Lecture 3: IPC--- FIFO

Refereneces for Lecture 3:

1) Unix Network Programming, W.R. Stevens, 1990,Prentice-Hall, Chapter 3.
2) Unix Network Programming, W.R. Stevens, 1999,Prentice-Hall, Chapter 2-6.

Definition:

A FIFO is similar to a pipe. A FIFO (First In First Out) is a one-way flow of data. FIFOs have a name, so unrelated processes can share the FIFO. FIFO is a named pipe. This is the main difference between pipes and FIFOs.

Creat: A FIFO is created by the **mkfifo** function:

#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname*, mode_t *mode*);
pathname – a UNIX pathname (path and filename). The name of the FIFO
mode – the file permission bits.

FIFO can also be created by the **mknod** system call,
e.g., mknod("fifo1", S_IFIFO|0666, 0) is same as mkfifo("fifo1", 0666).

Open: mkfifo tries to create a new FIFO. If the FIFO already exists, then an EEXIST error is returned. To open an existing FIFO, use **open()**, **fopen()** or **freopen()**

Close: to close an open FIFO, use close(). To delete a created FIFO, use unlink().

Properties: (**Some** of them are also applicable to PIPES)

1) After a FIFO is created, it can be opened for read or write.
2) Normally, opening a FIFO for read or write, it blocks until another process opens it for write or read.
3) **A read gets as much data as it requests or as much data as the FIFO has, whichever is less.**
4) **A write to a FIFO is atomic, as long as the write does not exceed the capacity of the FIFO. The capacity is at least 4k[①].**

① **Why?**

(1)The main operations (system calls) on PIPE or FIFO are write(...,length) and read(...,length).
(2)The length of write() or read() should not exceed 4k if we want the code portable among different unix systems.
(3) That is, all systems should guarantee write(...,length) or read(...,length) correctly executable when length <= 4k.
(4) **Accordingly, the capacity of PIPE/FIFO should be at least 4k.**
(5) Statements 1 and 2 are excerpted from textbooks.

5) **One step further :How to verify the minimum capacity of a FIFO ? I have verified that the capacity of a PIPE or a FIFO on my Unix System is 9k.**
6) **Blocked if read from an empty FIFO, or write to a full FIFO.**
7) **The O_NDELAY flag or O_NONBLOCK flag can be set for FIFO to affect the behavior of various operations. This prevents accesses to the FIFO from blocking. See the Table below.**
   Example: how to set the flags?
   writefd=open(FIFO1, O_WRONLY | O_NONBLOCK, 0);

For a pipe, however, fcntl() must be used to set this option. This is because pipes are opened as part of the pipe() system call:

```
int flags;
flags=fcntl(fd, F_GETFL,0); /* get the flags */
flag |= O_NONBLOCK;
fcntl(fd, F_SETFL, flags); /* set the new flagts */
```

8) **The best way to handle a SIGPIPE is to igore the signal, The write will then fail with an EPIPE. The default behavior for SIGPIPE is to terminate the process.**

Above properties are summarized in:

Operation	Existing opens of pipe or FIFO	Blocking (default)	O_NONBLOCK set
Open FIFO for reading	FIFO open for writing	Returns OK	Returns OK
	FIFO not open for writing①	**Blocks until FIFO is opened for writing**	Returns OK④
Open FIFO for writing	FIFO open for reading	Returns OK	Returns OK
	FIFO not open for reading①	**Blocked until FIFO is opened for reading**	Returns an error of ENXIO③
Read empty pipe of FIFO	Pipe or FIFO open for writing	Blocked until there is data or the pipe or FIFO is closed for writing	Returns an error of EAGAIN③
	Pipe or FIFO not open for writing②	Read returns 0 (EOF)	Read return 0 (EOF)
Write to pipe or FIFO	Pipe or FIFO open for reading	Return OK	Return OK
	Pipe or FIFO is full	**Blocked until space is available, then write data**	Returns an error of EAGAIN③
	Pipe or FIFO not open for reading②	**SIGPIPE generated, write process terminated**	Returns an error of EPIPE③

Notes: ① FIFO must be opened by two processes; one opens it as reader on one end, the other opens it as sender on the other end. The first/earlier opener has to wait until the second/later opener to come. This is somewhat like a hand-shaking.

② If one end of a PIPE/FIFO is being read or written, but the other end is not open for writing or reading, that is usually because the other end was once opened but now is closed.

