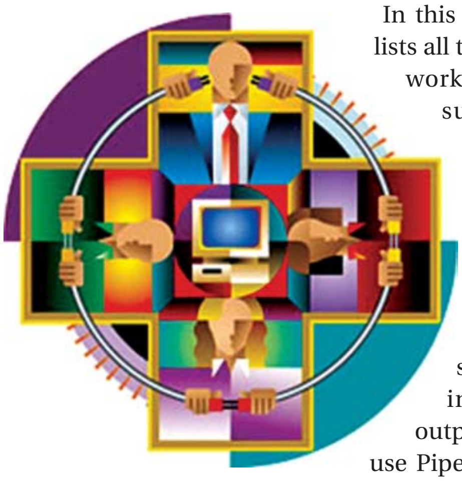
Communicate Between Related Processes Through Pipe Part–I

Pipe is a very useful Linux feature, as it enables different processes to communicate.

Pipe is one of the most significant contributions of UNIX to the development of operating systems. Pipe can be created in a C program by a pipe system call. It can be used in the shell as a command to link two or more commands, for example, if we want to get a number of files from a top-level directory to all its sub-directories. This can be done in the following manner.

From the shell, execute...

```
 "ls ñRl > temp;"
ìwc ñl < temp;î
```



In this case the first command lists all the files from the present working directory to all its sub-directories and redirects the output to *temp* file. The next command, *wc,* takes the *temp* file as an input, counts the number of lines, then displays the result on the shell. In this situation, instead of getting an output with two steps, if we use Pipe, we can get the output in a single step as...

```
ls ñRl | wc ñl.
```

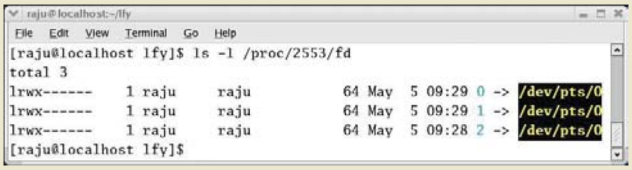
The "|" symbol denotes Pipe in the shell. Pipe connects two or more processes. For example, one process output goes as an input of another process, through Pipe. The Unix shell programmer cannot ignore the use of Pipe since its use is incredible in many situations.

Pipe is one of the simplest forms of inter-process communication. The data written to Pipe comes out in 'first in first out' order. Pipe uses the circular buffer to store data inside. It has a fixed size, generally the size of the data block. Pipe is half duplex, i.e., it can pass data in any one direction. If you want to use it for two-way communication, then you need two Pipes—one for sending and the other for receiving. Even though we can use Pipe for a single process, its use is very limited. Instead of using Pipe in a single process, if we use it to communicate between two related processes, then we could perform many tasks. Since Pipe is anonymous, we can use it only between related processes.

To understand Pipe, we need to have some knowledge of file descriptors, standard input, standard output, and redirection. Also about duplicate file descriptors using the *dup* system call, creating a new process by the *fork* system call and executing system commands through the *exec* family of calls.

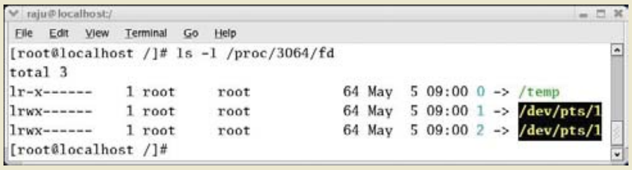
Fundamentals of dup, exec, fork and pipe

Whenever a process is created, the kernel creates a file descriptor table for the process. Its file descriptor table has three default entries, such as 0, 1 and 2. 0 is for standard input, 1 is for standard output and 2 is for standard error. If any file is opened by the process, a File Descriptor 3 will be created in the file descriptor table. After getting the file descriptor, file handling can be performed by using only the file descriptor that points to a file table. Unlike the file descriptor table, file table is a global table, which has an entry for the open files and points to an *inode* table. From the user's perspective, a file is identified by its name, but the kernel allocates an *inode* number for each file and identifies it with the number. Like the file table, the *inode* table is also a global table. Each file *inode* has an entry in the *inode* table and the *inode* entry points to where the file is



