# Topic to be covered

#### Process Creation

* + fork ()
  + getpid()
  + getppid()

#### Process Completion

* + wait (int \*)
  + exit (int)

#### Orphan Process

* Zombie Process
* Multiple forks in a program.
* Process Binary Replacement
  + exec () family of system calls
* Example Programs for proof of concept

Objectives

#### Understanding the concept of creating new processes in Linux.

#### Understanding the process id and parent process id using getpid() and getpid()

#### Parent waiting for its child to terminate using wait() system call.

* Concept of orphan and zombie Processes.
* Understanding the use of multiple forks in a program using tree diagram.
* Assigning task to a child using exec() family system calls.

Prerequisite:

* Visual Studio Code
* GCC compiler
* Basic C Programing
* Use of man Page

# Process Creation:

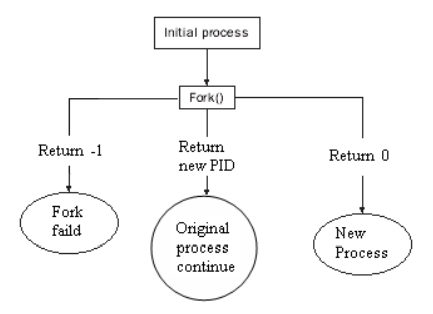
The fork function creates a new process.

Duplicating a process image:

To use processes to perform more than one function at a time we can either use threads or create an entirely separate process from within program by calling fork. System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

* If fork() returns a negative value, the creation of a child process was unsuccessful.
* fork() returns a zero to the newly created child process.
* fork() returns a positive value, the process ID of the child process, to the parent.
* The returned process ID is of type pid\_t defined in sys/types.h and it is an integer value.

Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.



**pid\_t fork()**

# 

# Example-01: Creating a new process.

# //This program uses fork() to create a new process.

# #include <stdio.h>

# #include <unistd.h>

# #include <stdlib.h>

# int main()

# {

# int cpid;

# cpid = fork();

# if (cpid == -1){

# printf("Fork failed\n");

# exit(1);

# }

# if (cpid == 0){

# printf("Hello I m Child Process\n");

# }

# else{

# printf("Hello I m Parent Process\n);

# }

# return 0;

# }

# Task01: Run the above program and understand the output.

# Paste output below and write description of output briefly.

**OUTPUT:**

**Description:**

# Example-02: Creating a new process and printing ids.

# //This program just use fork and displays the IDs of the parent and child process

# #include <stdio.h>

# #include <unistd.h>

# #include <stdlib.h>

# int main()

# {

# int cpid;

# cpid = fork();

# if (cpid == -1){

# printf("Fork failed\n");

# exit(1);

# }

# if (cpid == 0){

# printf("Hello I m Child and my PID is:%ld\n",(long)getpid());

# }

# else{

# printf("Hello I m Parent and my PI is:%ld\n",(long)getpid());

# }

# return 0;

# }

# Task02: Run the above program and understand the output.

# Paste output below and write description of output briefly.

**OUTPUT:**

**Description:**

**Process Completion:**

The functions described in this section are used to **wait** for a child process to terminate or stop, and determine its status. These functions are declared in the header file "**sys/wait.h**".

**pid\_t wait (int \* status)**

**wait()** will force a parent process to wait for a child process to stop or terminate. **wait()** return the pid of the child or

-1 for an error. The exit status of the child is returned to **status**.

**void exit (int status)**

**exit()** terminates the process which calls this function and returns the exit status value. Both UNIX and C (forked) programs can read the status value.

