpid()
                        <<" and my parent pid is "<<getppid()
                        <<endl;
                kill(getppid(),SIGKILL);
                cout<<"My parent is now "<<getppid()<<endl;
                sleep(1);
                cout<<"My parent is now "<<getppid()<<endl;
        }

        return 0;
}
```

**EXAMPLE 02**

```
#include <iostream>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>
using namespace std;

void handler (int signum){
        cout<<"Signal is received "
                <<signum<<endl;
        sleep(5);
        wait(NULL);
}

int main(){
        int cpid, ppid;
        signal(SIGUSR1, handler);
        signal(SIGCHLD, handler);
        cpid = fork();
        switch(cpid){
                case -1:
                        cout<<"Error"<<endl;
                        break;
                case 0:
                        ppid =getppid();
                        kill(ppid, SIGUSR1);
                        exit(0);
                        break;
```

```
                default:
                        while(kill(cpid, 0)==0)
                                cout<<"waiting for child "<<endl;
                        cout<<"Child terminated. Now I am terminating"<<endl;
                        exit(0);
        }
        return 0;
}
```

## 57 MORE SYSTEM CALLS

1. **PAUSE**

```
int pause();
```

Suspend the calling process until any signal is received.

**EXAMPLE**

```
#include <iostream>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>
using namespace std;

void myhandler (int signum){
        cout<<"Processing a SIGINT signal "
                <<signum<<endl;
}

int main(){
        signal (SIGINT, myhandler);
        cout<<"Beginning execution -touch CTRL-C to continue program"
                <<endl;
        for(int i=0;i<5;i++){
                cout<<"Value of i is "<<i<<endl;
                pause();
        }
        return 0;
}
```

2. **RAISE**

```
int raise(int sig)
```

Sends the signal sig to the executing program. raise() actually uses kill() to send the signal to the executing program:

```
kill(getpid(), sig);
```

**EXAMPLE**

```
#include <iostream>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>
using namespace std;

int main(){
        cout<<"A suicidal program"<<endl;
        raise (SIGKILL);
        cout<<"Terminating..."<<endl;
        return 0;
}
```

3. **ALARM**

```
int alarm(unsigned int sec)
```

- alarm - set an alarm clock for delivery of a signal.
- alarm() arranges for a SIGALRM signal to be delivered to the calling process in sec seconds.
- If sec is zero, any pending alarm is canceled.

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
ISLAMABAD

OPERATING SYSTEMS LAB SPRING 2019

# Lab Manual 07
# Signals Set

## 58 WHAT IS SIGNALS SET?

A signal set is a collection of one or more signals just like we define a set in mathematics.

1. Need a data type to represent multiple signals.
2. The sigset_t data type is used to represent a signal set.

```
sigset_t myset
```

## 59 FUNCTIONS TO MANIPULATE SIGNAL SETS

Following are the functions to manipulate signal sets:

```
Sigemptyset(sigset_t *set)
```

- Empty a signal set
- e.g. sigemptyset(&mysignals);

```
Sigfillset(sigset_t *set)
```

- Initialize a signal set with all signals
- e.g. sigfillset(&mysignals);

```
Sigaddset(sigset_t *set, int signo)
```

- Add sig.no into the specified set
- e.g.sigaddset(&mysignals,SIGUSR1)

```
Sigdelset(sigset_t *set, int signo)
```

- Delete signo from the specified set
- e.g. sigdelset(&mysignals,SIGUSR1)

```
Sigismember(sigset_t *set, int signo)
```

- Returns True if sig.no is the member of the set specified in sigset_t.
- e.g. sigismember(&mysignals,SIGINT)

## 60 EXAMPLE: MANIPULATING SIGNAL SETS

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>

int main(){
        sigset_t sigset;
        // fill sigset with all available OS signals
        sigfillset(&sigset);
        /* Checking for membership*/
        if (sigismember(&sigset, SIGINT))
                printf(" SIGINT");
        if (sigismember(&sigset, SIGQUIT))
                printf(" SIGQUIT");
        if (sigismember(&sigset, SIGUSR1))
                printf(" SIGUSR1");
        if (sigismember(&sigset, SIGALRM))
                printf(" SIGALRM\n");

        /* Deleting SIGUSR1 from sigset */
        sigdelset(&sigset,SIGUSR1);
        if (sigismember(&sigset, SIGUSR1))
                printf("SIGUSR1\n");
        else
                printf("SIGUSR1 is Now not member of sigset \n");

             /* Empty the sigset */
        sigemptyset(&sigset);
        if (sigismember(&sigset, SIGALRM))
                printf(" SIGALRM");
        else
                printf("Sigset is empty\n");
```

```
        return 0;
}
```

## 61 SIGPROCMASK FUNCTION

Each process has a signal mask that defines the set of signals currently blocked from delivery to that process.
A process can:

1. Examine its signal mask
2. Change its signal mask
3. Perform both operations in one step by calling the following function:

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

### FUNCTION PARAMETERS

1. **const sigset_t *set**
   New signal mask of sigset_t type
2. **sigset_t *oset**
   Old signal mask of sigset_t type
3. **int how**
   (a) **SIG_BLOCK** The set of blocked signals is the union of the current set and the set argument.
   (b) **SIG_UNBLOCK** The signals in set are removed from the current set of blocked signals. It is legal to attempt to unblock a signal which is not blocked.
   (c) **SIG_SETMASK** The set of blocked signals is set to the argument set.

