Practical no: 3

Name: Saloni Vishwakarma
Batch-Roll no: C1-13
Subject: Operating System

Aim: To demonstrate process system calls.

Process System Calls in OS

1. What do you mean by system call?

Ans: A system call is a mechanism used by programs to request a service from the operating system kernel. It allows programs to interact with the operating system and access its resources such as file systems, network interfaces, and other hardware devices. System calls provide an interface between user-level applications and the operating system kernel, allowing programs to execute privileged instructions without directly accessing hardware resources.

System calls are typically implemented as function calls in high-level programming languages, and they are an essential component of modern operating systems. Examples of system calls include opening and closing files, reading and writing data from files, creating and deleting processes, and allocating and deallocating memory.

2. What is kernel mode?

Ans: Kernel mode, also known as system mode, is a privileged mode of operation in which the operating system kernel executes. In this mode, the operating system has full access to all hardware resources, and it can perform privileged operations such as modifying memory, accessing I/O devices, and executing privileged instructions that cannot be executed in user mode.

Kernel mode is typically used to execute system services such as device drivers, interrupt handlers, and other low-level functions that require direct access to hardware resources. In contrast, user mode is a non-privileged mode of operation in which applications and other software run. User-mode programs are restricted in their access to hardware resources and must make system calls to the kernel to request access to these resources.

3. Which header file defines the "fork()" function?

Ans: The "fork()" function is defined in the <unistd.h> header file in C programming language. This header file provides access to the POSIX (Portable Operating System

Interface) API, which is a standard interface for interacting with the operating system in a portable and consistent manner across different Unix-like systems.

The "fork()" function creates a new process by duplicating the calling process, which becomes the parent process of the new process, called the child process. The child process has an identical copy of the parent's memory, including the program code, data, and stack. However, the two processes run independently, with their own copies of the CPU registers, program counters, and other state information.

4. Which process will execute the statement following the fork() call—the parent or the child?

Ans: After the fork() call, both the parent process and the child process will continue executing from the next statement after the fork() call.

However, the call fork() returns different values to the parent process and the child process. In the parent process, fork() returns the process ID (PID) of the child process, whereas in the child process, fork() returns 0. This allows the parent process and the child process to distinguish themselves from each other and execute different sections of code as needed.

5. What does the fork() function's negative value denote?

Ans: If the fork() function returns a negative value, it indicates that the creation of a new process has failed. Specifically, a negative return value indicates that fork() was unable to create a new process, usually because the system has run out of resources such as memory or process IDs.

In such cases, the value returned by fork() is typically -1, and the child process is not created. The parent process should handle this error condition by checking the return value of fork() and taking appropriate action, such as printing an error message and terminating the program.

6. Difference between fork() and exec()

Ans: `fork()` and `exec()` are two distinct system calls that are often used together to create new processes in Unix-like operating systems. Here are the differences between these two system calls:

1. `fork()` creates a new process by duplicating the calling process, while `exec()` replaces the current process image with a new process image.

2. `fork()` creates a new process with the same memory image as the parent process, while `exec()` loads a new program image into the current process's memory and begins executing it.

3. `fork()` returns the process ID (PID) of the child process to the parent process, while `exec()` does not return a value to the calling process if it succeeds (but may return an error code if it fails).

4. `fork()` creates a new process with the same environment and working directory as the parent process, while `exec()` can be used to launch a new process with a different environment and working directory.

5. `fork()` is often used to create a new process that will run concurrently with the parent process, while `exec()` is often used to replace the current process image with a new one, typically as part of a shell command or script.

In summary, `fork()` creates a new process by duplicating the calling process, while `exec()` replaces the current process image with a new one. These two system calls are often used together to create new processes in Unix-like operating systems.