# Example-03: Parent waiting for child to terminate.

# //Parent creates the child and waits for it to exit.

# #include <stdio.h>

# #include <unistd.h>

# #include <stdlib.h>

# int main()

# {

# int cpid;

# cpid = fork();

# if (cpid == -1){

# printf("Fork failed\n");

# exit(1);

# }

# if (cpid == 0){

# printf("Hello I m Child and my PID is:%ld\n",(long)getpid());

# exit(0);

# }

# else{

# wait();

# printf("Hello I m Parent and my PI is:%ld\n",(long)getpid());

# }

# return 0;

# }

# Task03: Run the above program and understand the output.

# Paste output below and write description of output briefly.

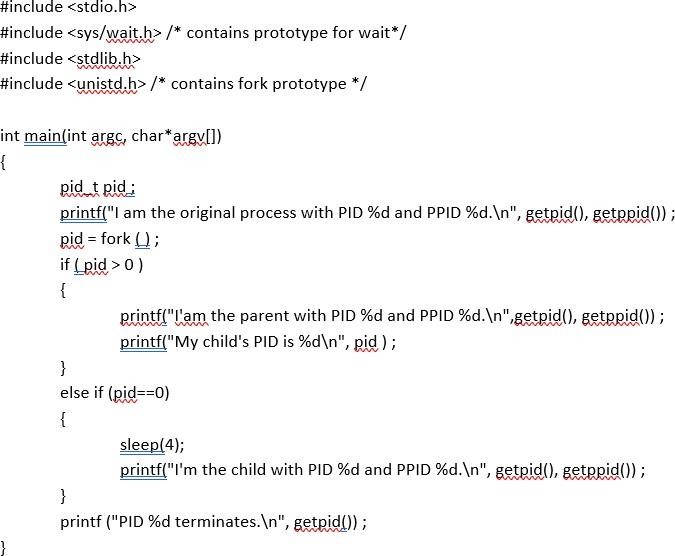
**OUTPUT:**

**Description:**

**Orphan processes:**

When a parent dies before its child, the child is automatically adopted by the original “init” process whose **PID** is 1. To illustrate this insert a **sleep** statement into the child’s code. This ensured that the parent process terminated before its child.

# Example-04: Creating Orphan Process.



# Task04: Run the above program and understand the output.

# Paste output below and write description of output briefly.

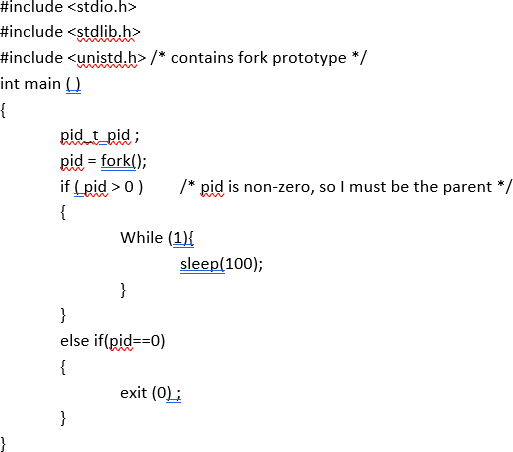
**OUTPUT:**

**Description:**

**Zombie processes:**

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 “**init**” 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*.**

# Example-05: Creating Zombie Process



# Task05: Run the above program and understand the output. See the status of both processes using “ps -a”. How the child can be terminated?

**Description:**

**Example-06:  *Describes what happens when fork() is called multiple times***

#include<stdio.h>

*int main(){*

*fork();*

*fork();*

*fork();*

*printf("Hello fork...\n");*

*return 0;*

*}*

**Task-06: Compile and run above program and discuss the output with tree diagram:**

**Output:**

**Description With Tree Diagram:**

**Example-07: *Describes what happens when fork() is called multiple times using &&.***

#include<stdio.h>

*int main(){*

*fork() && fork();*

*printf("UCP\n");*

*return 0;*

*}*

**Task-07: Compile and run above program and discuss the output with tree diagram:**

**Output:**

**Description With Tree Diagram:**

**Example-08: *Describes what happens when fork() is called multiple times using ||.***

#include<stdio.h>

*int main(){*

*fork() || fork();*

*printf("UCP\n");*

*return 0;*

*}*

**Task-08: Compile and run above program and discuss the output with tree diagram:**

**Output:**

**Description With Tree Diagram:**

**Replacing a process code with binary (executable):**

The exec system call is used to execute a file which is residing in an active process. When exec is called the previous executable file is replaced and new file is executed.

