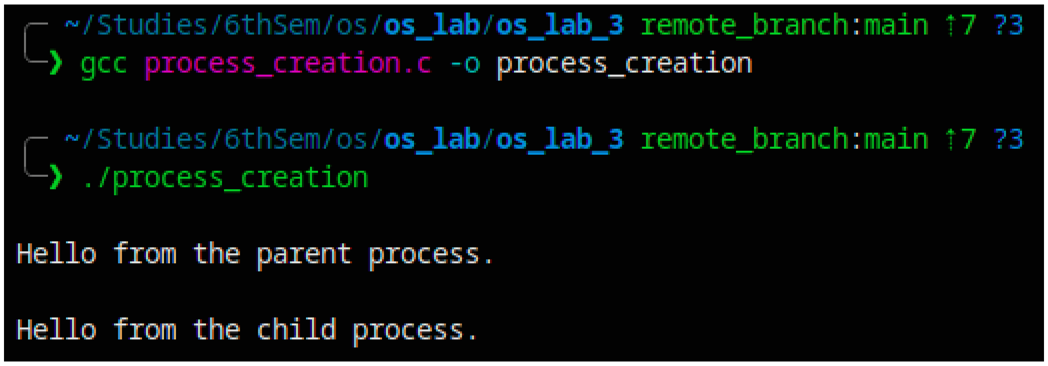
1. Process Creation

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
   int id;
   id = fork(); // Process creation
   if(id < 0)
   {
      printf("\nCan not create child process.\n");
      exit(-1);
   }
   else if(id == 0) // Child process
   {
      printf("\nHello from the child process.\n");
   }
   else
   {
      printf("\nHello from the parent process.\n");
   }
   return 0;
}
```

Output:



```
┌ ~/Studies/6thSem/os/os_lab/os_lab_3 remote_branch:main ↑7 ?3
└❯ gcc process_creation.c -o process_creation

┌ ~/Studies/6thSem/os/os_lab/os_lab_3 remote_branch:main ↑7 ?3
└❯ ./process_creation

Hello from the parent process.

Hello from the child process.
```

Discussion:

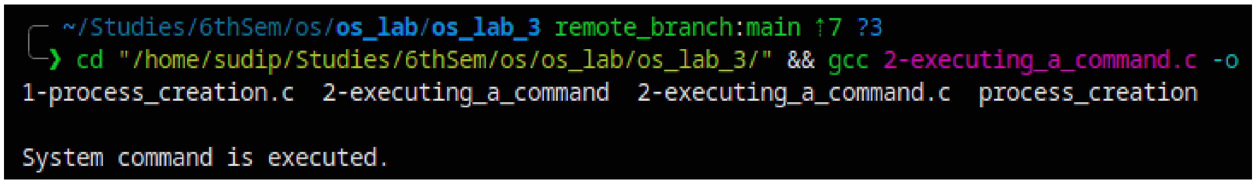
This C code demonstrates a simple program that uses the fork system call to create a child process. The program begins by declaring an integer variable 'id' and assigns the result of the fork() function, which initiates the process creation. The code then checks if the fork was successful (id < 0) and prints an error message if not. If the fork was successful, the program diverges into two branches: one for the parent process and one for the child process. In the child process branch (id == 0), a message is printed indicating that it is the child process. In the parent process branch (id > 0), a message is printed indicating that it is the parent process. Finally, the program returns 0, signaling successful execution.

2. Executing a command

Program:

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


int main()
{
   const char* system_command = "ls";

   int status = system(system_command);
   if(status == -1)
   {
      printf("\nCan not execute system command.\n");
      exit(-1);
   }
   else
   {
      printf("\nSystem command is executed.\n");
   }
   return 0;
}
```

Output:



```
╭─ ~/Studies/6thSem/os/os_lab/os_lab_3 remote_branch:main ↑7 ?3
╰─❯ cd "/home/sudip/Studies/6thSem/os/os_lab/os_lab_3/" && gcc 2-executing_a_command.c -o
1-process_creation.c  2-executing_a_command  2-executing_a_command.c  process_creation

System command is executed.
```