## 62 EXAMPLE: SIGPROCMASK FUNCTION

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include<unistd.h>
#include<sys/wait.h>
int main()
{
        // new_mask set to store different signals
         sigset_t new_mask;
         // Initilize and Empty new_mask set
         sigemptyset(&new_mask);
        // Adding SIGQUIT to set named as new_mask
         sigaddset(&new_mask,SIGQUIT);
```

```
        // Adding SIGTSTP to set named as new_mask
        sigaddset(&new_mask,SIGTSTP);
        // Adding SIGINT to set named as new_mask
        sigaddset(&new_mask,SIGINT);

        /*
                * Block All the signal present in new_mask set.
        */
        sigprocmask(SIG_BLOCK,&new_mask,NULL);
         int i=0;
        for(i;; i++){
                printf("\n %d \n",i);
                        printf("My PID is %d\n",getpid());
                        printf("you can't close me using conventional
                                                                commands\n");
                        sleep(2);
          }
        return 0;
}
```

## 63 SIGPENDING FUNCTION

```
int sigpending(sigset_t *set);
```

The sigpending function returns the set of signals that are blocked from delivery and currently pending for the calling process. The set of signals is returned through the set argument.

### EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
void sig_quit(int signo){
        printf("caught SIGQUIT\n");
        signal(SIGQUIT, SIG_DFL); //Reseting the signal handler for SIGQUIT
}
void sig_term(int signo){
        printf("caught SIGINT\n");
        signal(SIGINT, SIG_DFL); //Reseting the signal handler for SIGINT
}
int main(void){
        sigset_t newmask, oldmask, pendmask;
        signal(SIGQUIT, sig_quit);
        signal(SIGINT, sig_term);

        //Block SIGQUIT and save current signal mask.
        sigemptyset(&newmask);
        sigaddset(&newmask, SIGQUIT);
        sigaddset(&newmask, SIGINT);

        // apply "newmask set" as mask for process and
```

```
        store old mask to "oldmask set"
        sigprocmask(SIG_BLOCK, &newmask, &oldmask);
        printf("Sleep for 10 sec\n");
        sleep(10);

        // SIGQUIT && SIGINT here will remain pending
        sigpending(&pendmask);
        if (sigismember(&pendmask, SIGQUIT) && sigismember(&pendmask, SIGINT))
                printf("\nSIGQUIT && SIGINT pending\n");
        // Restore signal mask which unblocks SIGQUIT && SIGINT.
        //and This will unmask the process and deliver all pending sig
        sigprocmask(SIG_SETMASK, &oldmask, NULL) ;
        printf("SIGQUIT && SIGINT unblocked\n");

        while(1){}
        exit(0);
        /* SIGQUIT here will terminate with core file */
}
```

## 64 SIGSUSPEND FUNCTION

Suspend the process untill any signal is received except sigmask.

```
int sigsuspend(const sigset_t *sigmask);
```

1. The signal mask of the process is set to the value pointed to by sigmask.
2. Then the process is suspended until a signal is caught or until a signal occurs that terminates the process.
3. If a signal is caught and if the signal handler returns, then sigsuspend returns, and the signal mask of the process is set to its value before the call to sigsuspend.

### EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void sig_int(int signo){
        printf("\nin sig_int:\n ");
}

void sig_suspend(int signo){
        printf("\nin sig handler with %d number:\n ",signo);
}

int main(){
        sigset_t newmask, oldmask, waitmask;
        printf("program start: with pid %d \n",getpid());
        signal(SIGINT, sig_int);
```

```
        signal(SIGALRM, sig_suspend);
        signal(SIGUSR1, sig_suspend);
        sigemptyset(&waitmask);
        sigaddset(&waitmask, SIGUSR1);
        sigaddset(&waitmask, SIGALRM);
        sigemptyset(&newmask);
        sigaddset(&newmask, SIGINT);
        /*
                * Block SIGINT and save current signal mask.
        */

        sigprocmask(SIG_BLOCK, &newmask, &oldmask);

        /*
        * Critical region of code.
        */

        printf("in critical region: \n");

        /*
                * It will Pause untill received any signal
                except SIGUSR1 & SIGALRM.
                * newmask set will be suspended for that period of time
        */

        sigsuspend(&waitmask);

         // Now onwards old mask "newmask" is applicable
        printf("after return from sigsuspend: \n");
        sleep(15);
        /*
                        * unblocking newmask set.
        */
        sigprocmask(SIG_UNBLOCK, &newmask, NULL);

        /*
        * And continue processing ...
        */
        printf("program exit: \n");
        exit(0);
}
```

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
ISLAMABAD

OPERATING SYSTEMS LAB SPRING 2019

# Lab Manual 08
# Threads

## 65 WHAT IS A THREAD ?

"A thread is a sequence of control within a process"

To the software developer, the concept of a "procedure" that runs independently from its main program may best describe a thread.

## 66 PROPERTIES OF THREADS

Threads use and exist within process resources, yet are able to be scheduled by the operating system and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code.
This independent flow of control is accomplished because a thread maintains its own:

1. Stack pointer
2. Registers
3. Scheduling properties (such as policy or priority)
4. Set of pending and blocked signals
5. Thread specific data

## 67 SUMMARIZE THREADS

1. Exists within a process and uses the process resources.
2. Has its own independent flow of control as long as its parent process exists and the OS supports it.
3. Duplicates only the essential resources it needs to be independently schedulable.
4. May share the process resources with other threads that act equally independently (and dependently).
5. Dies if the parent process dies.
6. Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.
7. Threads do Not Share:
   (a) Thread ID
   (b) Set of registers, including program counter and stack pointer
   (c) Stack (for local variables and return addresses)
   (d) Signal mask
   (e) Priority

## 68 THREAD IMPLEMENTATION

In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).

Implementations adhering to this standard are referred to as POSIX threads, or **Pthreads**. Pthreads are defined as a set of C language programming types and procedure calls, implemented with a **pthread.h** header

To link this library to your program use

```
gcc -pthread -o a.out myprogram.c
```

## 69 THREAD CREATION

Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.

**pthread_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void*), void *arg);
```

**PTHREAD_CREATE ARGUMENTS:**

1. **thread:** unique identifier for the new thread returned by the subroutine.
2. **attr:** An attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
3. **start_routine:** The C routine that the thread will execute once it is created.
4. **arg:** A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

## 70 PTHREAD FUNCTIONS

### PTHREAD_SELF()

Return the thread ID of currently executing thread.

```
pthread_t pthread_self (void);
```

Returns the unique thread ID of the calling thread.