More precisely, we can say that using exec system call will replace the old file or program from the process with a new file or program. The entire content of the process is replaced with a new program. The user data segment which executes the exec() system call is replaced with the data file whose name is provided in the argument while calling exec(). The new program is loaded into the same process space. The current process is just turned into a new process and hence the process id PID is not changed, this is because we are not creating a new process we are just replacing a process with another process in exec. If the currently running process contains more than one thread then all the threads will be terminated and the new process image will be loaded and then executed. There are no destructor functions that terminate threads of current process. PID of the process is not changed but the data, code, stack, heap, etc. of the process are changed and are replaced with those of newly loaded process. The new process is executed from the entry point.

Exec system call is a collection of functions and in C programming language, the standard names for these functions are as follows:

1. **execl**
2. **execle**
3. **execlp**
4. **execv**
5. **execve**
6. **execvp**

It should be noted here that these functions have the same base *exec*followed by one or more letters. These are explained below:

**Why exec is used?**

exec is used when the user wants to launch a new file or program in the same process.

### **Inner Working of exec**

Consider the following points to understand the working of exec:

1. Current process image is overwritten with a new process image.
2. New process image is the one you passed as exec argument
3. The currently running process is ended
4. New process image has same process ID, same environment, and same file descriptor (because process is not replaced process image is replaced)
5. The CPU stat and virtual memory is affected. Virtual memory mapping of the current process image is replaced by virtual memory of new process image.

### Syntaxes of exec family functions:

The following are the syntaxes for each function of exec:

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

**Example-09: Use of exec() family system calls.**

#include<stdio.h>

*int main(){*

*int cpid = fork();*

*if (cpid == 0){*

*execl(”/bin/ls", "myls", "-l", "/home/", NULL);*

*printf("This line will not be printed\n");*

*}*

*else{*

*wait(NULL);*

*printf("Hello I m Parent.\n");*

*}*

*return 0;*

*}*

Output: Understand the output and paste here.

Description:

**Example-10: Running calculator program using exec in child.**

*int main()*

*{*

*int cpid = fork();*

*if (cpid == 0){*

*execl("/usr/bin/gnome-calculator", "mycalc",NULL);*

*printf("This line will not be printed\n");*

*}*

*else{*

*wait(NULL);*

*printf("Hello I m Parent.\n");*

*}*

*return 0;*

*}*

Output: Understand the output and paste here.

Description:

**Task-11: Write a program that adds two numbers . Compile and run it to test. Now write another program as given in Example-10. Now instead of passing gnome-calculator, pass it executable of your addition program. Does this work?**

**Task-12: Run the sample programs to get proof of concept about sperate copy of variables in child and parent, vfork, etc. See the attached list of programs.**

# Process Creation:

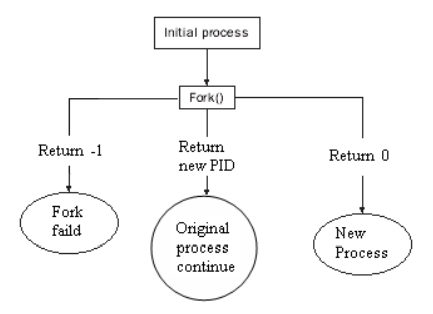
The fork function creates a new process.

Duplicating a process image:

To use processes to perform more than one function at a time we can either use threads or create an entirely separate process from within program by calling fork. System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

* If fork() returns a negative value, the creation of a child process was unsuccessful.
* fork() returns a zero to the newly created child process.
* fork() returns a positive value, the process ID of the child process, to the parent.
* The returned process ID is of type pid\_t defined in sys/types.h and it is an integer value.

Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.



