# 

# **Lab objectives**

* Students understand that there can be different manpages and how to access them
* Students can use ps to list and analyze running processes
* Students are able to understand the mechanism for process creation in LINUX
* Students understand the importance of wait() function call
* Students understand what orphan and zombie processes are
* Students are able to understand how exec works
* Students can successfully use exec to load new binaries into running proceses

# **Monitoring Processes**

The **ps** command displays information about a selection of the active processes. If you want a repetitive update of the selection and the displayed information use top(1) instead.

Run the following command and list its output

|  |
| --- |
| > ps |

Output:

|  |
| --- |
|  |

Now run the following command and list its output

|  |
| --- |
| > top |

|  |
| --- |
|  |

### What does the command pstree do? Use man page to figure it out.

Your answer:

|  |
| --- |
|  |

Check out the man page for ps and list three helpful options that can be used

List of options to use with ps

|  |
| --- |
|  |

# Give output for each option:

|  |
| --- |
|  |
|  |
|  |

# **Installing the manpages:**

In order to list useful information for functions we will be using we need to install some additional manpages, run the following command to install them:

|  |
| --- |
| sudo apt-get install manpages-posix manpages-posix-dev |

# **Process Identification:**

Check out the manpage for getpid and getppid what do they do?

|  |
| --- |
|  |
|  |

What is the return type for getpid? Is it a signed or unsigned integer?

|  |
| --- |
|  |

**Process Creation:**

The fork() function can be used to create a child process. Refer to the manpage to answer the following:

What is the return value of fork() on success?

|  |
| --- |
|  |

What is the return value of fork() on failure?

|  |
| --- |
|  |

The following program creates a child process, copy and run the code and observe its output

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <sys/types.h>  int main(void){  printf("Hello World!\n");  pid\_t result = fork( );  printf("Hello from process id = %d\n", getpid( ));  return 0;  } |

What is the output?

|  |
| --- |
|  |

# Why is the output being displayed twice?

|  |
| --- |
|  |

Task 1: Notice in the above code we are not using the return value from fork(). Modify the above code such that the output is only displayed by the child process. Past your code here:

|  |
| --- |
|  |

|  |
| --- |
|  |

# **Process Completion:**

Check out the manpage for wait(2) and answer the following questions:

What is the output for **man wait?**

|  |
| --- |
|  |

What is the output for **man 2 wait** ?

|  |
| --- |
|  |

Can you now explain the **(2)** after **wait** in the original statement **wait(2)**?

|  |
| --- |
|  |

What is the difference between **wait()** and **waitpid()**?

|  |
| --- |
|  |

**Orphan processes:**

An **Orphan process** is a *running process* whose **parent process** has *finished* or *terminated*. In a Unix-like operating system any *orphaned process* will be immediately *adopted* by the special init system process. This operation is called *re-parenting* and occurs automatically. Even though technically the **orphan process** has the init process as its *parent*, it is still called an **orphan process** since the process that originally created it no longer exists.

### **Unintentional Orphan**

A process can be **orphaned unintentionally**: such as when the parent process terminates or crashes. The process group mechanism in most Unix-like operation systems can be used to help protect against *accidental orphaning*, where in coordination with the user’s shell will try to terminate all the child processes with the SIGHUP process signal, rather than letting them continue to run as orphans.

Run the following code a few times, and observe the value of parent pid on every line.