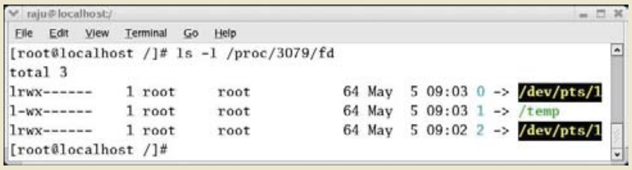
```
[raju@localhost lfy]$ ls -l /proc/2553/fd
total 3
lrwx------    1 raju     raju            64 May  5 09:29 0 -> /dev/pts/0
lrwx------    1 raju     raju            64 May  5 09:29 1 -> /dev/pts/0
lrwx------    1 raju     raju            64 May  5 09:28 2 -> /dev/pts/0
[raju@localhost lfy]$
```

Figure 1: List of standard file descriptors of 'ls -Rl' process



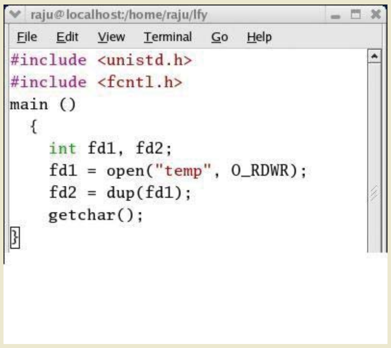
```
[root@localhost /]# ls -l /proc/3064/fd
total 3
lr-x------    1 root     root            64 May  5 09:00 0 -> /temp
lrwx------    1 root     root            64 May  5 09:00 1 -> /dev/pts/1
lrwx------    1 root     root            64 May  5 09:00 2 -> /dev/pts/1
[root@localhost /]#
```

Figure 2: Redirect STDOUT to temp file



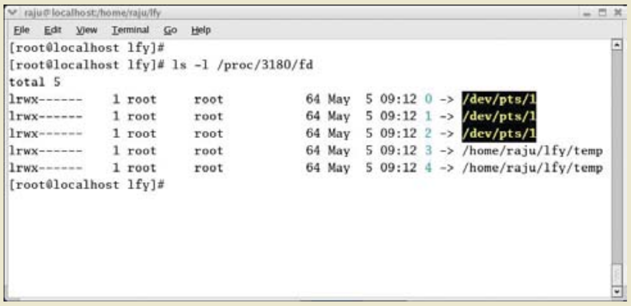
```
[root@localhost /]# ls -l /proc/3079/fd
total 3
lrwx------    1 root     root            64 May  5 09:03 0 -> /dev/pts/1
l-wx------    1 root     root            64 May  5 09:03 1 -> /temp
lrwx------    1 root     root            64 May  5 09:02 2 -> /dev/pts/1
[root@localhost /]#
```

Figure 3: Redirect STDIN to temp file



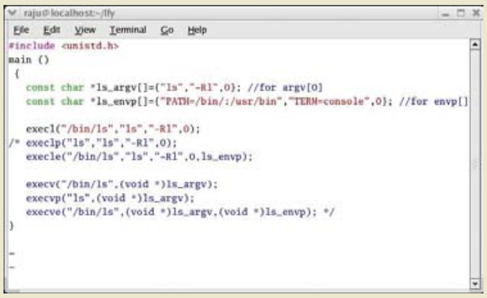
```
#include <unistd.h>
#include <fcntl.h>
main ()
  {
    int fd1, fd2;
    fd1 = open("temp", O_RDWR);
    fd2 = dup(fd1);
    getchar();
}
```

Figure 4a: Program for duplicating a file descriptor



```
[root@localhost lfy]#
[root@localhost lfy]# ls -l /proc/3180/fd
total 5
lrwx------    1 root     root            64 May  5 09:12 0 -> /dev/pts/1
lrwx------    1 root     root            64 May  5 09:12 1 -> /dev/pts/1
lrwx------    1 root     root            64 May  5 09:12 2 -> /dev/pts/1
lrwx------    1 root     root            64 May  5 09:12 3 -> /home/raju/lfy/temp
lrwx------    1 root     root            64 May  5 09:12 4 -> /home/raju/lfy/temp
[root@localhost lfy]#
```

Figure 4b: Shows new and old descriptors pointing to the same temp file



```
#include <unistd.h>
main ()
 {
   const char *ls_argv[]={"ls","-Rl",0}; //for argv[0]
   const char *ls_envp[]={"PATH=/bin/:/usr/bin","TERM=console",0}; //for envp[]

   execl("/bin/ls","ls","-Rl",0);
/* execlp("ls","ls","-Rl",0);
   execle("/bin/ls","ls","-Rl",0,ls_envp);

   execv("/bin/ls",(void *)ls_argv);
   execvp("ls",(void *)ls_argv);
   execve("/bin/ls",(void *)ls_argv,(void *)ls_envp); */
}
```

Figure 5: Program to implement exec family of calls

stored in the data block.

