Ex. 1 Parent Child Processes Creation using Fork and Communicating using Pipe

Dr S.Rajarajan APIII/CSE

Objective

You need to develop a program with Fork system call to create a pair of parent & child processes. Then let them communicate with each other using Pipe.

Fork

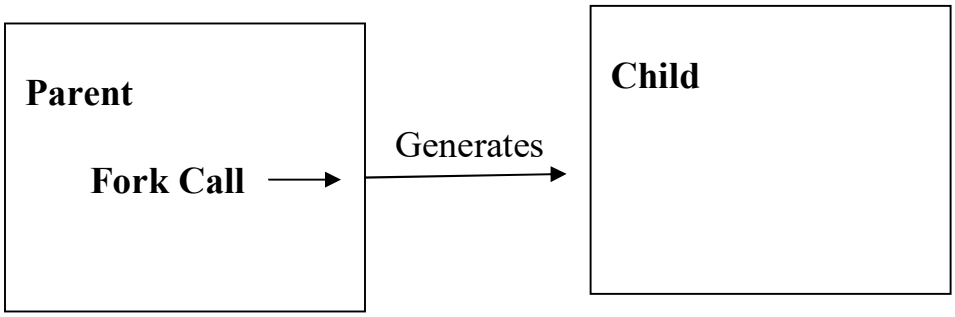
- In computing, particularly in the context of the Unix operating system and its work likes, **fork** is an operation whereby **a process creates a copy of itself** or the parent is replicated. It is usually a system call, implemented in the kernel. Fork is the primary (and historically, only) method of process creation on Unix-like operating systems.
- A new process is created with the *fork* system call.
- When *fork* is called, the operating system creates a new process: it assigns a new process entry in the **process table** and **clones** the content information from the current one.
- All file descriptors that are open in the parent will be open in the child
- The executable memory image is copied as well.
- As soon as the *fork* call returns, both the parent and child are now running at the same point in the program
- Since the parent and child are having exactly the same content, the only way to differentiate them is thorough the return value of fork call.
- The parent gets the **procsss ID** of the child that was just created. The child gets a return of 0. By checking the return value it could be determined whether what is executed now is the parent or the child.
- If fork returns -1 then the operating system was unable to create the process.
- The crucial thing to note with *fork* is that nothing is shared after the *fork*. Even though the child is running the same code and has the same files open, it maintains its own seek pointers (positions in the file) and it has its own copy of all memory. If a child changes a memory location, the parent won't see the change (and vice versa).

Pipe

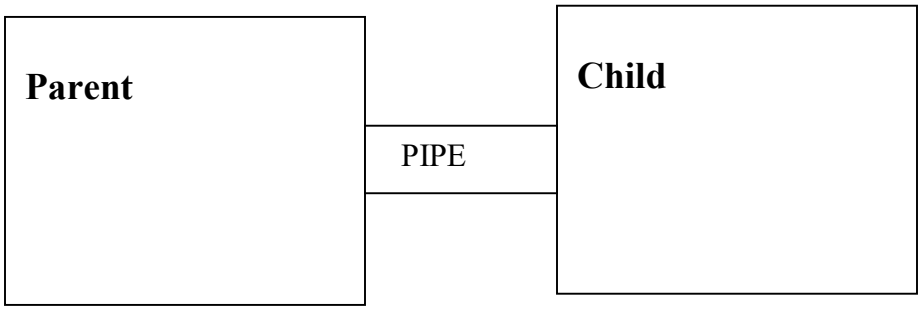
- A **pipe** is a system call that creates a unidirectional communication link between two file descriptors.
- The *pipe* system call is called with a pointer to an array of two integers.
- Upon return, the first element of the array contains the file descriptor that corresponds to the output of the pipe (**stuff to be read**).

- The second element of the array contains the file descriptor that corresponds to the input of the pipe (the place where you **write stuff**).
- **pipe**() creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by *filedes*. *filedes[0]* is for reading, *filedes[1]* is for writing.
- Whatever bytes are sent into the input of the pipe can be read from the other end of the pipe

1. Fork Command



2. **Pipe Command**



SAMPLE PROGRAM

Program 1: Without pipe

```
#include <stdio.h>
 main()
 { int  pid;
   printf("I'm the original process with PID %d and PPID %d. \n",  getpid(), getppid() );
   pid = fork();   /* Duplicate.  Child and parent continue from here */
   if ( pid!= 0 )  /* pid is non-zero, so I must be the parent */
   {
                printf("I'm the parent process with PID %d and PPID %d. \n",
                 getpid(), getppid() );
                printf("My child's PID is %d \n", pid );
   }
```

```
  else  /* pid is zero, so I must be the child */
  {
        printf("I'm the child process with PID %d and PPID %d. \n",
        getpid(), getppid() );
  }
```