### PTHREAD_EXIT()

Used to terminate threads

```
pthread_exit(*status)
```

This is also the mechanism used to get a return value from a thread. Several ways of termination:

1. The thread returns from its starting routine (the main routine for the initial thread).
2. The thread makes a call to the pthread_exit subroutine.
3. The thread is canceled by another thread via the pthread_cancel routine.
4. The entire process (main) is terminated due to a call to the exit subroutines.

### EXAMPLE 01

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
pthread_t thread_id;      //creating object of type "pthread_t"
                                    //for storing thread id
void printids(const char *s){
        pid_t pid;
        pthread_t tid;
        pid = getpid();
        tid = pthread_self();
```

```
        printf("%s pid %lu tid %lu\n", s, (unsigned long)pid,
                                        (unsigned long)tid);
 }

void *thr_fn(void *arg){  // New thread will call this function/routine.
        printids("New thread: ");
         return(NULL);
 }

int main(void){
        int ret;
        /*create a thread,store its id in thread_id with
                                        default attrib by passing NULL*/
        /* Thread will immediately execute "thr_fn"
                                         function with NULL parameters */
        ret = pthread_create(&thread_id, NULL, thr_fn, NULL);
        if (ret != 0)
                printf("can't create thread\n");
        printids("Main thread:");
       /* Exit from main() will cause new thread to destroy,
                                        lets give it some time */
        sleep(1);
        exit(0);
}
```

## 71 JOINING THREADS

```
int pthread_join (pthread_t tid, void **status);
```

1. "Joining" is one way to accomplish synchronization between threads.
2. The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates.
3. The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit().
4. A joining thread can match one pthread_join() call. It is a logical error to attempt multiple joins on the same thread.

### EXAMPLE 02

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thr_fn1(void *arg){
        printf("Thread 1 exiting\n");
        void *status="Success";
        pthread_exit((void*)status);
}
```

```
void *thr_fn2(void *arg){
        printf("Thread 2 exiting\n");
        pthread_exit((void *)2);
}

int main(void){
        pthread_t tid1, tid2;
        pthread_create(&tid1, NULL, thr_fn1, NULL);
        pthread_create(&tid2, NULL, thr_fn2, NULL);
        void *tret1;
        void *tret2;
/*      why passing adress of pointer tret1 ? see Pthread_join prototype!! */
        pthread_join(tid1, &tret1);
        pthread_join(tid2, &tret2);
/*      casting tret1 to char*                                              */
        printf("Thread 1 exit code is: %s\n", (char*)tret1);
        printf("Thread 2 exit code is: %ld\n", (long)tret2);
        exit(0);
}
```

## 72 PASSING ARGUMENTS TO THREADS

The pthread_create() routine permits the programmer to pass one argument to the thread start routine.

For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the pthread_create() routine.

All arguments must be passed by reference and cast to (void *).

### EXAMPLE 03

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
/* this function is run by the second thread */
void *inc_x(void *arg){
        /* increment x to 100 */
        printf("Thread: Process Id is %d and Thread Id is %ld\n",getpid()
                                                        ,pthread_self());
        int *input;
        input=(int *)arg;
        while(++(*input) < 100);
        printf("x increment finished\n");
        /* the function must return something - NULL will do */
        pthread_exit(NULL);
        }
void main(){    // main Thread;
        int x = 0, y = 0;
```

```
        /* show the initial values of x and y */
        printf("x: %d, y: %d\n", x, y);
        printf("main: Process Id is %d and Thread Id is %ld\n",getpid()
                                                        ,pthread_self());
        /* this variable is our reference to the second thread */
        pthread_t thread_2;

        /* create a second thread which executes inc_x(&x) */
        (pthread_create(&thread_2, NULL, inc_x, &x));
        /* increment y to 100 in the first thread */
        while(++y < 100);
        printf("y increment finished\n");
         /* wait for the second thread to finish */
         (pthread_join(thread_2, NULL));

         /* show the results - x is now 100 thanks to the second thread */
         printf("x: %d, y: %d\n", x, y);

         pthread_exit(NULL);

}
```

**EXAMPLE 04**

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
struct student_database{
        int rollno;
        float cgpa;
};

struct student_database obj;

void *print_fn(void *threadarg){
        struct student_database *temp_data;
        temp_data = (struct student_database *) threadarg;
        int rollno= temp_data->rollno;
        float cgpa = temp_data->cgpa;
        printf("Inside Thread\n");
        printf("Roll no %d has cgpa :%f\n",rollno,cgpa);
        pthread_exit(NULL);
}

int main(int argc, char *argv[]){
        pthread_t thread_id;
        obj.rollno=90010;
        obj.cgpa=3.88;
        printf("Creating thread\n");
        pthread_create(&thread_id, NULL, print_fn, &obj);
        pthread_exit(NULL);
}
```