To visualise a file descriptor table, we can look at a simple command execution process. Figure 1 shows the long list of standard file descriptors of the *ls –Rl* process. Linux uses the *proc* file system to store the running processes, dynamic information and systems information.

Redirection by dup( )

When we want to find the number of files in a directory, we need to store the output of *ls –Rl* to a *temp* file. How do we redirect an output of a process to a file? The shell uses '>' or '<' for redirection. Let us see what will happen if we execute *ls –Rl* > *temp. ls –Rl* lists the files in the directory and the redirection operator > stores the output to the *temp* file. In this case, the standard output (File Descriptor 1) is closed before redirection and the *temp* file is created. Whenever a file is created, the kernel allocates a minimum possible value for the descriptor from the process's file descriptor table, so the *temp* file gets 1 as its file descriptor. This way, instead of showing output on the standard output monitor, it is redirected to the *temp* file. The other command *wc* < *temp* is working in an opposite manner. The standard input is closed and the program *wc* takes the input from the *temp* file, which is done by the redirection operator '<'.

Figures 2 and 3 illustrate the redirection with the *temp* file. Since Pipe is used to link two programs like "*ls –Rl* | *wc –l*", we will see later how it redirects the first process output as an input of another process.

A file descriptor can be duplicated by executing the *dup( )* system call. The *dup* function returns with the new file descriptor of the lowest available value from the file descriptor table. After that both the new and old descriptors can be used to access the same file.

Figures 4a and 4b explain the concept of the *dup* system call. Figure 4a is the program to duplicate the file descriptor and Figure 4b is the file descriptor table of the process. The main use of duplicating a file descriptor is to implement redirection of the input or output. This can be performed not only by using *dup* but also *fcntl*. But *dup* or *dup2* are more convenient than *fcntl*. The *fcntl* function is declared in *fcntl.h* and the *dup* is declared in *unistd.h*.

```
newfd = dup(oldfd);
```

This function copies *oldfd* to the first available number for *newfd*. This is equivalent to...

```
newfd = fcntl(oldfd, F_DUPFD, 0);
```

The other function…

```
dup2 (int oldfd, int newfd);
```

…closes the *oldfd* if necessary and copies the *oldfd* to *newfd.* If *oldfd* is invalid, then *dup2* doesn't do anything. The equivalent *fctnl* function is...

```
close(newfd);
fcntl (oldfd, F_DUPFD, newfd);
```

But *dup2* will do these two steps atomically. The *dup* call will work for all file types. The *newfd* and *oldfd* are sharing locks, file offset and flags. Both the functions return a new descriptor if execution is successful, else -1 is returned and set the *errno* appropriately. The possible error flags are:

EBADF –	*oldfd* is not an open file descriptor or *newfd* is out of the allowed range.
EMFILE –	the process already has the maximum number of file descriptors.
EINTR –	the *dup2* is interrupted by a signal.
EBUSY –	a race condition between open and dup calls. The best use of *dup2* is first close *newfd,* then call *dup2(oldfd, newfd).*

dup ( ) can be called after closing the standard output. Then the new file descriptor becomes 1 as standard output. The consequence of this is that instead of displaying the output to the monitor, it goes into the file. The same way, a program can get an input from a file instead of the standard input. Pipe leverages this functionality of the *dup* function when it redirects the standard output of one program to the input of another program.

Executing a program using *execl*

It is important to know how the shell executes *ls*. When we execute the *ls* program from a shell, it searches the *ls* executable file in one of the directories specified by the *PATH* environment variable, and finds whether the user has the permission to execute it. Then it creates a sub-process, which executes the *ls* program. The output of the process sent back to the parent process, and then it is terminated. Within a C program, if we want to run any executable program, then we need to use any one of the *exec* families of calls.