Discussion:

This C code exemplifies a program that utilizes the system function to execute a shell command, in this case, the "ls" command. The program begins by defining a constant character pointer 'system_command' containing the shell command to be executed. It then uses the system() function to execute the command and captures the return status. If the status is -1, an error message is printed, indicating that the system command could not be executed. Otherwise, if the status is non-negative, a success message is displayed, indicating the successful execution of the system command. The program concludes by returning 0, indicating successful completion.

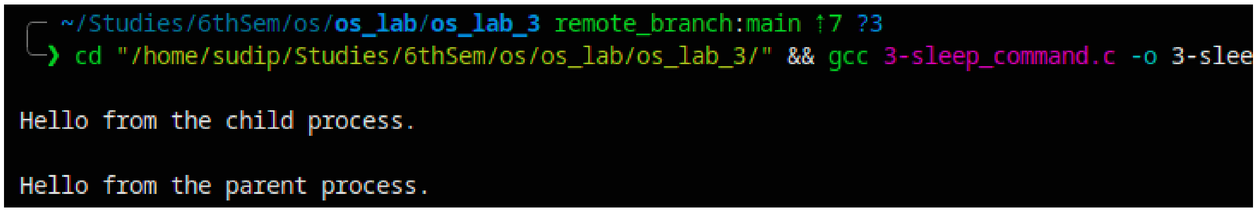
3. Sleep Command

Program:

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

```
int main()
{
    int id;
    id = fork(); // Process creation
    if(id < 0)
    {
        printf("\nCan not create child process.\n");
        exit(-1);
    }
    else if(id == 0) // Child process
    {
        printf("\nHello from the child process.\n");
    }
    else
    {
        sleep(3); // Parent Process sleep for 3 seconds
        printf("\nHello from the parent process.\n"); // Output after 3 seconds delay
    }
    return 0;
}
```

Output:



```
╭─ ~/Studies/6thSem/os/os_lab/os_lab_3 remote_branch:main ↑7 ?3
╰─❯ cd "/home/sudip/Studies/6thSem/os/os_lab/os_lab_3/" && gcc 3-sleep_command.c -o 3-slee

Hello from the child process.

Hello from the parent process.
```

Discussion:

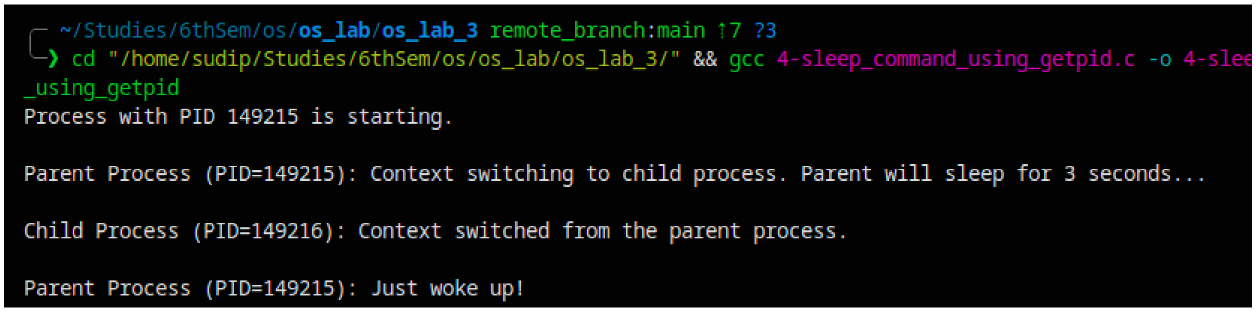
This C code showcases a program using the fork system call to create a child process. After the fork, the code checks for successful process creation. If successful, the program splits into two branches for the parent and child processes. In the child process branch (id == 0), a message is printed indicating it is the child process. In the parent process branch, the parent process sleeps for 3 seconds using the sleep function and then prints a message. The intentional delay in the parent process results in the child process message being displayed first, followed by the parent process message after the specified sleep period. The program concludes by returning 0, indicating successful execution.