7. Calculation in parent and child process using fork()

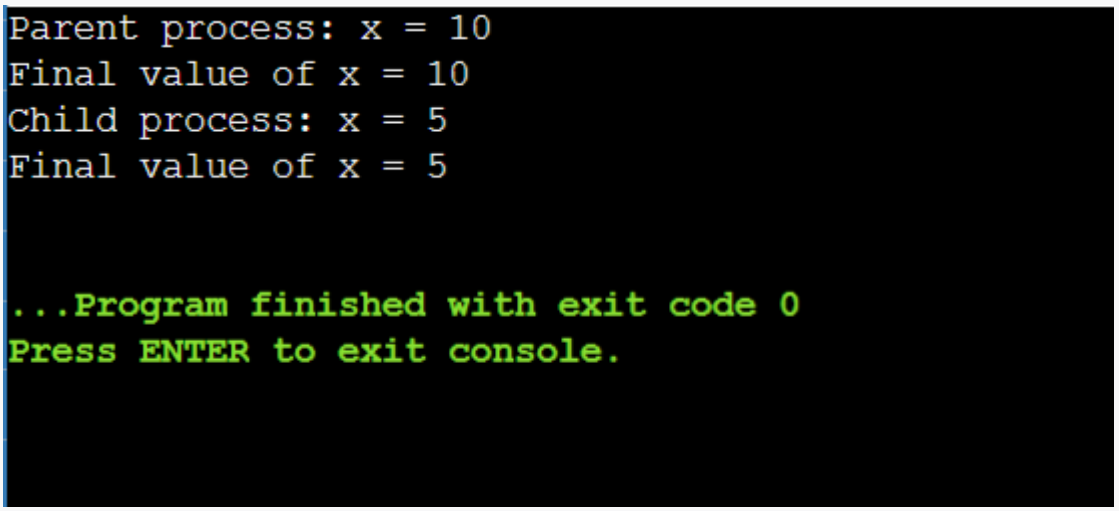
Ans: When a new process is created using fork(), the child process is a duplicate of the parent process, with its own memory space and copy-on-write pages. Any changes made to the memory space of one process do not affect the other process.

Therefore, if both the parent and child processes perform calculations after the fork() call, the calculations will be performed independently by each process, with no direct interaction between the two processes.

Here's an example of how the parent and child processes can perform separate calculations after the fork() call:

```
#include <stdio.h>

#include <unistd.h>

int main() {

   pid_t pid;

   int x = 0;

   pid = fork();

   if (pid == 0) {

      // child process
```

```
        x = 5;
        printf("Child process: x = %d\n", x);
    } else if (pid > 0) {
        // parent process
        x = 10;
        printf("Parent process: x = %d\n", x);
    } else {
        // fork() failed
        fprintf(stderr, "fork() failed.\n");
        return 1;
    }
    // Both parent and child processes continue executing from here
    printf("Final value of x = %d\n", x);
    return 0;
}
```