The *exec* family includes the five library functions listed below. All *exec* like functions, except *execve,* are wrapper routines defined in the C library and make use of *execve*, which is the only system call offered by Linux to deal with program execution.

- 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, char *const argv[]);
- int execvp(const char *file, char *const argv[]);

Irrespective of any one of the above calls, the kernel invokes the *execve* system call to perform the given task. The library functions are at the front end of the *execve* system call. All have the same functionality except in the way arguments are passed to these calls. *exec* functions replace the current process image with the specified executable file's text, variables, stack and heap portions. The *exec* functions are declared in *unistd.h*. Even though *exec* replaces the old process image with the new process image, some of the attributes remain unchanged. For example, the process id (pid), parent process id (ppid), group id (guid), pending alarms, present working directory (pwd), root directory, file mode creation mask, signal mask, and the elapsed processor time associated with the process. Signals pending on the calling process are cleared. Signals set to be caught by the calling process are reset to their default behaviour.

The syntax for the *execve* system call is like this...

```
int  execve(const  char  *filename,  char  *const  argv [],
char *const envp[]);
```

The file name of the first argument is the name of the executable file or shell script; *argv* is a pointer to an array of character pointers, which are the parameters to the file; and *envp* is a pointer to an array of character pointers, which are the environment of the executed program. If execution is successful, *execve* will not return, else -1 is returned and the *errno* is set appropriatly.

The five library functions are mainly of two types—*execl* and *execv*. *Execl, execlp* and *execle* take a variable number of arguments with a NULL pointer, but *execv* and *execvp* use an array of strings as their second argument, like all the command-line arguments with a single parameter. The 'v' in *execv*, as the name suggests, is the parameter and the address of a vector of pointers to command-line argument strings whose last component of the array must store the NULL value. The *execle* and *exeve* functions receive, as their last parameter, the address of an array pointer to environment strings. As usual, the last component of the array must be NULL. The functions, *execlp* and *execvp,* duplicate the actions of the shell in searching for an executable file if the specified file name is not specified by an absolute path name. The search path is specified in the environment by the PATH variable or else the default path*: /bin:/usr/bin* is used.

Let us see how to execute *ls –Rl* through the *execl* function.

```
execl("/bin/ls", "ls", 0);
```

When the process invokes the *execl* call, the kernel finds the file */bin/ls* in the file system and checks whether it is an executable file and whether the user has permission to execute it. If it is so, then the kernel overlaying the *ls* image with the old process image executes. The program that the process is executing is called its process image. Figure 5 is the program to execute the *ls –Rl* program and it further explains the syntax of the usage of the remaining *exec* calls.

Creating a new process by *fork ( )*

Pipe can be used within a process, but its use is very limited. The actual use of Pipe is to communicate between related processes. In Linux the *fork* system call is used to create a new process. Whenever a user request comes, the kernel creates a new process to satisfy the request. In this section, we see briefly the functionality of the *fork* system call.

When a process is created, a unique process id (pid) is allocated for it. During its lifetime, it has been allocated many resources and once its termination is reported to its parent process, then the resources, including its *pid*, are freed.

The term 'process' was first used by a predecessor of the UNIX operating system, i.e., the Multics operating system designers. Except for some of the system processes, which were created by the kernel—for example, the swapper process (pid = 0)—all other processes are created by the *fork* system call.

A system call is a function that causes the kernel to provide the required service for a program. The new process created by the *fork* system call is called a child process and the former is called a parent process. The child process is just a clone of the parent process except for its *pid*. Apart from the *pid*, the child inherits the environment, privileges and some of the resources like duplicating open file descriptors. The elapsed times for the child process are set to zero. However, some of the parent's attributes are not inherited by a child process—like file locks, alarms and the set of pending signals.

The syntax of the *fork* system call is...

```
int ret = fork( );
```

When we execute a *fork* system call, if it is successful it returns with two values, 0 and greater than 0. Zero is for the child process return value and >0 is the parent process return value. *Pid* 0 is the swapper process and is created internally by the kernel when the system is booted. This is the only process not created via the fork. The zero is allocated for the return value of the child process and the parent returns >0, which is the child's *pid*. The significance of the return value is that whenever child process wants to communicate to its parent, it can call the *getppid ( )* function since it has only one parent. But the parent may have many children, so it has to return the value of the child pid (>0). This is the only way for the parent to identify its children so that it can keep track of the activities of its own child processes.