## 73 EXAMPLES

### EXAMPLE 05

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

pthread_t thread_id1,thread_id2;

void *thr_fn1(void *threadid){
        sleep(1);
        printf("thread 1: #%ld! and Iam not waiting\n",pthread_self());
        int i=0;
        for(i;i<5;i++){printf("value of i is %d\n",i);}
        pthread_exit(NULL);
}

void *thr_fn2(void *threadid){
        printf("thread 2: #%ld! waiting for thread 1\n",pthread_self());
        pthread_join(thread_id1,NULL);
        printf("thread 2: thread 1 exited and now me exiting \n");
        pthread_exit(NULL);
}

int main(int argc, char *argv[]){
        printf("main: creating thread\n");
        pthread_create(&thread_id1, NULL, thr_fn1,NULL);
        pthread_create(&thread_id2, NULL, thr_fn2,NULL);
        pthread_join(thread_id2,NULL);
        printf("main:Both threads exited\n");
        pthread_exit(NULL);
}
```

### EXAMPLE 06

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid){
        long *tid;
        tid = (long *)threadid;
        printf("It's me, thread #%ld\n", *tid);
        pthread_exit(NULL);
}

int main (int argc, char *argv[]){
        pthread_t threads[NUM_THREADS];
        long t;
        for(t=0; t<NUM_THREADS; t++){
                printf("In main: creating thread %ld\n", t);
                pthread_create(&threads[t], NULL, PrintHello, (void *)&t);
                sleep(1);
```

```
        }

        /* Last thing that main() should do */
        pthread_exit(NULL);
}
```

**EXAMPLE 07**

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
char *message;
struct thread_data
{
   pthread_t *thread_id;
   char *message;
};
struct thread_data thread_data;

void *thr_fn(void *threadarg){
        struct thread_data *my_data;
        my_data = (struct thread_data *)threadarg;
        pthread_t *taskid = my_data->thread_id;
        char *hello_msg= my_data->message;
        printf("Thread id is: %ld \nMessage: %s", *taskid, hello_msg);
        pthread_exit((void *)"Successfull");
}

int main(int argc, char *argv[]){
        pthread_t thread_id1;
        message = "Have you learn something today ?\n";
        thread_data.thread_id =&thread_id1;
        thread_data.message = message;
        printf("Creating thread\n");
        pthread_create(&thread_id1, NULL, thr_fn, (void *) &thread_data);
        void * status;
        pthread_join(thread_id1,&status);
        printf("Thread Exited with status: %s\n",(char*)status);
        exit(0);
}
```

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
ISLAMABAD

OPERATING SYSTEMS LAB SPRING 2019

# Lab Manual 09
# Thread Attributes

## 74 THREAD STATES

Threads are created in two ways

1. Joinable Threads
2. Detached Threads

## 75 JOINABLE OR NOT?

1. When a thread is created, one of its attributes defines whether it is joinable or detached.
2. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
3. When we create a thread with default attributes a joinable thread is created.

## 76 PTHREAD FUNCTIONS

### PTHREAD_SELF()

Return the thread ID of currently executing thread.

```
pthread_t pthread_self (void);
```

Returns the unique thread ID of the calling thread.

### PTHREAD_EXIT()

Used to terminate threads

```
pthread_exit(*status)
```

This is also the mechanism used to get a return value from a thread. Several ways of termination:

1. The thread returns from its starting routine (the main routine for the initial thread).
2. The thread makes a call to the pthread_exit subroutine.
3. The thread is canceled by another thread via the pthread_cancel routine.
4. The entire process (main) is terminated due to a call to the exit subroutines.

## 77 JOINING THREADS

```
int pthread_join (pthread_t tid, void **status);
```

1. "Joining" is one way to accomplish synchronization between threads.
2. The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates.
3. The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit().
4. A joining thread can match one pthread_join() call. It is a logical error to attempt multiple joins on the same thread.

### EXAMPLE 0

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thr_fn1(void *arg){
        printf("Thread 1 exiting\n");
        void *status="Success";
        pthread_exit((void*)status);
}

void *thr_fn2(void *arg){
        printf("Thread 2 exiting\n");
        pthread_exit((void *)2);
}

int main(void){
        pthread_t tid1, tid2;
        pthread_create(&tid1, NULL, thr_fn1, NULL);
        pthread_create(&tid2, NULL, thr_fn2, NULL);
```

```
        void *tret1;
        void *tret2;
/*      why passing adress of pointer tret1 ? see Pthread_join prototype!! */
        pthread_join(tid1, &tret1);
        pthread_join(tid2, &tret2);
/*      casting tret1 to char*                                              */
        printf("Thread 1 exit code is: %s\n", (char*)tret1);
        printf("Thread 2 exit code is: %ld\n", (long)tret2);
        exit(0);
}
```

## 78 THREAD DETACHING

1. By default threads are created joinable.
2. Instead of waiting for a thread, the executing thread can specify that
   (a) It does not require a return value.
   (b) Or any explicit synchronization with that thread.
3. The pthread_detach() routine can be used to explicitly detach a thread even though it was created as joinable.
4. After this call, no thread can wait for detached thread and it executes independently until termination.

```
int pthread_detach(pthread_t tid);
```

5. There is no converse routine.
6. Threads can detach themselves by calling pthread_detach with an argument of pthread_self().

### EXAMPLE 01

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void * thread_function(void *arg);
char message[] = "iam thread 1";
int main(){
        pthread_t tid;
        pthread_create(&tid,NULL,thread_function,(void*)message);
        // In 2nd run make following statement comment then see
        pthread_detach(tid);
        int joinret;
        joinret=pthread_join(tid,NULL);

        if(joinret==0)   //when join successfull
                printf("join was successfull:The main thread was
                                        waiting for thread 1\n");
```