③ When a system call returns –1 on error, a specific reason or error number is stored in *errno*, such as EEXIT, ENXIO, EAGAIN. The value of such symbolic constants are defined in <sys/errno.h>. Whenever an error occurs, you can use perror( ) to print a system error message describing the last error encountered during a system call or library function.

④ Only in this case, O_NONBLOCK makes the originally blocked operation return immediately without error.

Examples: use FIFOs not PIPEs to implement the IPC channels so that any two processes/programs are capable of IPC as long as they reside on the same host.

```
/* use FIFOs, not PIPES to implement the simple client-server model. */
#include      <sys/types.h>
#include      <sys/stat.h>
#include      <sys/errno.h>
extern int    errno;
#define  FIFO1   "/tmp/fifo.1"                          fifo.h
#define  FIFO2   "/tmp/fifo.2"
#define  PERMS       0666     /* octal value*/

main()
{     int  childpid, readfd, writefd;
      if ( (mkfifo(FIFO1, PERMS) < 0) && (errno != EEXIST))
            err_sys("can't create fifo 1: %s", FIFO1);
      if ( (mkfifo(FIFO2, PERMS) < 0) && (errno != EEXIST)) {
            unlink(FIFO1);
            err_sys("can't create fifo 2: %s", FIFO2);     }

      if ( (childpid = fork()) < 0) {err_sys("can't fork");      }
           else if (childpid > 0) {                  /* parent */
            if ( (writefd = open(FIFO1, 1)) < 0)
                  err_sys("parent: can't open write fifo");
            if ( (readfd = open(FIFO2, 0)) < 0)
                  err_sys("parent: can't open read fifo");
            client(readfd, writefd);
            while (wait((int *) 0) != childpid)/* wait for child */
                  ;
            close(readfd);                                          parent process
            close(writefd);
            if (unlink(FIFO1) < 0)
                  err_sys("parent: can't unlink %s", FIFO1);
            if (unlink(FIFO2) < 0)
                  err_sys("parent: can't unlink %s", FIFO2);
            exit(0);

      } else {                    /* child */
            if ( (readfd = open(FIFO1, 0)) < 0)
                  err_sys("child: can't open read fifo");
            if ( (writefd = open(FIFO2, 1)) < 0)
                  err_sys("child: can't open write fifo");
            server(readfd, writefd);                                child process
            close(readfd);
            close(writefd);
            exit(0);
      }
}
```

```
/*Revise the above program into 2 separate programs, without using fork() . One is for the
  client; the other is for the server.*/
include  "fifo.h"

main( )
{
       int   readfd, writefd;

       /*
         * Open the FIFOs.    We assume the server has already created them.
         */

       if ( (writefd = open(FIFO1, 1)) < 0)
              printf("client: can't open write fifo: %s", FIFO1);
       if ( (readfd = open(FIFO2, 0)) < 0)
              printf("client: can't open read fifo: %s", FIFO2);

       client(readfd, writefd);

       close(readfd);
       close(writefd);

       /*
         * Delete the FIFOs, now that we're finished.
         */

       if (unlink(FIFO1) < 0)
              printf("client: can't unlink %s", FIFO1);
       if (unlink(FIFO2) < 0)
              printf("client: can't unlink %s", FIFO2);

       exit(0);
}
```

```
/* Program for server */
#include      "fifo.h"

main()
{
       int  readfd, writefd;

       /*
        * Create the FIFOs, then open them -- one for reading and one for writing.
        */

       if ( (mknod(FIFO1, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST))
              printf("can't create fifo: %s", FIFO1);
       if ( (mknod(FIFO2, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST)) {
              unlink(FIFO1);
              printf("can't create fifo: %s", FIFO2);
       }

       if ( (readfd = open(FIFO1, 0)) < 0)
              printf("server: can't open read fifo: %s", FIFO1);
       if ( (writefd = open(FIFO2, 1)) < 0)
              printf("server: can't open write fifo: %s", FIFO2);

       server(readfd, writefd);

       close(readfd);
       close(writefd);

       exit(0);
}
```

Question: if we reverse the order of 2 open operations in server (i.e. the first call to open is for FIFO2 instead of FIFO1), then what would happen?

One step further: Consider such a possible problem: the server has done everything and closed the 2 FIFOs while the client has not finished the reading job. As a result, the client cannot read the remaining data in the FIFO since the other end has been closed. This problem may also occur in PIPE; see example in Lecture 2. What is worse, we cannot distinguish it from the normal transfer end since read(…) of the client will also return 0(EOF) in this case. One solution is that the client will inform the server via the 1st FIFO when job is done, and then the server will close everything. Fortunately our examples are very small so that you will not drop into this trouble.