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <sys/types.h>  #include <stdlib.h>    int main(void){    pid\_t val = fork( );  if (val == 0){  for (int i = 0; i < 10; i++){  printf("parent pid:%d i = %d\n",getppid(),i);  }  }  return 0;  } |

Do you notice any strange behavior, what is it?

|  |
| --- |
|  |

Explain the reason for the odd behavior that you just noticed?

|  |
| --- |
|  |

A process that terminates cannot leave the system until its parent accepts its return code. If its parent process is already dead, it’ll already have been adopted by the “Run the following command and answer the following regarding zombie processes:

|  |
| --- |
| man 2 wait |

Note you can use grep to quickly filter out the relevant part like follows:” process, which always accepts its children’s return codes. However, **if a process’s parent is alive but never executes a wait ( ), the process’s return code will never be accepted and the process will remain a *zombie*.**

|  |
| --- |
| man 2 wait | grep zombie |

What can be the consequence of not removing zombie processes from the system?

|  |
| --- |
|  |

When is the info for a zombie finally removed from the system?

|  |
| --- |
|  |

Run the following code and observe its output

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h> /\* contains fork prototype \*/  int main ( )  {  pid\_t pid ;  pid = fork();  if ( pid > 0 )  {  While (1)  {  sleep(100);  }  }  else if (pid==0)  {  exit (0) ;  }  } |

# The above code attempts to create a zombie process, how can you tell if the process has been zombied? What command will you run to verify?

|  |
| --- |
|  |

Output of the command that shows that process has been zombied:

|  |
| --- |
|  |

# 

# **Process Binary Replacement:**

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. The following code replaces the child process with the binary created for hello.c

## **Step 1:** Create a file “hello.c” and type following source code

|  |
| --- |
| #include <stdio.h>  int main()  {  printf("Hello World\n");  } |

## **Step 2:** Compile and make a binary file named hello.out

## **Step3:** Create another file named “parent.c” with the following code.

|  |
| --- |
| #include <stdio.h>  #include <unistd.h> /\* contains fork prototype \*/  int main(void)  {  pid\_t pid;  printf("I am the parent process and my pid is : %d\n",getpid());  pid = fork(); //new process is created  if (pid == 0)  {  printf("I am the child process and my pid is :%d\n",getpid());  printf(“Loading binary \“hello.out\‟ \n”);  execv(“hello.out”,”hello.out”,NULL);  printf(“The binary has been loaded\n”);  }  else  printf("I am the parent process and my pid is: %d\n",getpid());  } |

Run the above code and observe the output. Have all the printf statements executed? If not, which one did not execute and why?

|  |
| --- |
|  |

Use the manpage and list prototypes for all different variants of exec(3) that are available

|  |
| --- |
|  |

The above code uses execv, modify the code so that it instead uses execl

|  |
| --- |
|  |

What is the difference between execl and execlp?

|  |
| --- |
|  |

Run the following code as above:

**Step 1:** Compile and make binary for the following program. Run it to see that it works as expected

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <sys/types.h>  #include <stdlib.h>    int main(int argc, char \*\*argv){  printf("the sum is %d\n",atoi(argv[1])+atoi(argv[2]));  return 0;  } |

**Step 2:** Write a program that now forks a child and loads the above binary into the child process, it also passes the parameters from the command line to the child process.

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <sys/types.h>  #include <sys/wait.h>  #include <stdlib.h>    int main(int argc, char \*\*argv){  pid\_t val = fork();  if (val == 0){  execv("sum",argv);  }  wait(NULL);  return 0;  } |

Modify the above the generate to output as below:

|  |
| --- |
| Sample run:  > ./task 6 7  > parent process: creating child and loading binary “sum.out”  > parent process: arguments passed ./task 6 7  > child process: arguments received = 3  > child process: arguments received ./task 6 7  > child process: the sum is 13 |

When should we use execv vs execl?

|  |
| --- |
|  |

**Graded Task:**

|  |
| --- |
| 1. Write a program that displays whether a given number passed from the command line is prime or not. 2. Write a program that forks a child and loads it with the above binary, additionally it passes the arguments it received on the command line to the child as above. |

**Graded Task:**

|  |  |
| --- | --- |
| 1. Modify the above program such that the parameters to be passed to child are taken as input in parent process rather being sent via command line. 2. Sample run:  |  | | --- | | > ./task  > parent process: Input a number: 7  > child process: 7 is a prime  >./task  > parent process: Input a number: 8  > child process: 8 is not a prime | |