```
        else
                printf("Join failed:The main thread is not
                                        waiting for thread 1\n");
        printf("The main thread finished, bye!\n");
        pthread_exit(NULL);
}

void * thread_function(void *arg){
        sleep(2);
        printf("thread_function is running. Argument was\" %s\"\n",
                                        (char *)arg);
        printf("thread 1 awaked from sleep, and exiting now\n");
        pthread_exit(NULL);
}
```

**EXAMPLE 02**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void * thread_function1(void *arg);
void * thread_function2(void *arg);
char message1[] = "iam thread 1";char message2[] = "iam thread 2";
pthread_t tid1,tid2;
int main(){
        pthread_create(&tid1,NULL,thread_function1,(void*)message1);
        pthread_create(&tid2,NULL,thread_function2,(void*)message2);
        pthread_detach(tid1);
        int joinret;
        joinret=pthread_join(tid1,NULL);

         //when join successfull
        if(joinret==0)
           printf("join was successfull:The main thread was waiting for thread 1\n");
        else
           printf("Join failed:The main thread is not waiting for thread 1\n");

        pthread_join(tid2,NULL);
        printf("The main received exit status from tid2,now exiting, bye!\n");
        pthread_exit(NULL);
}

void * thread_function1(void *arg){
        sleep(2);
        printf("thread_function 1 is running. Argument was %s\n", (char *)arg);
        sleep(5);
        printf("thread 1 awaked from sleep, and exiting now\n");
        pthread_exit(NULL);
}

void * thread_function2(void *arg){
        sleep(1);
        int joinret;
        joinret=pthread_join(tid1,NULL);
```

```
        if(joinret==0)   //when join successfull
           printf("join was successfull:The thread 2 was waiting for thread 1\n");
        else
           printf("join failed:The thread 2 is not waiting for thread 1\n");
        printf("thread_function 2 is running. Argument was %s\n", (char *)arg);
        printf("thread 2 is exiting now\n");
        pthread_exit(NULL);
}
```

## 79 THREAD ATTRIBUTES

The pthread interface allows us to fine-tune the behavior of threads using the pthread_attr_t structure by modifying the default attributes.

1. An initialization function exists to set the attributes to their default values.
2. Another function exists to destroy the attributes object. If the initialization function allocated any resources associated with the attributes object, the destroy function frees those resources.
3. Each attribute has a function to get the value of the attribute from the attribute object.
4. Each attribute has a function to set the value of the attribute. In this case, the value is passed as an argument, by value.

## 80 THREAD DETACHING USING ATTRIBUTES

To explicitly create a thread as joinable or detached, the attr argument in the pthread_create() routine is used. The typical 4 step process is:

1. Declare a pthread attribute variable of the pthread_attr_t data type.
2. Initialize the attribute variable with pthread_attr_init().
3. Set the attribute detached status with pthread_attr_setdetachstate().
4. When done, free library resources used by the attribute with pthread_attr_destroy().

### IMPORTANT POINTS

- A detach thread can also be created using thread attributes.

```
pthread_attr_t thread_attr;
```

- Calling pthread_attr_init, the pthread_attr_t structure contains the default values for all the thread attributes supported by the implementation.

```
pthread_attr_init(&thread_attr);
```

- detachedstate: This attribute allows us to avoid the need for threads to rejoin.

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

- The two possible flag values for pthread_attr_setdetachstate are PTHREAD_CREATE_JOINABLE and PTHREAD_CREATE_DETACHED.

## 81 PTHREAD_ATTR_DESTROY()

```
pthread_attr_destroy(pthread_attr_t *attr)
```

1. To deinitialize a pthread_attr_t structure, we call pthread_attr_destroy.
2. If an implementation of pthread_attr_init allocated any dynamic memory for the attribute object, pthread_attr_destroy will free that memory
3. pthread_attr_destroy will initialize the attribute object with invalid values, so if it is used by mistake, pthread_create will return an error code.

### EXAMPLE 03

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void * thread_function(void *arg);
char message[] = "Iam thread 1";
int main(){
        pthread_t tid;
        pthread_attr_t thread_attr;
        pthread_attr_init(&thread_attr);
        //comment following line in 2nd run
        pthread_attr_setdetachstate(&thread_attr,PTHREAD_CREATE_DETACHED);
        //uncomment following line in 2nd run
        //pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_JOINABLE);
        pthread_create(&tid,&thread_attr,thread_function,(void*)message);
        pthread_attr_destroy(&thread_attr);
        int joinret;
        joinret=pthread_join(tid,NULL);

        if(joinret==0)   //when join successfull
           printf("join was successfull:The main thread was waiting for thread 1\n");
        else
           printf("Join failed:The main thread is not waiting for thread 1\n");
        printf("The main thread finished, bye!\n");
        pthread_exit(NULL);
}

void * thread_function(void *arg){
        printf("thread_function is running. Argument was\" %s\"\n", (char *)arg);
        sleep(4);
        printf("thread awaked from sleep, and exiting now\n");
        pthread_exit(NULL);
}
```

## 82 Some Other Thread Attributes

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
int pthread_attr_setstacksize(pthread_attr_t *attr, int scope);
int pthread_attr_getstacksize(const pthread_attr_t *attr, int *scope);
```

## 83 Thread Cancelation

1. Sometimes, we want one thread to be able to ask another thread to terminate, rather like sending it a signal. There is a way to do this with threads using pthread_cancel() system call.
2. One thread can request that another in the same process be canceled by calling the pthread_cancel function.

   ```
   pthread_cancel(pthread_t tid)
   ```

3. In the default circumstances, pthread_cancel will cause the thread specified by **tid** to behave as if it had called **pthread_exit** with an argument of PTHREAD_CANCELED.

## 84 Thread Cancelation Options

A thread can change its cancelability state by calling pthread_setcancelstate.

```
pthread_setcancelstate(int state, int *oldstate);
```

The first parameter is either:

1. **PTHREAD_CANCEL_ENABLE** which allows it to receive cancel requests.
2. **PTHREAD_CANCEL_DISABLE** which causes them to be ignored. The oldstate pointer allows the previous state to be retrieved. If you are not interested, you can simply pass NULL.