**pid\_t fork()**

# Example-01: Creating a new process.

# //This program uses fork() to create a new process.

# #include <stdio.h>

# #include <unistd.h>

# #include <stdlib.h>

# int main()

# {

# int cpid;

# cpid = fork();

# if (cpid == -1){

# printf("Fork failed\n");

# exit(1);

# }

# if (cpid == 0){

# printf("Hello I m Child Process\n");

# }

# else{

# printf("Hello I m Parent Process\n);

# }

# return 0;

# }

# Task01: Run the above program and understand the output.

# Paste output below and write description of output briefly.

**OUTPUT:**

**Description:**

# Example-02: Creating a new process and printing ids.

# //This program just use fork and displays the IDs of the parent and child process

# #include <stdio.h>

# #include <unistd.h>

# #include <stdlib.h>

# int main()

# {

# int cpid;

# cpid = fork();

# if (cpid == -1){

# printf("Fork failed\n");

# exit(1);

# }

# if (cpid == 0){

# printf("Hello I m Child and my PID is:%ld\n",(long)getpid());

# }

# else{

# printf("Hello I m Parent and my PI is:%ld\n",(long)getpid());

# }

# return 0;

# }

# Task02: Run the above program and understand the output.

# Paste output below and write description of output briefly.

**OUTPUT:**

**Description:**

**Process Completion:**

The functions described in this section are used to **wait** for a child process to terminate or stop, and determine its status. These functions are declared in the header file "**sys/wait.h**".

**pid\_t wait (int \* status)**

**wait()** will force a parent process to wait for a child process to stop or terminate. **wait()** return the pid of the child or

-1 for an error. The exit status of the child is returned to **status**.

**void exit (int status)**

**exit()** terminates the process which calls this function and returns the exit status value. Both UNIX and C (forked) programs can read the status value.

# Example-03: Parent waiting for child to terminate.

# //Parent creates the child and waits for it to exit.

# #include <stdio.h>

# #include <unistd.h>

# #include <stdlib.h>

# int main()

# {

# int cpid;

# cpid = fork();

# if (cpid == -1){

# printf("Fork failed\n");

# exit(1);

# }

# if (cpid == 0){

# printf("Hello I m Child and my PID is:%ld\n",(long)getpid());

# exit(0);

# }

# else{

# wait();

# printf("Hello I m Parent and my PI is:%ld\n",(long)getpid());

# }

# return 0;

# }

# Task03: Run the above program and understand the output.

# Paste output below and write description of output briefly.

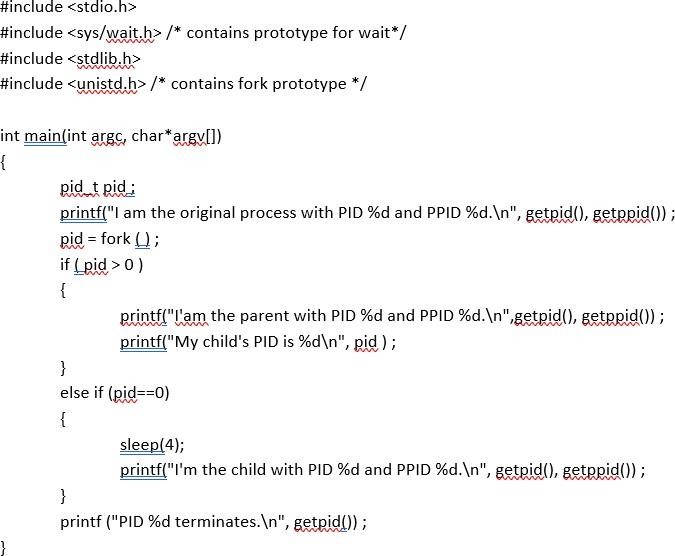
**OUTPUT:**

**Description:**

**Orphan processes:**

When a parent dies before its child, the child is automatically adopted by the original “init” process whose **PID** is 1. To illustrate this insert a **sleep** statement into the child’s code. This ensured that the parent process terminated before its child.

# Example-04: Creating Orphan Process.



# Task04: Run the above program and understand the output.

# Paste output below and write description of output briefly.

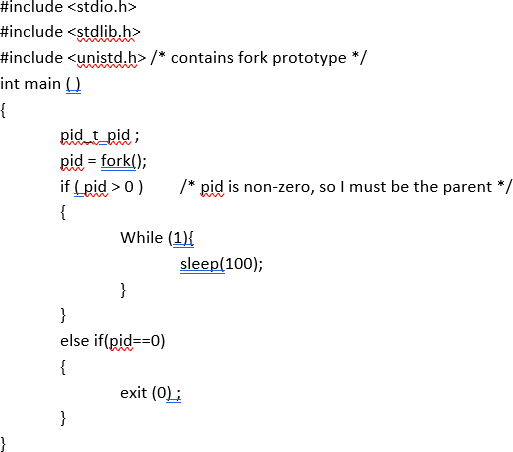
**OUTPUT:**

**Description:**

**Zombie processes:**

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 “**init**” 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*.**

# Example-05: Creating Zombie Process



# Task05: Run the above program and understand the output. See the status of both processes using “ps -a”. How the child can be terminated?

