CSC 4304 - Systems Programming
Fall 2008

LECTURE - XXI
INTERPROCESS COMMUNICATION

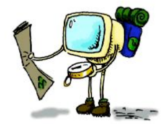
Tevfik Koşar

Louisiana State University

November 25th, 2008

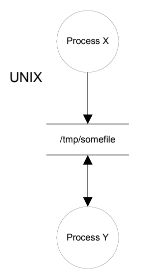
Roadmap

- Interprocess Communication
  - Pipes
  - FIFOs
  - Message Queues



Interprocess Communication

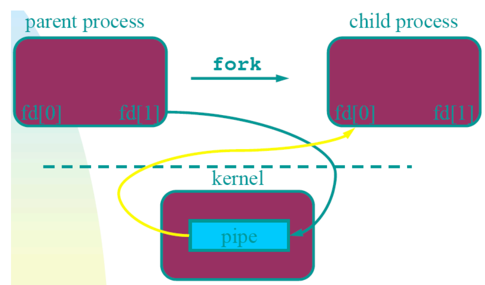
In the old days:



Px >/tmp/somefile
Py < /tmp/somefile

Interprocess Communication

Using Pipes:



What's a Pipe?

- A pipe is an interface between two processes that allows those two processes to communicate (i.e., pass data back and forth)
- A pipe connects the STDOUT of one process (writer) and the STDIN of another (reader)
- A pipe is represented by an array of two file descriptors, each of which, instead of referencing a normal disk file, represent input and output paths for interprocess communication
- Examples:
  ls | sort
  ypcat passwd | awk –F: '{print $1}' | sort
  echo "2 + 3" | bc

Pipe Facts

- Pipes are half duplex by default, meaning that one pipe is opened specifically for unidirectional writing, and the other is opened for unidirectional reading (i.e., there is a specific "read" end and "write" end of the pipe)
- The net effect of this is that across a given pipe, only one process does the writing (the "writer"), and the other does the reading (the "reader")
- If two way communication is necessary, two separate pipe() calls must be made, or, use SVR5's full duplex capability (stream pipes)

How to Create a Pipe?

```
#include <unistd.h>

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

Returns: 0 if OK, -1 otherwise

- filedes represents the pipe, and data written to filedes[1] (think STDOUT) can be read from filedes[0] (think STDIN)
- pipe() returns 0 if successful
- pipe() returns –1 if unsuccessful, and sets the reason for failure in errno (accessible through perror())

Example

```
int main(void)
{
    int      n, fd[2];
    pid_t    pid;
    char line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");

    else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);

    } else {              /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }

    exit(0);
```


Traditional Pipes

- How would you mimic the following command in a program:
  $ ls /usr/bin | sort

1. Create the pipe
2. associate stdin and stdout with the proper read/write pipes via dup2
3. close unneeded ends of the pipe
4. call exec()


```
main(int ac, char *av[])
{
        int     thepipe[2], newfd, pid;*/
        if ( ac != 3 ){fprintf(stderr, "usage: pipe cmd1 cmd2\n");exit(1);}

        if (pipe(thepipe) == -1){perror( "cannot create pipe"); exit(1); }

        if ((pid = fork()) == -1){fprintf(stderr,"cannot fork\n"); exit(1);}

         /*
         *      parent will read from reading end of pipe
         */

        if ( pid > 0 ){                 /* the child will be av[2]      */
                close(thepipe[1]);      /* close writing end            */
                close(0);               /* will read from pipe          */
                newfd=dup(thepipe[0]);  /* so duplicate the reading end */
                if ( newfd != 0 ){      /* if not the new stdin..       */
                        fprintf(stderr,"Dupe failed on reading end\n");
                        exit(1);
                }
                close(thepipe[0]);      /* stdin is duped, close pipe   */
                execlp( av[2], av[2], NULL);
                exit(1);                /* oops                         */
        }
```



```
        /*
         *      child will write into writing end of pipe
         */
        close(thepipe[0]);      /* close reading end            */
        close(1);               /* will write into pipe         */
        newfd=dup(thepipe[1]);  /* so duplicate writing end     */
        if ( newfd != 1 ){      /* if not the new stdout..      */
                fprintf(stderr,"Dupe failed on writing end\n");
                exit(1);
        }
        close(thepipe[1]);      /* stdout is duped, close pipe  */
        execlp( av[1], av[1], NULL);
        exit(1);                /* oops                         */
}
```


Easy way Pipes: popen()

```
#include <stdio.h>

FILE *popen(const char *cmd, const char *type);
```

Returns: file pointer if OK, NULL otherwise

```
int pclose(FILE *fp);
```

Returns: termination status *cmd* or -1 on error

- The simplest way (and like system() vs. fork(), the most expensive way) to create a pipe is to use popen(): ptr = popen(“/usr/bin/ls”, “r”);
- popen() is similar to fopen(), except popen() returns a pipe via a FILE *
- you close the pipe via pclose(FILE *);

popen()

- When called, popen() does the following:
  1. creates a new process
  2. creates a pipe to the new process, and assigns it to either stdin or stdout (depending on char * type)
     "r": you will be reading *from* the executing command
     "w": you will be writing *to* the executing command
  3. executes the command cmd via a bourne shell

popen(): read vs write

fp = popen(*command*, "r")

fp = popen(*command*, "w")

Example