### Important Points

- A thread starts with a default cancelability state of PTHREAD_CANCEL_ENABLE.
- When the state is set to PTHREAD_CANCEL_DISABLE, a call to pthread_cancel will not kill the thread.
- The cancellation request remains pending for the thread.When the state is enabled again, the thread will act on any pending cancellation requests at the next cancellation point.

## 85 THREAD CANCELATION TYPES

If cancel requests are accepted, there is a second level of control the thread can take, the cancel type, which is set with pthread_setcanceltype.

```
int pthread_setcanceltype(int type, int *oldtype);
```

The type can take one of two values:

1. **PTHREAD_CANCEL_ASYNCHRONOUS** which causes cancellation requests to be acted upon immediately.
2. **PTHREAD_CANCEL_DEFERRED** which makes cancellation requests wait until the thread executes one of these functions: pthread_join, pthread_cond_wait, pthread_cond_timedwait, sem_wait etc.

By default, threads start with the *cancellation state* as **PTHREAD_CANCEL_ENABLE** and the *cancellation type* as **PTHREAD_CANCEL_DEFERRED**.

**EXAMPLE 04**

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void *thread_function1(void *arg);
pthread_t tid1;
int main(){
        pthread_create(&tid1, NULL,thread_function1, NULL);
        sleep(2);
        printf("Main thread:Canceling thread 1 ...\n");
        pthread_cancel(tid1);
        printf("Main thread: exiting\n");
        pthread_exit(NULL);
}

void * thread_function1(void *arg){
        int i,oldtype,oldstate;
        // comment following line in 2nd run
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,&oldstate);
        // uncomment following line in 2nd run
        //pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,&oldtype);
        printf("Thread 1:iam running and my old cancel state was %d\n",oldstate);
        for(i = 0; i < 10; i++) {
           printf("Thread 1:iam still running (%d)...\n",i);
           sleep(2);
        }
        pthread_exit(0);
}
```

**EXAMPLE 05**

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void *thread_function1(void *arg);
```

```
void *thread_function2(void *arg);
pthread_t tid1,tid2;
int main(){
        pthread_create(&tid1, NULL,thread_function1, NULL);
        pthread_create(&tid2, NULL,thread_function2, NULL);
        sleep(1);
        printf("Main thread:Canceling thread 1 ...\n");
        pthread_cancel(tid1);
        printf("Main thread: exiting\n");
        pthread_exit(NULL);
}

void * thread_function1(void *arg){
        int i,oldstate,oldtype;;
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,&oldstate);
        pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,&oldtype);
        //pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,&oldtype);
        //sleep(1);
        printf("Thread 1:my old cancel state was %d and old cancel type
                                                was %d\n",oldstate,oldtype);
        for(i = 0; i < 5; i++) {
        printf("Thread 1:iam running (%d)\n",i);
        if (i=1){
         printf("Thread 1: iam waiting for thread 2 to finish\n");
         pthread_join(tid2, NULL);}
        }
        // Following statement will never run!!!!!
        printf("Thread 1:Thread 1 exited, now me aswell\n");
        pthread_exit(0);
}

void * thread_function2(void *arg){
        int i;
        sleep(2);
        for(i = 0; i < 5; i++) {
           printf("Thread 2:iam running (%d)\n",i);
           printf("Thread 2:thread 1 received cancel signal but is
                                        waiting for me to finish\n");
        }
        printf("Thread 2:iam exiting\n");
        pthread_exit(0);
}
```

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
ISLAMABAD

OPERATING SYSTEMS LAB SPRING 2019

# Lab Manual 10
# MUTEX

## 86 WHAT IS A MUTEX?

1. Mutex is an abbreviation for “mutual exclusion”
2. It is a special variable which act as a mutual exclusion device to protect sections of code (hence the name mutex)
3. Mutex variables are one of the primary means of implementing thread synchronization

## 87 MUTEX VARIABLES

1. A mutex variable acts like a "lock" protecting access to a shared data resource (variables, section of code e.t.c).
2. It can be either in:
   (a) **Locked state**: a distinguished thread that holds or owns the mutex, or
   (b) **Unlocked state**: no thread holds the mutex
3. The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time..
4. If several threads try to lock a mutex only one thread will be successful. *The rest block at that call*.

5. No other thread can own that mutex until the owning thread unlocks that mutex. Threads must "take turns" accessing protected data.

## 88 MUTEX SYSTEM CALLS

```
pthread_mutex_init()
pthread_mutex_destroy()
pthread_mutex_lock()
pthread_mutex_trylock()
pthread_mutex_unlock()
```

## 89 MUTEX VARIABLES

A typical sequence in the use of a mutex is as follows:

1. Create and initialize a mutex variable
2. Several threads attempt to lock the mutex
3. Only one succeeds and that thread owns the mutex
4. The owner thread performs some set of actions
5. The owner unlocks the mutex
6. Another thread acquires the mutex and repeats the process
7. Finally the mutex is destroyed

**THE IDEA:**

```
lock the mutex
        critical section
unlock the mutex
```

## 90 USING MUTEX

1. **Declare** an object of type pthread_mutex_t.
2. **Initialize** the object by calling pthread_mutex_init().
3. Call pthread_mutex_lock() to **gain exclusive access** to the shared data object.
4. Call pthread_mutex_unlock() to **release the exclusive access** and allow another thread to use the shared data object.
5. **Get rid of the object** by calling pthread_mutex_destroy().

## 91 MUTEX SYSTEM CALLS EXPLAINED

1. **Initialization:** Mutex variables must be declared with type pthread_mutex_t and must be initialized before they can be used.

   (a) Statically it is declared as:
   pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;

   (b) Dynamically,with the pthread_mutex_init() routine. This methos allows to set mutex attributes, may be specified as NULL to accept defaults

   **The mutex is initially unlocked**.

2. **pthread_mutex_destroy()** should be used to free a mutex object which is no longer needed.

