IPC

Pipe is one of the message passing IPC resource widely used on *nix operating system platforms. Pipes provide unidirectional interprocess communication channel. A pipe has a read end and a write end. Data written to the write end of a pipe can be read from the read end of the pipe.

pipes can be created using *pipe* api, which creates a new pipe and returns two file descriptors, one referring to the read end of the pipe, the other referring to the write end. Pipes can be used to create a communication channel between related processes

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

The array fd[2] returns two file descriptors referring to two ends of the created pipe: fd[0] for reading and fd[1] for writing. Data written to the pipe is buffered by the kernel until it is read by the read end. The communication channel provided by a pipe is a byte stream: there is no concept of message boundaries. it is not possible to seek on a pipe(lseek).

In Linux versions before 2.6.11, the capacity of a pipe was the same as the system page size (e.g., 4096 bytes on IA32). Since Linux 2.6.11, the pipe capacity is 65536 bytes.

The following program creates a pipe, and then forks to create a child process; the child inherits pipe file descriptors that refer to the same pipe. After fork() each process closes the descriptors that it doesn't need for the pipe . The parent then waits on pipe1 for data and child process sends a string message , for which parent sends response message through pipe2, and the child reads this string a byte at a time from the pipe and echoes it on standard output.

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

void client(int readfd, int writefd)
{
        char msg[100];
        int n;
        char eof = EOF;
        printf("%d enter msg to be sent to server: ", getpid());
        fgets(msg, 100, stdin);
        if (write(writefd, msg, strlen(msg) - 1) == -1){
                perror("error writing...");
                exit(0);
        }
        if ((n = read(readfd, msg, 100)) < 0) {
                perror("error reading...");
                exit(0);
        }
        msg[n] = '\0';
        printf("received from server: %s\n", msg);
}

void server(int readfd, int writefd)
```

```
{
        char msg[100];
        int n;
        if ((n = read(readfd, msg, 100)) < 0) {
                perror("error reading...");
        }
        msg[n] = '\0';
        printf("%d server received from client: %s\n", getpid(), msg);
        printf("enter msg to be sent to client: ");
        fgets(msg, 100, stdin);
        write(writefd, msg, strlen(msg) - 1);
}

int main()
{
        int pipe1fd[2], pipe2fd[2];
        int pid;

        /* create two pipes  */
        if (pipe(pipe1fd) == -1) {
                perror("pipe:");
                return 0;
        }
        if (pipe(pipe2fd) == -1) {
                perror("pipe:");
                return 0;
        }
   /* start child process and run client code in it */
        pid = fork();
        if (pid == 0) {
                close(pipe1fd[1]);      //close the write end of the first
pipe
                close(pipe2fd[0]);      //close the read end of the second
pipe
                client(pipe1fd[0], pipe2fd[1]);
                exit(0);
        }
        else {
                /* code that runs in parent ; runs as server */
                close(pipe1fd[0]); // close read end of the first pipe
                close(pipe2fd[1]); // close write end of the second pipe
                server(pipe2fd[0], pipe1fd[1]);
                wait(NULL);     //wait for child process to finish
                return 0;
        }
}
root@ubuntu:~# gcc pipe1.c
root@ubuntu:~# ./a.out
24978 enter msg to be sent to server: hello techveda
24977 server received from client: hello techveda
enter msg to be sent to client: hi! what have you learnt today ?
received from server: hi! what have you learnt today ?
root@ubuntu:~#
```

The program above creates two pipes and use them for two way communication between client and server

1. Create two pipes, pipe1 (pipe1fd[2]) and pipe2 (pipe2fd[2]).
2. call fork to create a child process and let child process run client code
3. In child process close write end of pipe1 (pipe1fd[1]) and read end of pipe2 (pipe2fd[0])
4. parent process close read end of pipe1 (pipe1fd[0]) and write end of pipe2 (pipe2fd[1])

If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read from the pipe will see end-of-file (read will return 0).If all file descriptors referring to the read end of a pipe have been closed, then a write will cause a SIGPIPE signal to be generated for the calling process.

popen and pclose

Linux provides two functions (popen and pclose) to create pipe from a process. It avoids the usage of pipe, fork, close, and wait.

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);

int pclose(FILE *stream);
```

popen() function starts a process by creating a pipe, forking and invoking a shell. The command argument accepts a shell command line, and the command is passed to /bin/sh using -c flag. The type argument expects either “r” or “w”. Note that since a pipe is unidirectional, we cannot pass both “r” and “w”, and the returned pipe stream is either read-only or write-only.

If the returned pipe stream is read-only, the calling process reads from the standard output. If the returned pipe stream is write-only, the calling process writes to the standard input. pclose() function waits for the associated process to terminate and returns the exit status of the command passed to popen().

