CSE3241: Operating System and System Programming

Class-10

Sangeeta Biswas, Ph.D.
Assistant Professor
Dept. of Computer Science and Engineering (CSE)
Faculty of Engineering
University of Rajshahi (RU)
Rajshahi-6205, Bangladesh
E-mail: sangeeta.cse@ru.ac.bd / sangeeta.cse.ru@gmail.com

November 12, 2017

System Calls for Process Creation & Termination

- Header Files: unistd.h, sys/wait.h and stdlib.h

- System Calls:

1. **fork()**: for creating a child process.

   childPID = fork()

2. **getpid()**: for getting PID of the current process.

   myPID = getpid()

3. **wait()**: for waiting for the termination of child process.

   deadChildPID = wait(NULL)

4. **exit()**: for terminating a normal process.

   exit(0)

5. **execlp()**: for replacing the process's memory with a new program.

   execlp(exeFile, arg0, arg1,...)

System Calls for Shared Memory Model

- Header Files: sys/shm.h and sys/stat.h
- System Calls:
  1. **shmget()**: for allocating a shared memory segment into the address space of a process.

     shrSegID = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR)
  2. **shmat()**: for attaching the shared memory segment with a process.

     shrSegMem = (char *) shmat(shrSegID, NULL, 0)
  3. **shmdt()**: for detaching the shared memory segment with a process.

     shmdt(shrSegMem)
  4. **shmctl()**: for removing the shared memory segment from a process.

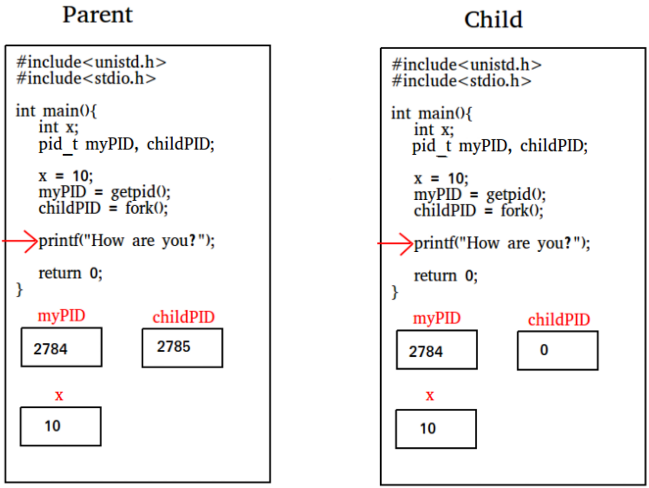
     shmctl(shrSegID, IPC_RMID, NULL)

System Call: fork() I

- When the system call, fork(), is executed successfully:
  - Linux makes two identical copies of address spaces, one for the parent process and the other for the child process.
  - Both processes starts their execution at the next statement following the fork() call.
  - Since both processes have identical but separate address spaces, those variables initialized before the fork() call have the same values in both address spaces.
  - Since every process has its own address space, any modifications will be independent of the others.

System Call: fork() II

- Both parent and child process will start their execution at the next statement following the fork() call.



Parent

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

int main(){
    int x;
    pid_t myPID, childPID;

    x = 10;
    myPID = getpid();
    childPID = fork();

    printf("How are you?");

    return 0;
}
```

myPID

2784

childPID

2785

x

10

Child

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

int main(){
    int x;
    pid_t myPID, childPID;

    x = 10;
    myPID = getpid();
    childPID = fork();

    printf("How are you?");

    return 0;
}
```

myPID

2784

childPID

0

x

10

Pipe I

- Pipes are a simple, synchronised way of passing information between processes.

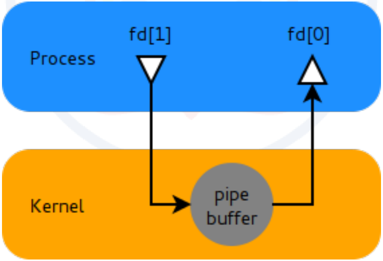
- There are two types of pipe:

1. Ordinary or unnamed pipe:
   - it cannot be accessed outside the process that creates it.
   - parent-child relationship is necessary between the communicating processes.
   - it exists only while the processes are communicating with one another.
   - communication is unidirectional.
2. Named pipe:
   - it can be accessed by any number of processes.
   - no parent-child relationship is necessary for communication.
   - it exists until it is deleted from the file sytem.
   - communication can be bidirectional.

Unnamed Pipe I

- Unnamed pipe is actually implemented using a piece of kernel memory.
- System call pipe() creates an unnamed pipe and provides two associated file descriptors:
  1. fd[0] for reading from the pipe
  2. fd[1] for writing to the pipe.

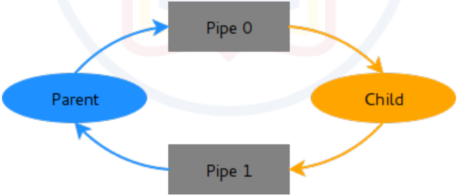
Figure: Taken from http://hzqtc.github.io/2012/07/linux-ipc-with-pipes.html



Unnamed Pipe II

- A unnamed pipe is unidirectional.
  - If the parent process write the pipe1 and then read from pipe1, it will get the same data written before.
  - And thats why two pipes are created, pipe1 for data flow from parent to child and pipe2 for data flow from child to parent.
  - Unused pipe descriptor needs to be closed.

Figure: Taken from http://hzqtc.github.io/2012/07/linux-ipc-with-pipes.html



System Calls for Unnamed Pipe

- Header Files: unistd.h and sys/stat.h
- System Calls:
  1. **pipe()**: for creating a pipe.

     pipeStatus = pipe(fd)
  2. **write()**: for writing message to the pipe.

     (write(fd[1], msgBuffer, msgLength)
  3. **read()**: for reading message from the pipe.

     read(fd[0], msgBuffer, msgLength)
  4. **close()**: for closing unused/used end.

     close(fd[0])

     close(fd[1])

References

P. B. Galvin A. Silbeschatz and G. Gagne.
Operating System Concepts.
John Wiley & Sons, 9 edition, 2012.