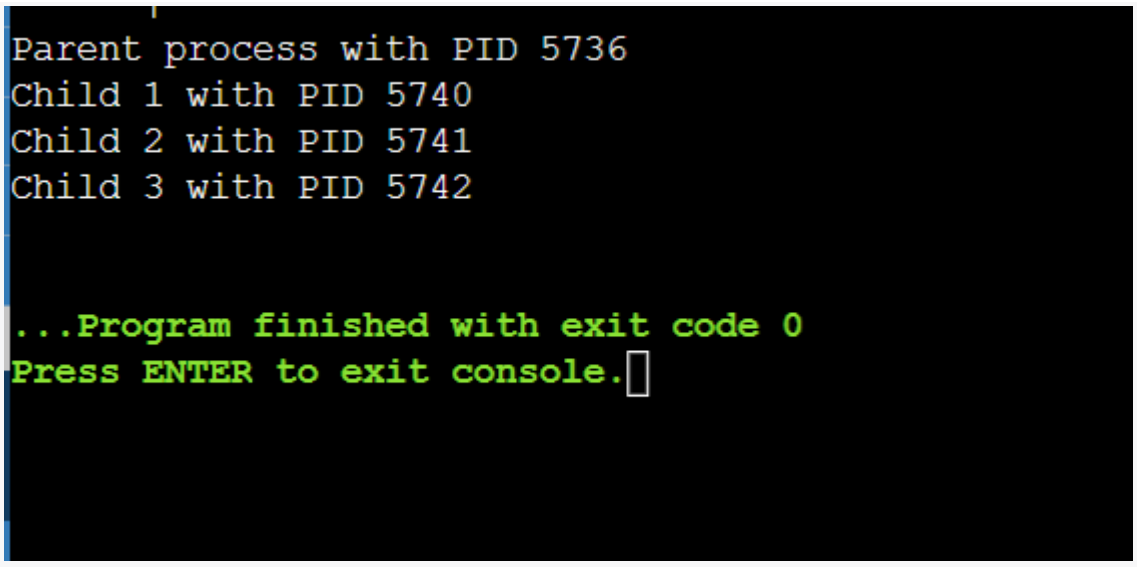


8. Creating multiple process using fork()

Ans: Using fork(), it is possible to create multiple child processes from a single parent process. Each child process created by the fork() call will have its own process ID (PID) and will be a duplicate of the parent process, with its own copy of the parent's memory space.

Here is an example of how to create multiple child processes using fork():

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
  int i, num_children = 3;
  for (i = 0; i < num_children; i++) {
    pid_t pid = fork();
    if (pid == 0) {
      // child process
      printf("Child %d with PID %d\n", i+1, getpid());
      exit(0);
    } else if (pid < 0) {
      // fork() failed
      fprintf(stderr, "fork() failed.\n");
      return 1;
    }
  }
  // parent process
  printf("Parent process with PID %d\n", getpid());
  // wait for child processes to terminate
  for (i = 0; i < num_children; i++) {
    wait(NULL);
  }
  return 0;
}
```



9. Creating n-child process from same parent process using fork() in C

Ans: To create n child processes from the same parent process using fork() in C, you can use a loop to call fork() n times. Here's an example program that creates n child processes:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Usage: %s <num_children>\n", argv[0]);
    return 1;
  }
  int num_children = atoi(argv[1]);
  int i;
  for (i = 0; i < num_children; i++) {
    pid_t pid = fork();

    if (pid < 0) {
      // fork() failed
      fprintf(stderr, "fork() failed.\n");
      return 1;
    } else if (pid == 0) {
      // child process
      printf("Child process with PID %d\n", getpid());
      exit(0);
    }
  }
```

```
  // parent process
  printf("Parent process with PID %d\n", getpid());
  // wait for all child processes to terminate
  for (i = 0; i < num_children; i++) {
     wait(NULL);
  }
  return 0;
}
```

```
Child process with PID 1234
Child process with PID 1235
Child process with PID 1236
Parent process with PID 1233
```

1) Parent child:

```
// parent.c:
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
int main (int argc, char **argv)
{
 int i=0;
long sum;
int pid;
int status, ret;
char *myargs [] = { NULL };
char *myenv [] = { NULL };
printf ("Parent: Hello, World!\n");
pid = fork ();
if (pid == 0) {
// I am the child
execve ("child", myargs, myenv);
```