**Description:**

**Example-06:  *Describes what happens when fork() is called multiple times***

#include<stdio.h>

*int main(){*

*fork();*

*fork();*

*fork();*

*printf("Hello fork...\n");*

*return 0;*

*}*

**Task-06: Compile and run above program and discuss the output with tree diagram:**

**Output:**

**Description With Tree Diagram:**

**Example-07: *Describes what happens when fork() is called multiple times using &&.***

#include<stdio.h>

*int main(){*

*fork() && fork();*

*printf("UCP\n");*

*return 0;*

*}*

**Task-07: Compile and run above program and discuss the output with tree diagram:**

**Output:**

**Description With Tree Diagram:**

**Example-08: *Describes what happens when fork() is called multiple times using ||.***

#include<stdio.h>

*int main(){*

*fork() || fork();*

*printf("UCP\n");*

*return 0;*

*}*

**Task-08: Compile and run above program and discuss the output with tree diagram:**

**Output:**

**Description With Tree Diagram:**

**Replacing a process code with binary (executable):**

The exec system call is used to execute a file which is residing in an active process. When exec is called the previous executable file is replaced and new file is executed.

More precisely, we can say that using exec system call will replace the old file or program from the process with a new file or program. The entire content of the process is replaced with a new program. The user data segment which executes the exec() system call is replaced with the data file whose name is provided in the argument while calling exec(). The new program is loaded into the same process space. The current process is just turned into a new process and hence the process id PID is not changed, this is because we are not creating a new process we are just replacing a process with another process in exec. If the currently running process contains more than one thread then all the threads will be terminated and the new process image will be loaded and then executed. There are no destructor functions that terminate threads of current process. PID of the process is not changed but the data, code, stack, heap, etc. of the process are changed and are replaced with those of newly loaded process. The new process is executed from the entry point.

Exec system call is a collection of functions and in C programming language, the standard names for these functions are as follows:

1. **execl**
2. **execle**
3. **execlp**
4. **execv**
5. **execve**
6. **execvp**

It should be noted here that these functions have the same base *exec*followed by one or more letters. These are explained below:

**Why exec is used?**

exec is used when the user wants to launch a new file or program in the same process.

### **Inner Working of exec**

Consider the following points to understand the working of exec:

1. Current process image is overwritten with a new process image.
2. New process image is the one you passed as exec argument
3. The currently running process is ended
4. New process image has same process ID, same environment, and same file descriptor (because process is not replaced process image is replaced)
5. The CPU stat and virtual memory is affected. Virtual memory mapping of the current process image is replaced by virtual memory of new process image.

### Syntaxes of exec family functions:

The following are the syntaxes for each function of exec:

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

**Example-09: Use of exec() family system calls.**

#include<stdio.h>

*int main(){*

*int cpid = fork();*

*if (cpid == 0){*

*execl(”/bin/ls", "myls", "-l", "/home/", NULL);*

*printf("This line will not be printed\n");*

*}*

*else{*

*wait(NULL);*

*printf("Hello I m Parent.\n");*

*}*

*return 0;*

*}*

Output: Understand the output and paste here.

Description:

**Example-10: Running calculator program using exec in child.**

*int main()*

*{*

*int cpid = fork();*

*if (cpid == 0){*

*execl("/usr/bin/gnome-calculator", "mycalc",NULL);*

*printf("This line will not be printed\n");*

*}*

*else{*

*wait(NULL);*

*printf("Hello I m Parent.\n");*

*}*

*return 0;*

*}*

Output: Understand the output and paste here.

Description:

**Task-11: Write a program that adds two numbers . Compile and run it to test. Now write another program as given in Example-10. Now instead of passing gnome-calculator, pass it executable of your addition program. Does this work?**

**Task-12: Run the sample programs to get proof of concept about sperate copy of variables in child and parent, vfork, etc. See the attached list of programs.**