```
int main(int argc, char *argv[])
{
    char line[MAXLINE];
    FILE *fpin, *fpout;

    if (argc != 2)
        err_quit("usage: %s <pathname>", argv[0]);
    if ( (fpin = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);

    if ( (fpout = popen(argv[2], "w")) == NULL)
        err_sys("popen error");

    while (fgets(line, MAXLINE, fpin) != NULL) {
        if (fputs(line, fpout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (ferror(fpin))
        err_sys("fgets error");
    if (pclose(fpout) == -1)
        err_sys("pclose error");
    exit(0);
}
```


But..

- One thing is in common between all the examples we've seen so far:

  All our examples have had *shared file descriptors*, shared from a parent processes forking a child process, which *inherits* the open file descriptors as part of the parent's environment for the pipe

- Question: How do two entirely *unrelated* processes communicate via a pipe?

FIFOs: Named Pipes

- FIFOs are "named" in the sense that they have a name in the filesystem
- This common name is used by two separate processes to communicate over a pipe
- The command mknod can be used to create a FIFO:

  mkfifo MYFIFO (or "mknod MYFIFO p")
  ls –l
  echo "hello world" >MYFIFO &
  ls –l
  cat <MYFIFO

Creating FIFOs in Code

```
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);
```
Returns: 0 if OK, -1 otherwise

- path is the pathname to the FIFO to be created on the filesystem
- mode is a bitmask of permissions for the file, modified by the default umask
- mkfifo returns 0 on success, -1 on failure and sets errno (perror())
- mkfifo("MYFIFO", 0666);

Example

```
int main(void)
{
    int      fdread, fdwrite;

    unlink(FIFO);
    if (mkfifo(FIFO, FILE_MODE) < 0)
        err_sys("mkfifo error");

    if ( (fdread = open(FIFO, O_RDONLY | O_NONBLOCK)) < 0)
        err_sys("open error for reading");
    if ( (fdwrite = open(FIFO, O_WRONLY)) < 0)
        err_sys("open error for writing");

    clr_fl(fdread, O_NONBLOCK);

    exit(0);
}
```


NONBLOCKING FIFO

- O_NONBLOCK
  - NO → an open for read-only blocks until some other process opens the FIFO for writing (write-only as well).
  - Yes → an open for read-only always returns, while that for write-only returns with an error (errno=ENXIO) if there is no reader.

Message Queues

- A Message Queue is a linked list of message structures stored inside the kernel's memory space and accessible by multiple processes
- Synchronization is provided automatically by the kernel
- New messages are added at the end of the queue
- Each message structure has a long *message type*
- Messages may be obtained from the queue either in a FIFO manner (default) or by requesting a specific *type* of message (based on *message type*)

Message Structure

- Each message structure must start with a long message type:

```
struct mymsg {
    long msg_type;
    char mytext[512]; /* rest of message */
    int somethingelse;
    ....
};
```


Message Queue Limits

- Each message queue is limited in terms of both the maximum number of messages it can contain and the maximum number of bytes it may contain
- New messages cannot be added if *either* limit is hit (new writes will normally block)
- On linux, these limits are defined as (in /usr/include/linux/msg.h):
  - MSGMAX 8192 /*total number of messages */
  - MSBMNB 16384 /* max bytes in a queue */

Creating a Message Queue

- #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/msg.h>
  int msgget(key_t key, int msgflg);
- The key parameter is either a non-zero identifier for the queue to be created or the value IPC_PRIVATE, which guarantees that a new queue is created.
- The msgflg parameter is the read-write permissions for the queue OR'd with one of two flags:
  - IPC_CREAT will create a new queue or return an existing one
  - IPC_EXCL added will force the creation of a new queue, or return an error

Writing to a Message Queue

- int msgsnd (int msqid, const void * msg_ptr, size_t msg_size, int msgflags);

- msgqid is the id returned from the msgget call
- msg_ptr is a pointer to the message structure
- msg_size is the size of that structure
- msgflags defines what happens when no message of the appropriate type is waiting, and can be set to the following:
  - IPC_NOWAIT (non-blocking, return –1 immediately if queue is empty)_

Reading from a Message Queue

- int msgrcv(int msqid, const void * msg_ptr, size_t msg_size, long msgtype, int msgflags);

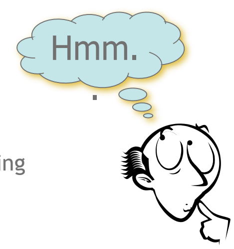
- msgqid is the id returned from the msgget call
- msg_ptr is a pointer to the message structure
- msg_size is the size of that structure
- msgtype is set to:

= 0	first message available in FIFO stack
> 0	first message on queue whose type equals type
< 0	first message on queue whose type is the lowest value less than or equal to the absolute value of msgtype

- msgflags defines what happens when no message of the appropriate type is waiting, and can be set to the following:
  - IPC_NOWAIT (non-blocking, return –1 immediately if queue is empty)

Summary

- Interprocess Communication
  - Pipes
  - FIFOs
  - Message Queues



- Next Lecture: Network Programming

- Read Ch.14 from Stevens
- Project-3 out today, due December 7th

Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), J.Kim (KAIST), and J. Schaumann (SIT).