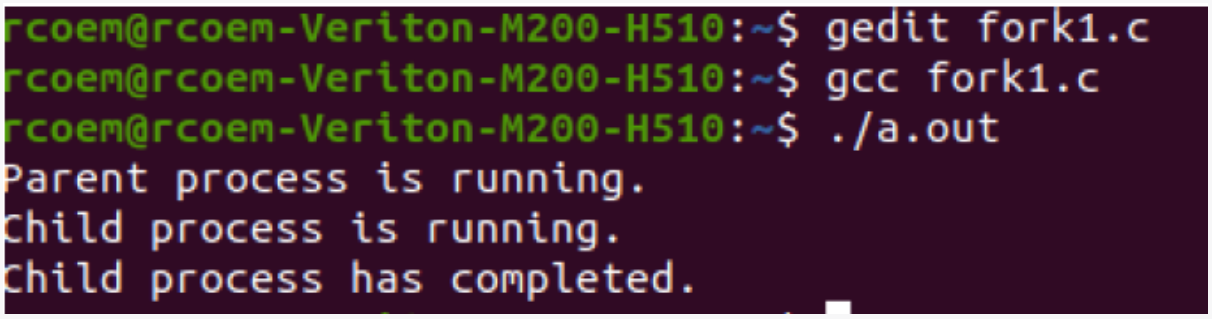
```
} // Iam the parent
printf ("Parent: Waiting for Child to complete.\n");
if ((ret = waitpid (pid, &status, 0)) == -1)
printf ("parent:error\n");
if (ret == pid)
printf ("Parent: Child process waited for.\n");
}
// child.c:
(source code snippet)
int main (int argc, char **argv)
{ int i, j;
long sum;
// Some arbitrary work done by the child
printf ("Child: Hello, World!\n");
for (j = 0; j < 30; j++ ) {
for (i =0; i < 900000; i++) {
sum = A * i + B * i * i + C;
sum %= 543;
}}
printf ("Child: Work completed!\n");
printf ("Child: Bye now.\n");
exit(0);
}
```

```
rcoem@rcoem-Veriton-M200-H510:~$ gcc parent.c -o parent
rcoem@rcoem-Veriton-M200-H510:~$ gcc child.c -o child
rcoem@rcoem-Veriton-M200-H510:~$ ./parent
Parent: Hello, World!
Parent: Waiting for Child to complete.
Child: Hello, World!
Child: Work completed!
Child: Bye now.
Parent: Child process waited for.
```

2) Fork system call eg 1:

```
(code snippet)
int main()
{
pid_t pid;
int count = 0;
```

```
pid = fork(); // create a child process
if (pid == 0) { // child process
printf("Child process is running.\n");
} else if (pid > 0)
{
// parent process
printf("Parent process is running.\n");
wait(NULL); // wait for child to complete
printf("Child process has completed.\n");
}
else {
// fork failed
printf("Fork failed.\n");
return 1;
}
return 0;
}
```



3) Fork system call eg 2:
(code snippet)