4. Sleep command using getpid

Program:

```
#include <stdio.h>
#include<stdlib.h> // for exit()
#include <unistd.h>

int main()
{
    // Get the PID of the current process
    int pid = getpid();
    printf("Process with PID %d is starting.\n", pid);

    // Forking to create a child process
    int id = fork();

    // Check if fork was successful
    if(id < 0)
    {
        printf("\nError: Cannot create child process.\n");
        exit(-1);
    } else if(id == 0)
    {
        // Code block for the child process
        printf("\nChild Process (PID=%d): Context switched from the parent process.\n", getpid());
    } else if(pid == getpid())
    {
        // Code block for the parent process
        printf("\nParent Process (PID=%d): Context switching to child process. Parent will sleep for
3 seconds...\n", getpid());

        // Sleep for 3 seconds to allow context switching
        sleep(3);

        // Parent wakes up after sleep
        printf("\nParent Process (PID=%d): Just woke up after 3 seconds!\n", getpid());
    }

    return 0;
}
```

Output:



```
╭─ ~/Studies/6thSem/os/os_lab/os_lab_3 remote_branch:main ↑7 ?3
╰─❯ cd "/home/sudip/Studies/6thSem/os/os_lab/os_lab_3/" && gcc 4-sleep_command_using_getpid.c -o 4-slee
_using_getpid
Process with PID 149215 is starting.

Parent Process (PID=149215): Context switching to child process. Parent will sleep for 3 seconds...

Child Process (PID=149216): Context switched from the parent process.

Parent Process (PID=149215): Just woke up!
```

Discussion:
This C code illustrates a program that starts by obtaining the PID (Process ID) of the current process and prints a corresponding message. It then uses the fork system call to create a child

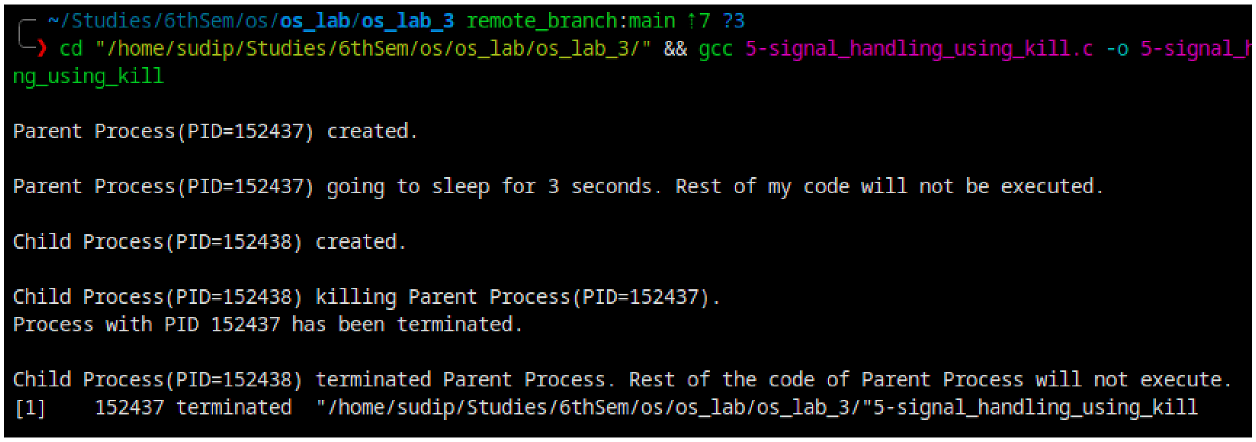
process. The code checks for successful process creation, printing an error message and exiting if the fork fails. In the child process branch (id == 0), a message is printed to indicate that the context has switched from the parent process. In the parent process branch, the program prints a message about context switching to the child process and proceeds to sleep for 3 seconds using the sleep function. After waking up, the parent process prints a message to signify its awakening. The intentional sleep allows for context switching between the parent and child processes. The program concludes by returning 0, indicating successful execution.