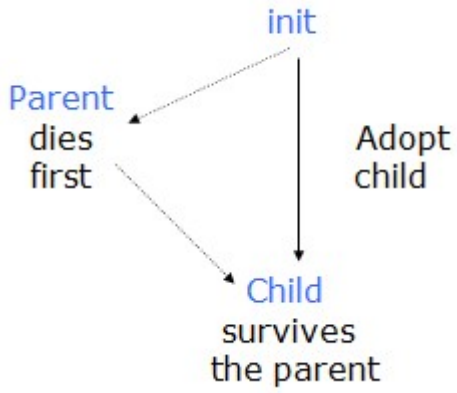
O/P

```
I'm the original process with PID 13292 and PPID 13273.
 I'm the parent process with PID 13292 and PPID 13273.
 My child's PID is 13293.
 I'm the child process with PID 13293 and PPID 13292.
 PID 13293 terminates.  ---> child terminates.
 PID 13292 terminates.  ---> parent terminates.
 $ _
```

Program 2: Orphan process

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

  main()
  {
    int pid;
    printf("I'm the original process with PID %d and PPID %d. \n",getpid(), getppid() );
    pid = fork();  /* Duplicate. Child and parent continue from here */
    if ( pid!= 0 )  /* Branch based on return value from fork() */
    {
       /* pid is non-zero, so I must be the parent */
       printf("I'm the parent process with PID %d and PPID %d. \n", getpid(), getppid() );
       printf("My child's PID is %d \n", pid );
    }
    else
    {
        /* pid is zero, so I must be the child */
       sleep(5);   /* Make sure that the parent terminates first */
       printf("I'm the child process with PID %d and PPID %d. \n",getpid(), getppid() );
    }
       printf("PID %d terminates. \n", getpid() );  /* Both processes execute this */
 }
```



Program 3: Zombie process

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

 main()
 {
  int pid;
  pid = fork();  /* Duplicate */
  if ( pid!= 0 )  /* Branch based on return value from fork() */
    {
      while (1) /* Never terminate, and never execute a wait() */
        sleep(1000);
    }
  else
    {
      exit(42);
     }
 }
```

Program 4: Parent waiting for child

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

 main()
 { int pid, status, childPid;
   printf("I'm the parent process and my PID is %d\n", getpid() );
   pid = fork();  /* Duplicate */
```

```
  if ( pid!=0 ) /* Branch based on return value from fork() */
   {
     printf("I'm the parent process with PID %d and PPID %d\n",  getpid(), getppid() );
     childPid = wait( &status );  /* Wait for a child to terminate. */
     printf(" A child with PID %d terminated with exit code %d \n",  childPid, status >> 8 );
   }
  else
   {
      printf("I'm the child process with PID %d and PPID %d \n", getpid(), getppid() );
      exit(42);  /* Exit with a silly number */
    }
  printf("PID %d terminates \n", getpid() );
}
```

Program 5: With pipe

```
#include<stdio.h>
#include<unistd.h>
int main()
{
        int p[2];
        int pid;
        char inbuf[10],outbuf[10];
        pipe(p);
        pid=fork();
         // Fork call to create child process //
        if(pid) //// Code of Parent process
        {
                printf("In parent process\n");
                printf("%d, getpid());
                printf("type the data to be sent to child");
                scanf("%s",outbuf);
                // Writing a message into the pipe
                write (p[1],outbuf, sizeof(outbuf));  //p[1] indicates write
                sleep(2); // To allow the child to run
                printf("after sleep in parent process\n");
        }
        else // Coding of child process //
        {
                printf("%d, getppid());
```

```
                printf("In child process\n");
                read(p[0],inbuf,10); // Read the content written by parent
                printf("the data received by the child is %s\n",inbuf);
                sleep(50);
        }
            return 0;
}
}
```

Sample problem

Pass a list of numbers from parent to the child, sort them and display.

Program 6: Create two child processes and communicate between parent and children using pipe

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
int pipefds[2];
int p;
int pid1,pid2;
int readmessage[1];
p = pipe(pipefds);
if (p == -1)
{
        printf("Unable to create pipe\n");
        return 1;
}
Pid1 = fork();// Child1
if (pid1 != 0) // Parent process
{
        pid2 = fork(); //Child2
        if (pid2 != 0) // Parent
        {
                for (int i =0;i<10;i++)
                {
                        read(pipefds[0],readmessage, sizeof(readmessage));
                        printf("%d is the number read by parent\n",readmessage[0]);
                 }
        }
```

```
        else //Child 2
        {
                for (int i =0; i<5;i++)
                {
                        int rand_n = rand()%100;
                        printf("Child Process 2 - Writing to pipe - Message 1 is %d\n", rand_n);
                        write(pipefds[1], &rand_n, sizeof(rand_n));
                }
        }
}
else //Child 1
{
        for (int i =0; i<5;i++)
        {
                int rand_n = rand()%100;
                printf("Child Process 1- Writing to pipe - Message 1 is %d\n", rand_n);
                write(pipefds[1], &rand_n, sizeof(rand_n));
        }
}
return 0;
}
```