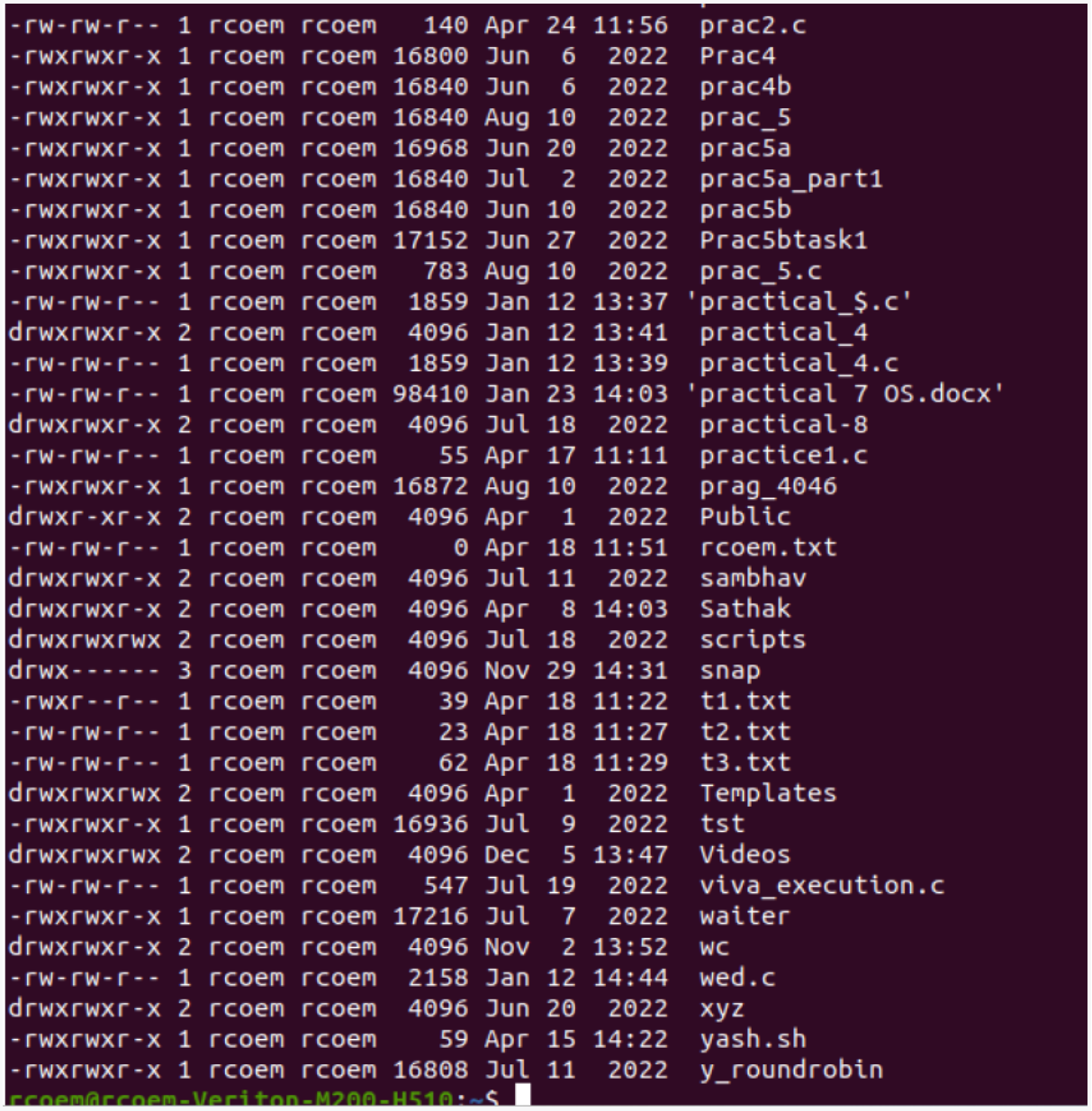
```
int main() {
pid_t pid;
int value = 0;
pid = fork(); // create a new process
if (pid == -1) {
printf("Fork failed\n");
exit(1);
}
else if (pid == 0) {
printf("I am the child process. My pid is %d\n", getpid());
value = 1;
}
```

```
else {
printf("I am the parent process. My pid is %d\n", getpid());
value = 2;
}
printf("I am process %d and my value is %d\n", getpid(), value);
return 0;
}
```

```
rcoem@rcoem-Veriton-M200-H510:~$ gedit fork2.c
rcoem@rcoem-Veriton-M200-H510:~$ gcc fork2.c
rcoem@rcoem-Veriton-M200-H510:~$ ./a.out
I am the parent process. My pid is 24974
I am process 24974 and my value is 2
I am the child process. My pid is 24975
I am process 24975 and my value is 1
```

4) Exec system call:

```
int main()
{
printf("Executing ls command using execvp() system call.\n");
char *args[] = {"ls", "-l", NULL};
execvp(args[0], args);
printf("execvp() system call failed.\n");
return 0;
}
```



5) Wait system call:
(code snippet)

```
int main()
{
pid_t pid;
int status;
pid = fork();
if (pid == 0) { // child process
printf("Child process is running.\n");
sleep(2);
printf("Child process is exiting.\n");
```

```
return 10;
}
else if (pid > 0) { // parent process
printf("Parent process is waiting for child to complete.\n");
wait(&status); // wait for child to complete and get its exit status
printf("Child exit status: %d\n", WEXITSTATUS(status));
}
else { // fork failed
printf("Fork failed.\n");
return 1;
} return 0;
}
```