If fork fails it returns -1 and sets an appropriate error flag. The possible error is *EAGAIN,* which is due to insufficient memory or that the call can't allocate memory or is not able to create a task structure. The *ENOMEM* error is due to insufficient memory. Once the child completes its execution, it has to give the exit status to the parent. If the parent exits before the completion of the child process, then the child process becomes an orphan process and the *init* process will take care of it. To synchronise parent and child process execution, *wait* or *waitpid* system call is invoked inside the parent process. *wait ( )* call waits for the termination of the child process and if the parent has many children, then the *waitpid( )* system call is used to specify waiting for a particular child. One of the arguments of *waitpid* is *pid.* If the argument is < -1, then the parent can wait for any child process whose process group id is equivalent to the absolute value of the *pid*. If the value is -1, then the parent waits for any child process, and if the value is 0, the parent process waits for any child process whose process gid is equivalent to that of the calling process. If the value is >0, then the parent process waits for the child process, whose *pid* is equal to the value of *pid*. Suppose the child completes its execution and waits for the parent, then immediately the kernel puts the child process into the zombie state. Here, all the resources of the child process are deallocated, except for its entry in the process table. Once the parent collects the exit status from the child, all the child's resources are freed.

When the kernel creates a new process, it allocates an entry in the process table, assigns a unique number as the *pid* of the process, makes a logical copy of the context of the parent process and the new *pid* returns to the parent process with a 0 return value to the child process. Linux uses the clone system call to implement the fork. The clone system call, in turn, calls *do_fork*. The actual work of copying the parent process image to the child process is done by this function, which is defined in *fork.c*.

Instead of copying the address space of the parent, Linux uses the Copy-On-Write (COW) technique for economical use of the memory page. The parent address space is not copied, but can be shared by both the parent and the child process, whereas the memory pages are marked as write protected. If the parent or the child wants to modify the pages, then a page fault is triggered by the memory management hardware and the process is interrupted. The kernel then copies the parent pages to the child process, so that both the parent and child have their own copies of the process image. The advantage of this technique is that the kernel can defer or prevent copying of a parent process address space, if required. This is because copying the entire address space of a current process makes process creation very slow and inefficient. In many cases, once a child process is created, it immediately will execute some program by calling one of the *exec* calls. In this case, the only overhead is duplication of the parent's page tables and the creation of a unique process descriptor for the child. One more optimisation is that when a new process is created, the kernel intentionally runs the child process first. In general, the child calls *exec* immediately, so this avoids the COW overhead. Otherwise, if the parent executes first, then the kernel unnecessarily makes its own copy of memory pages for both the parent and child. Later, the child overwrites the memory pages with the given executable file image. So the only overheads that fork leads to are the time and memory required to copy the parent's page table and create a new node in task structure for the child.

Like *fork, vfork* is also used to create a new process. *vfork* is implemented by the kernel through a special case of the *clone* system call. It is used to create new processes without copying the parent's page tables. Before the implementation of the COW technique, the *vfork* system call has been used for optimisation. It has the same effect as *fork*, except that it will not allow a separate copy of parent and child process images—instead, both can share the address space. The parent is blocked until the child completes its execution. If any one wants to write to the pages, then only the kernel creates separate copies of the images for both the parent and child. Since Linux has already implemented the COW technique and child execution first, it eliminates the necessity of *vfork* usage and, moreover, the *vfork* suspends the parent process execution until the child calls exec or exit. The only advantage to *vfork* is not copying the parent page table entries. But, currently, patches are available to add this functionality (in fork) to Linux. Even though the overhead is lower when we use *vfork*, the design is not good and the advantages of *vfork* can be achieved now by using fork itself. So the usage of *vfork* instead of *fork* is highly discouraged.

In the next article we will deal with the working procedure of Pipe and some library functions like *popen ()* or *system (),* that are available to perform Pipe's task. LFY

By: Dr B. Thangaraju is working as a tech lead in Embedded and Product Engineering Solutions (E&PE), Wipro Technologies, Bangalore.

Acknowledgement: The author would like to thank S. Jaisankar, Indian Institute of Science, Bangalore for his critical reading of the manuscript.