5. Signal Handling using kill

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // Unix Standard for fork(), sleep()
#include <signal.h> // for kill()

int main()
{
    int id;
    id = fork();
    if (id < 0)
    {
        printf("\nCould not create a child process.\n");
        exit(-1);
    }
    else if (id == 0) /*Child Process*/
    {
        printf("\nChild Process(PID=%d) created.\n", getpid());
        printf("\nChild Process(PID=%d) killing Parent Process(PID=%d).\n", getpid(), getppid());

        // kill(getppid());

        // Send SIGTERM to terminate the process
        if (kill(getppid(), SIGTERM) == 0)
        {
            printf("Process with PID %d has been terminated.\n", getppid());
        }
        else
        {
            perror("Error killing process");
        }

        printf("\nChild Process(PID=%d) terminated Parent Process. Rest of the code of Parent
Process will not execute.\n", getpid());
    }
    else
    {
        // This is a Parent Process.
        // Parent Process executes before child process normally.
        printf("\nParent Process(PID=%d) created.\n", getpid());
        printf("\nParent Process(PID=%d) going to sleep for 3 seconds. Rest of my code will not be
executed. \n", getpid());
        sleep(3);
        printf("\nParent Process(PID=%d)'s this code after sleep will never be executed.\n",
getpid());
    }
    return 0;
}
```

Output:



```
┌ ~/Studies/6thSem/os/os_lab/os_lab_3 remote_branch:main ↑7 ?3
└❯ cd "/home/sudip/Studies/6thSem/os/os_lab/os_lab_3/" && gcc 5-signal_handling_using_kill.c -o 5-signal_h
ng_using_kill

Parent Process(PID=152437) created.

Parent Process(PID=152437) going to sleep for 3 seconds. Rest of my code will not be executed.

Child Process(PID=152438) created.

Child Process(PID=152438) killing Parent Process(PID=152437).
Process with PID 152437 has been terminated.

Child Process(PID=152438) terminated Parent Process. Rest of the code of Parent Process will not execute.
[1]    152437 terminated  "/home/sudip/Studies/6thSem/os/os_lab/os_lab_3/"5-signal_handling_using_kill
```

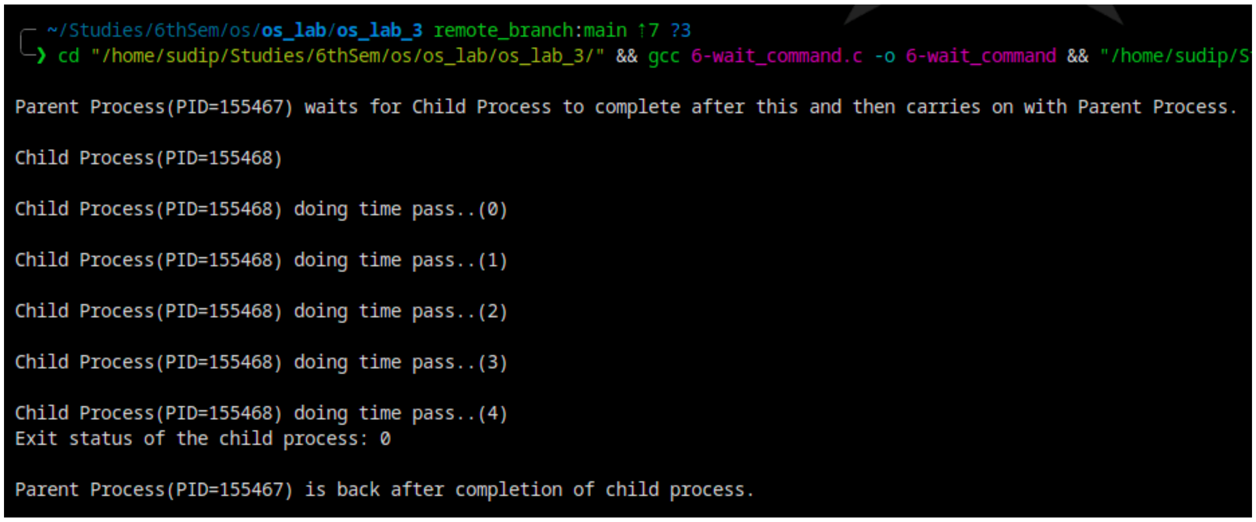
Discussion:
This C code demonstrates a program using the fork system call to create a child process. After forking, the program checks for successful process creation. If successful, it splits into two branches for the parent and child processes. In the child process branch (id == 0), the child process prints messages indicating its creation and attempts to terminate the parent process using the kill function with SIGTERM (15). If successful, it prints a termination message; otherwise, an error is displayed. The parent process branch prints messages indicating its creation, announces it will sleep for 3 seconds, and then sleeps accordingly. After waking up, the parent process prints a message indicating that the subsequent code will not be executed. The program concludes by returning 0, indicating successful execution. Note that the commented-out line `// kill(getppid());` can be uncommented to terminate the parent process without using SIGTERM.

6. Wait command

Program:

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

