Operating System Lab

Lab - 03

Objectives:

1. Understanding the C compilation process.
2. Understanding the Command Line Arguments
3. Understanding the Environment Variables
4. Understanding the fork and exec syscalls
5. Getting familiar with simple attributes of binaries

Compilation Process

Task 01:

Write a C program to print "Learning is Fun with Arif Butt" on stdout

a) Create a **preprocessed** code file of your source C program.
b) Create an **assembly** code file of your source C program.
c) Create an **object** file of your source C program.
d) Create an **executable** file of your source C program.

Note:

- *You must have a clear understanding about each file generated during the compilation process.*
- *Mention the file formats created during the task.*

Extracting Information from Binaries

Task 01: Compile and Link the Given Program **Statically** and **Dynamically**. Give each binary a different name.

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<fcntl.h>
int
main(int argc, char *argv[]) {
    write(1, "Hi, I am your own cat ;)\n", 31);
    int fd = 0;
    if(argc > 1)
    {
        fd = open(argv[1], O_RDONLY)
    }
     int rv = 0;
    char buff[1024];
    while(rv = read(fd, buff, 1024))
    {
        write(1, buff, rv);
    }
    return 0;
}
```

The above program mimics the general behavior of “`cat`”. “`strace`” both of the statically linked and dynamically linked **“`cat`”.** What difference do you see? Pass each of the binary **`/proc/self/maps`** as arguments and see if they both print the same or different result. Can you guess what it is printing ?

Note : *Don't worry about /proc/self/maps right now*

Task 02:

a) Display the “**disassembly**” of the executable in intel format.
b) Display “**Section Headers”** of the executable and note down the **count** of section headers.
c) Display the **“Program Headers”**
d) Display the **“ELF header”**.

Task 03:

Perform the following task on **`cat.c`**

a) Use `readelf(1)`, `od(1)`, `size(1)` commands to get different attributes of your executable program file. Compile your program file with `-g` option to `gcc` and run the commands again to see the differences. Compile your program file with `--static` option to `gcc` and run the commands again to see the differences
b) Perform strace on both **Statically** and **Dynamically** Linked binaries created in step (a)

Process Management

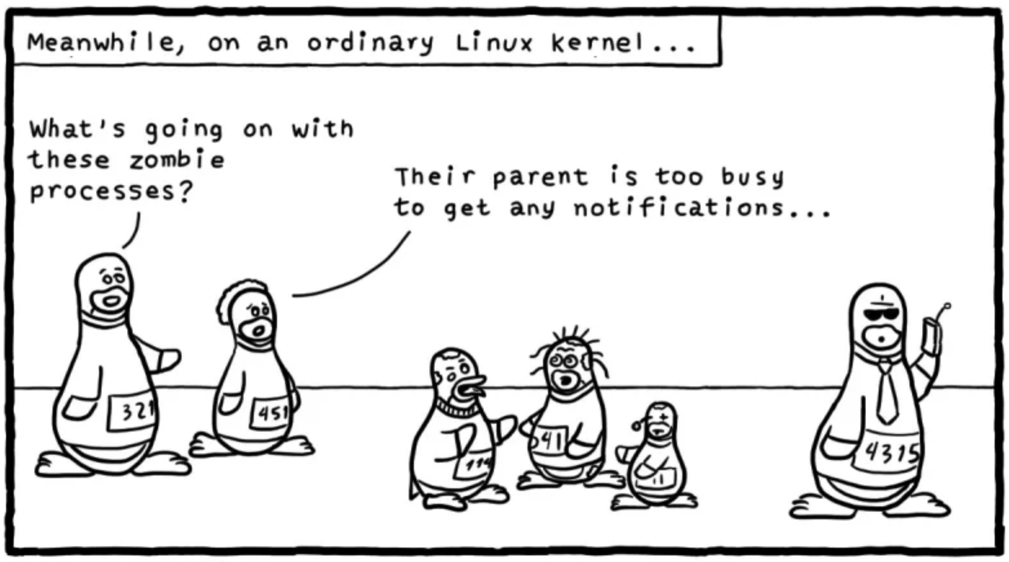
Task 01:

a) Write a command to show currently running jobs in your terminal. How can you get information about all the processes running on your linux OS?
b) Write a command to bring a process to the foreground which is currently running in background.
c) Write a command to get information about how long the system has been running.
d) Display the free and used memory on your system.

Task 02: Run the sleep command for 50 sec, suspend its execution and then run it in the background. Execute the program in the background, and then bring it to the foreground. Meanwhile in another terminal keep checking various statistics of your process using `ps(1)` with `-u` and `-l` options.

Zombie Processes

Task 01:
Zombie Process is a process that has terminated but its parent has not collected its exit status and has not reaped it. So a parent must reap its children .When a process terminates but still is holding system resources like PCB and various tables maintained by OS. It is half-alive & half-dead because it is holding resources like memory but it is never scheduled on the CPU .
Zombies can't be killed by a signal, not even with the silver bullet (SIGKILL). The only way to remove them from the system is to kill their parents, at which time they become orphan and adopted by init or systemd.


Meanwhile, on an ordinary Linux kernel...
What's going on with these zombie processes?
Their parent is too busy to get any notifications...
321
451
114
541
4315

Looking at above definition of zombie process:

a) Write down a C Program that will create a **zombie** process.
b) How can you verify that the process has become a zombie?

Concept Check

Task 01:
What will be the output of this C Program :