3. **pthread_mutex_lock()** routine is used by a thread to acquire a lock on the specified mutex variable.

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

4. **pthread_mutex_trylock()** will attempt to lock a mutex. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

   If the mutex is already locked, the routine will return immediately with a "busy" error code

5. **pthread_mutex_unlock()** will unlock a mutex if called by the owning thread.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

   Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data.

   An error will be returned:

   (a) If the mutex was already unlocked.

   (b) If the mutex is owned by another thread.

## 92 THINGS TO AVOID WHILE USING MUTEX

1. No thread should attempt to lock or unlock a mutex that has not been initialized.

2. The thread that locks a mutex must be the thread that unlocks it.

3. No thread should have the mutex locked when you destroy the mutex.
4. Any mutex that is initialized should eventually be destroyed, but only after any thread that uses it has either terminated or is no longer interesting in using it.

## 93 EXAMPLES

### EXAMPLE 01

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *functionC(void *);
//pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex1;
pthread_t thread1, thread2;
int counter = 0;

int main() {

        pthread_mutex_init(&mutex1,NULL);
        pthread_create(&thread1,NULL,&functionC,NULL);
        pthread_create(&thread2,NULL,&functionC,NULL);

        printf("Thread 1 ID: %ld \n",thread1);
        printf("Thread 2 ID: %ld \n",thread2);
        sleep(3);
       // pthread_join( thread1, NULL);
       // pthread_join( thread2, NULL);
        pthread_mutex_destroy(&mutex1);
        pthread_exit(NULL);
        exit(0);
}

void * functionC(void * p){

        pthread_mutex_lock( &mutex1 );
        counter++;

        printf("Thread %ld Counter value: %d\n",pthread_self(),counter);
        pthread_mutex_unlock( &mutex1 );
        pthread_exit(NULL);
}
```

### EXAMPLE 02

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *functionA(void *);
void *functionB(void *);
//pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex1;
pthread_t thread1, thread2;
```

```
int counter = 0;
int main() {
        pthread_mutex_init(&mutex1,NULL);
        pthread_create(&thread1,NULL,&functionA,NULL);
        pthread_create(&thread2,NULL,&functionB,NULL);
        printf("Thread 1 ID: %ld \n",thread1);
        printf("Thread 2 ID: %ld \n",thread2);
        pthread_join( thread1, NULL);
        pthread_join( thread2, NULL);
        pthread_mutex_destroy(&mutex1);
        pthread_exit(NULL);
        exit(0);
}
void * functionA(void * p){
        int a;
        pthread_mutex_lock( &mutex1 );
        a=counter;a--;counter=a;
        printf("Thread 1 ID: %ld Counter value: %d\n",pthread_self(),counter);
        pthread_mutex_unlock( &mutex1 );
        pthread_exit(NULL);
}

void * functionB(void * p){
        int b;
        pthread_mutex_lock( &mutex1 );
        b=counter;b++;counter=b;
        printf("Thread 2 ID: %ld Counter value: %d\n",pthread_self(),counter);
        pthread_mutex_unlock( &mutex1 );
        pthread_exit(NULL);
}
```

**EXAMPLE 03**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *functionA(void *);
void *functionB(void *);
pthread_mutex_t mutex1;
pthread_t thread1, thread2;
int counter = 0;
int main() {
        pthread_mutex_init(&mutex1,NULL);
        pthread_create(&thread1,NULL,&functionA,NULL);
        pthread_create(&thread2,NULL,&functionB,NULL);
        printf("Thread %ld \n",thread1);
        printf("Thread %ld \n",thread2);
        pthread_join( thread1, NULL);
        pthread_join( thread2, NULL);
        pthread_mutex_destroy(&mutex1);
        pthread_exit(NULL);
        exit(0);
}
void * functionA(void * p){
        int a;
```

```
        pthread_mutex_lock( &mutex1 );
        a=counter;a--;counter=a;
        sleep(1);
        printf("Thread %ld Counter value: %d\n",pthread_self(),counter);
        pthread_mutex_unlock( &mutex1 );
        pthread_exit(NULL);
}

void * functionB(void * p){
        int b;int mycount=0;
       while(pthread_mutex_trylock( &mutex1 )!=0){
            while(mycount<=800000){
              mycount++;
           }
            mycount=0;
            printf("Trying to own lock\n");
        }
        b=counter;b++;counter=b;
        printf("Thread %ld Counter value: %d\n",pthread_self(),counter);
        pthread_mutex_unlock( &mutex1 );
        pthread_exit(NULL);
}
```

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
ISLAMABAD

OPERATING SYSTEMS LAB SPRING 2019

# Lab Manual 11
# SEMAPHORES

## 94 WHAT IS A SEMAPHORE ?

1. Dijkestra proposed a significant technique for managing concurrent processes for complex mutual exclusion problems. He introduced a new synchronization tool called Semaphore.
2. Posix Semaphores are Inter Process or Threads synchronization technique, just like mutex.
3. A semaphore is an integer value variable, which can be incremented and unlocked or decremented and locked by threads or processes.

## 95 POSIX TYPES OF SEMAPHORE

POSIX Semaphores are of 2 types:

1. Named Semaphore
2. Memory-mapped Semaphore

## 96 UNNAMED SEMAPHORES (MEMORY-BASED SEMAPHORES)

1. No name is associated with these semaphores
2. Provides synchronization between threads and between related processes

3. Placed in a region of main memory that is shared between processes/threads
4. For threads this is done by simply making the semaphore a global variable

## 97 KINDS OF SEMAPHORES

Depending upon the value a semaphore is made to hold, it can be:

1. Binary semaphore
2. Counting semaphore

## 98 USES OF SEMAPHORES

Semaphores can be use for synchronization between Threads/Processes. It also provide a way to avoid Dead-locks.

1. **For placing Locks**
   Just like mutex ( can me binary or counting).
   (a) Counting Semaphores:
   Permit a limited number of threads to execute a section of the code.
   (b) Binary Semaphores:
   Permit only one thread to execute a section of the code.
2. **Semaphores As Condition Variables**
   (a) Semaphores are also useful when a thread wants to halt its progress waiting for a condition to become true.
   (b) Communicate information about the state of shared data.

## 99 SEMAPHORES VS MUTEX

1. Mutex can be locked or unlocked, like binary semaphore. Semaphores (counting) can have multiple values.
2. A locked mutex can be unlocked by the thread holding the lock. A locked semaphore can be unlocked by any thread.
3. Semaphore has state (value of semaphore) associated with it.
4. Mutex and condition variables are used together in most scenario. Looking at their functionality, it can be thought as : Semaphore = Mutex + Condition Variable
5. Posix Named Semaphore are kernel persistent. Posix Memory based semaphore, Posix Condition Variable and Posix Mutex are process persistent.

## 100 SEMAPHORE SYSTEM CALLS