```
#include <unistd.h> // Unix Standard for fork(), sleep()
#include <sys/wait.h>

int main()
{
   int id;
   id = fork(); // Process creation
   if (id < 0)
   {
      printf("\nCan not create child process.\n");
      exit(-1);
   }
   else if (id == 0) // Child process
   {
      printf("\nChild Process(PID=%d)\n", getpid());
      for (int i = 0; i < 5; ++i)
      {
         printf("\nChild Process(PID=%d) doing time pass..(%d)\n", getpid(),i); // Simulating child
process to be busy with its own task
      }
   }
   else
   {
      // Parent Process
      printf("\nParent Process(PID=%d) waits for Child Process to complete after this and then
carries on with Parent Process.\n", getpid());
      // wait();
      int status;
      wait(&status); // Wait for Child Process to execute fully.
      printf("Exit status of the child process: %d\n", WEXITSTATUS(status));
      printf("\nParent Process(PID=%d) is back after completion of child process.\n", getpid());
   }
   return 0;
}
```
Output:



```
╭─ ~/Studies/6thSem/os/os_lab/os_lab_3 remote_branch:main ↑7 ?3
╰─❯ cd "/home/sudip/Studies/6thSem/os/os_lab/os_lab_3/" && gcc 6-wait_command.c -o 6-wait_command && "/home/sudip/S

Parent Process(PID=155467) waits for Child Process to complete after this and then carries on with Parent Process.

Child Process(PID=155468)

Child Process(PID=155468) doing time pass..(0)

Child Process(PID=155468) doing time pass..(1)

Child Process(PID=155468) doing time pass..(2)

Child Process(PID=155468) doing time pass..(3)

Child Process(PID=155468) doing time pass..(4)
Exit status of the child process: 0

Parent Process(PID=155467) is back after completion of child process.
```

Discussion:
This C code illustrates a program using the fork system call to create a child process. After forking, the program checks for successful process creation. If successful, it diverges into two branches for the parent and child processes. In the child process branch (id == 0), a loop simulates the child process being busy with its own task, printing messages accordingly. In the parent process branch, it waits for the child process to complete using the wait function. The status of the child process is then obtained using the wait macro, and the exit status is printed. Finally, a message indicates that the parent process has resumed after the completion of the child process. The program concludes by returning 0, indicating successful execution. Note that the line `// wait();` is commented out, and it can be replaced with `wait(&status);` to properly obtain the exit status of the child process.