```
int main(){
  for (int i=1;i<=4;i++){
    fork();
    fprintf(stderr, "%s\n","ARIF");
  }
  exit(0);
}
```

Task 02:

What will be the output of this C Program :

```
if (fork() || fork())   fork();
printf("1 ");
```

Task 03:

What will be the output of the following C program

```
int cpid;
cpid = fork();
switch(cpid)
{
      case 0:
            printf(“I am not a zombie process”);
            break;
      default:
            while(1);
}
```

What is the output of the above code? Does the parent reap the child successfully or the child becomes a **zombie** or **orphan**? Provide proof to whatever you think is the case.

Task 04:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int cake = 11;
int another[3];
int calendar[4] = {1, 2, 3, 4};
int main(int argc, char **argv, char **envp)
{
    int x[5] = {1, 2, 3, 4, 5};
    int hello = 0x10;
    int *writer = (int *)malloc(sizeof(int) * 10);
    return 0;
}
```

Consider the above program:

a) Where in memory the variable **`cake`** will be created.
b) Where in memory **`another`** array will be created.
c) What about the **`calendar`**? It seems like it is initialized.
d) What about the variable **`hello`** ?
e) What do you think about the array **`x`**.
f) What about the **`writer`** and wow what about those parameters of main function?

Bonus Tasks

You Can Solve these tasks as your home task.

Cmd Line Args & Environment Variables & Fork & Exec

Task 01:

a) Write a command to display the **`HOSTNAME`** variable.
b) Change the **`HOSTNAME`** environment variable to “`Miffy`”.
c) Write a C program to print all the environment variables on stdout.
d) Write a command to show all the environment variables on the terminal.
e) Write a C Program to take 2 numbers from the command line and display their sum on stdout.
f) Write a C program to print the name of the program on stdout.
g) What does the export command do ?

Task 02:

Write a c program and perform the following tasks.

a) Your program should take exactly one command line argument
b) Do fork and run the given cat program using execve syscall in the child process with user argument.
c) Your parent process should wait for the child process.
d) Once the child process is terminated, print the pid of the terminated child process.

Task 03:

```
#include<stdio.h>
#include<math.h>
int main(int argc, char **argv, char **envp)
{
    for(int idx = 0; envp[idx] != NULL; idx++)
    {
        printf("%s\n", envp[idx]);
    }
    return 0;
}
```

Run the given program and see what it prints.

1. Write a C program which runs the above program as its child process.
2. Can you make the above program print out nothing? if yes what it shows.
3. Can you make the program output of your choice without editing the given process?

Run the Program by giving it an argument to env modify its output using `env`.

Note: *Do not make changes in the given program.*