```
#include <semaphore.h>

int sem_init();

int sem_wait();

int sem_trywait();

int sem_post();

int sem_destroy();
```

## 101 CREATE A SEMAPHORE

```
int sem_init(sem_t *sem, int pshared, unsigned int value)
```

1. **sem:** Target semaphore
2. **pshared:** The pshared argument indicates whether this semaphore is to be shared-between the threads of a process, or between processes.
   (a) **0:** only threads of the creating process can use the semaphore.
   (b) **Non-0:** other processes can use the semaphore.
3. **value:** Initial value of the semaphore.

**EXAMPLE:**

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1)
```

We declare a semaphore s and initialize it to the value 1 by passing 1 in as the third argument. The second argument to sem init() will be set to 0 in all of the examples we'll see; this indicates t hat the semaphore is shared between threads in the same process.

## 102 SEMAPHORE OPERATIONS

1. **sem_wait()** decrements (locks) the semaphore pointed to by sem.

   ```
   int sem_wait(sem_t *sem)
   ```

   (a) If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns( gets lock), immediately.

(b) If the value of the semaphore is negative, the calling thread blocks; one of the blocked threads wakes up when another thread calls sem_post()

2. **sem_post()** does not wait for some particular condition to hold like sem_wait() does.

```
int sem_post(sem_t *sem)
```

Rather, it simply increments the value of the semaphore and then, if there is a thread waiting to be woken, wakes one of them up.

```
int sem_wait(sem_t*s) {
        decrement the value of semaphore s by one
        wait if value of semaphore s is negative
}

int sem_post(sem_t*s)
        increment the value of semaphore s by one
        if there are one or more threads waiting, wake one
}
```

3. **sem_trywait()** is the version of of the sem_wait() which does not block.

```
int sem_trywait(sem_t * sem)
```

Decreases the semaphore by one if the semaphore does not equal to zero. If it is zero it does not block, returns zero with error code EAGAIN.

4. **sem_destroy** releases the resources that semaphore has and destroys it

```
int sem_destroy(sem_t * sem)
```

## 103 EXAMPLES

### EXAMPLE 01: SEMAPHORE A BINARY LOCK

```
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>
#include<pthread.h>
#include <semaphore.h>
#include <unistd.h>

// Using semaphore as a Binary lock for IPC/Syncronization

sem_t sem;
void * f1(void *arg){
        sem_wait(&sem);
        printf("Thread with ID:%ld got lock\n",pthread_self());
        sleep(2);
        sem_post(&sem);
        pthread_exit(NULL);
```

```
}

int main(){
        pthread_t tid1,tid2;
        sem_init(&sem,0,1);
        pthread_create(&tid1,NULL,f1,NULL);
        pthread_create(&tid2,NULL,f1,NULL);
        pthread_exit(NULL);
}
```

**EXAMPLE 02: SEMAPHORE A COUNTING VARIABLE**

```
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>
#include<pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t sem;
void * f1(void *arg)
{
        int i=1,value;
        sem_wait(&sem);
        sem_getvalue(&sem, &value);
        printf("Thread with ID %ld return from wait() and
                                Sem state is :%d\n",pthread_self(),value);
        pthread_exit(0);
}
int main()
{
        pthread_t tid[5];
        int value,i;
        sem_init(&sem,0,3);
        for(i=0;i<5;i++){
                pthread_create(&tid[i],NULL,f1,NULL);
        }

        sleep(3);
        printf("Main:Going to Post\n");
        sem_post(&sem);
        sem_getvalue(&sem, &value);
        printf("Main:After Post sem state:%d \n",value);
        sleep(3);
        printf("Main:Going to Post\n");
        sem_post(&sem);
        sem_getvalue(&sem, &value);
        printf("Main:After Post sem state:%d \n",value);

        pthread_exit(0);

}
```

**EXAMPLE 03: SEMAPHORE A CONDITIONAL VARIABLE**

```
#include<stdio.h>
```

```
#include<stdlib.h>
#include<signal.h>
#include<pthread.h>
#include <semaphore.h>
#include <unistd.h>

// Using semaphore as a condtion variable for IPC/Syncronization

sem_t sem;
void * f1(void *arg){
        printf("Child:  begin\n");
        sem_post(&sem);
        printf("Child:  end\n");
        pthread_exit(NULL);
}

int main(){
        pthread_t tid;
        sem_init(&sem,0,0);
        printf("parent: begin\n");
        pthread_create(&tid,NULL,f1,NULL);
        sem_wait(&sem);
        printf("parent: end\n");
        pthread_exit(NULL);

}
```