Note that with popen() and pclose(), it is not convenient to create two pipes for two way communication. Below is a sample program using popen() and pclose(),

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

void createTest() {
    FILE *f;
    f = fopen("test", "w");
    fprintf(f, "hello techveda\n");
    fclose(f);
}

int main(int argc, char **argv) {
    char buf[100], command[100];
    FILE *pf;
    createTest();
    sprintf(command, "cat test");
```

```
    pf = popen(command, "r");    //read using cat
    while (fgets(buf, 100, pf) != NULL) {
        printf("%s", buf);
    }
    pclose(pf);
    return 0;
}
root@ubuntu:~# gcc pipe2.c
root@ubuntu:~# ./a.out
hello techveda
root@ubuntu:~#
```

PIPE_BUF

Writing less than PIPE_BUF bytes is atomic. Writing more than PIPE_BUT bytes may not be atomic: the kernel may interleave the data with data written by other processes. For example, if two processes trying to write “aaaaaa” and “bbbbb” respectively to the same pipe. If the writes are atomic, the content is either “aaaaaabbbbb” or “bbbbbaaaaaa”. But if it’s not atomic, the content can be something like “aaabbaabbba”.

In Linux, PIPE_BUF macro is defined at limits.h, and the program below can print it out.

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

int main() {
    printf("%d\n", PIPE_BUF);
    return 0;
}
root@ubuntu:~# gcc pipelim.c
root@ubuntu:~# ./a.out
4096
root@ubuntu:~#
```

Non Block Pipe

The default pipe blocks both reads and writes. A non blocking pipe can be created using fcntl() function with O_NONBLOCK flag.

FIFO's

Pipes can be used between processes that have common parent process. We refer to processes that have common parent process as related process. But for unrelated processes, pipe cannot be used, because one process has no way of referring to pipes have been created by another process.

When two unrelated processes share some information, an identifier must be associated with the shared information. Therefore, one process can create the IPC object and other processes can refer to the IPC object by the identifier. Linux provides named pipe (also called FIFO) to communication through pipe between two unrelated processes.

A FIFO special file (a named pipe) is similar to a pipe, except that it is accessed as part of the file system. It can be opened by multiple processes for reading or writing. When processes are exchanging data via the FIFO, the kernel passes all data internally without writing it to the file system. Thus, the FIFO special file has no contents on the file system; the file system entry merely serves as a reference point so that processes can access the pipe using a name in the file system.

FIFO's can be created using mkfifo function

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

mkfifo creates a special file with name pathname. mode specifies the special FIFO file's permissions. It is modified by the process's umask: the created file will have the permission (mode & ~umask).

Once a FIFO is created, it can be operated like a usual file with the exception that both ends of FIFO need to be open first before reading and writing can begin. In other words, opening a file for reading blocks until another process open it for writing, and vice versa.

Following is an example shows how FIFO's can be used for communication

programs communicating using FIFO's need to agree on the FIFO names. It is a common practice to define them in a header file that can be included by both of the program sources.

```
root@ubuntu:~# vim scfifo.h

#ifndef TEST_FIFO_H
#define TEST_FIFO_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define CS_FIFO_NAME "./cs"
#define SC_FIFO_NAME "./sc"

#define FIFO_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

#endif
```

Next, the server program creates two pipes for communication and opens one for read and one for write,

```
root@ubuntu:~# vim server.c
```

```
#include "scfifo.h"

void createfifo() {
    int rv;
    if ((rv = mkfifo(CS_FIFO_NAME, FIFO_MODE)) == -1) {
        perror("error creating cs fifo: ");
        return;
    }
    if ((rv = mkfifo(SC_FIFO_NAME, FIFO_MODE)) == -1) {
        perror("error creating sc fifo: ");
        return;
    }
}

void server(int readfd, int writefd) {
    char msg[100];
    int n;
    if ((n = read(readfd, msg, 100)) < 0) {
        perror("error reading...");
    }
    msg[n] = '\0';
    printf("%d server received from client: %s\n", getpid(), msg);
    printf("server: enter msg to be sent to client: ");
    fgets(msg, 100, stdin);
    write(writefd, msg, strlen(msg)-1);
}

int main() {
    int readfd, writefd;
    createfifo();
    readfd = open(CS_FIFO_NAME, O_RDONLY, 0);
    writefd = open(SC_FIFO_NAME, O_WRONLY, 0);
    server(readfd, writefd);
    close(readfd);
    close(writefd);
    return 0;
}
```

The client program also opens the two fifos for write and read.

```
root@ubuntu:~# vim client.c
#include "scfifo.h"

void client(int readfd, int writefd) {
    char msg[100];
    int n;
    char eof = EOF;
    printf("%d client: enter msg to be sent to server: ", getpid());
    fgets(msg, 100, stdin);
    if (write(writefd, msg, strlen(msg)-1) == -1) {
        perror("error writing...");
        exit(0);
    }
    if((n = read(readfd, msg, 100)) < 0){
        perror("error reading...");
        exit(0);
```

```
    }
    msg[n] = '\0';
    printf("client received from server: %s\n", msg);
}

void removefifo() {
    unlink(SC_FIFO_NAME);
    unlink(CS_FIFO_NAME);
}

int main() {
    int readfd, writefd;

    writefd = open(CS_FIFO_NAME, O_WRONLY, 0);
    readfd = open(SC_FIFO_NAME, O_RDONLY, 0);
    client(readfd, writefd);
    close(readfd);
    close(writefd);
    removefifo();
    return 0;
}
```

Note that if we swap the order of the two lines in the client source code, a deadlock will occur.

This is because the server blocks at opening CS_FIFO_NAME for reading and waits for client to open it for writing, if the client opens SC_FIFO_NAME for reading first, it also blocks and waits for server open it for writing. Both programs stuck forever.

Under Linux, opening a FIFO for read and write will succeed both in blocking and nonblocking mode. POSIX leaves this behavior undefined. This can be used to open a FIFO for writing while there are no readers available. PIPE_BUF limits apply to FIFOs.