CSE 344 System Programming

Week 6

*Introduction to **I**nter**P**rocess **C**ommunication (IPC)*

- *Pipes*
- *FIFOs*

Interprocess Communication

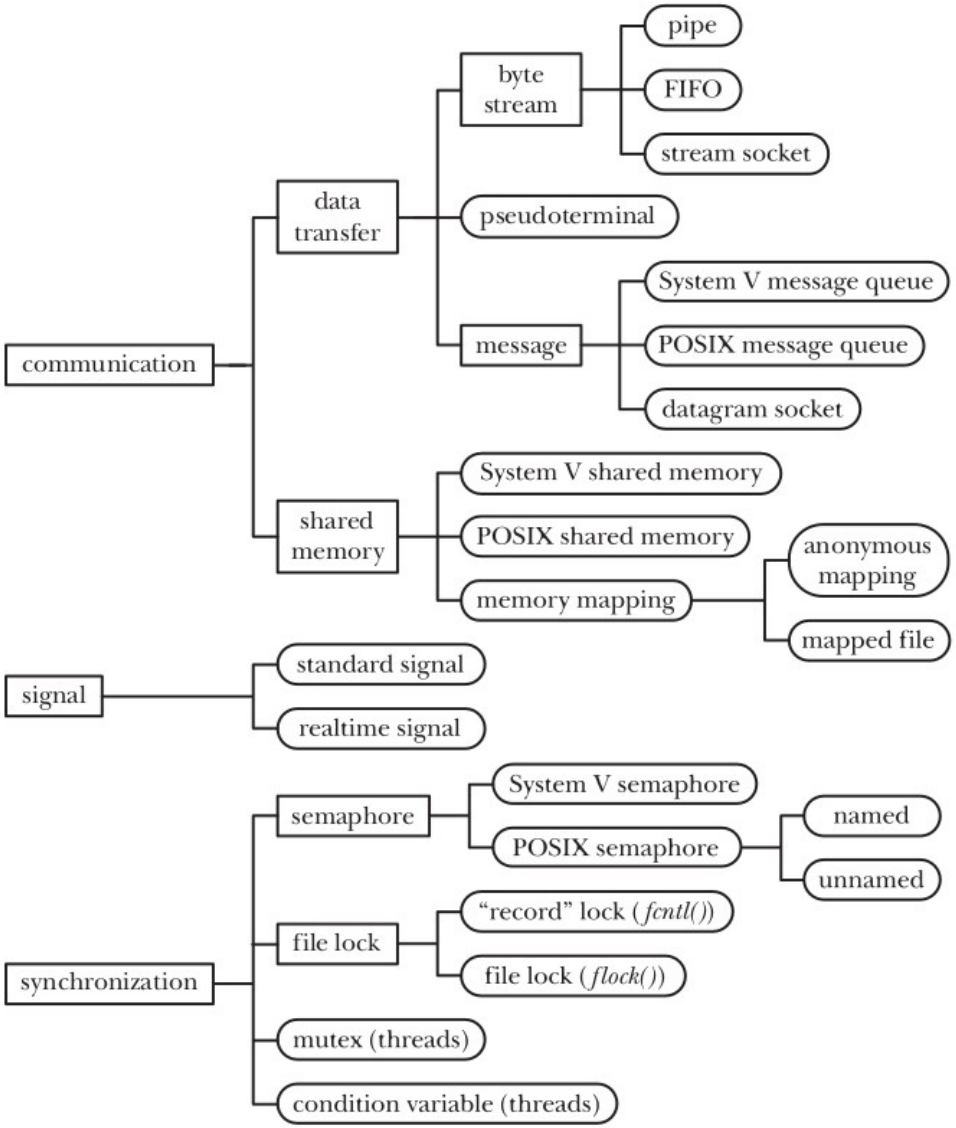


Figure 43-1: A taxonomy of UNIX IPC facilities

Interprocess Communication

An overview of communication options:

1) Shared memory permits processes to communicate by simply reading and writing to a shared memory page.

2) Mapped memory is similar to shared memory, except that it is associated with a file in the filesystem.

3) Pipes permit sequential communication from one process to a related process.

4) FIFOs are similar to pipes, except that unrelated processes can communicate because the pipe is given a name in the filesystem.

5) Sockets support communication between unrelated processes even on different computers.

and more (message queues, etc)..

Interprocess Communication

IPC criteria:

- Whether they restrict communication to related processes (processes with a common ancestor), to unrelated processes sharing the same file system, or to any computer connected to a network

- Whether a communicating process is limited to only writing data or only reading data

- The number of processes permitted to communicate

- Whether the communicating processes are synchronized by the IPC; for example, a reading process halts until data is available to read

Interprocess Communication

An overview of synchronization options:

1) **Semaphore**: a kernel maintained non-negative integer that can solve **many of** the commonly encountered synchronization issues.

2) **File locks**: are a synchronization method explicitly designed to coordinate the actions of multiple processes operating on the same file.

3) **Mutexes & condition variables (i.e. monitors)**: are intended to be used between threads and can solve (in theory) **any** synchronization issue.

More on these after the midterm exam/project.

Interprocess Communication

IPC continues to grow beyond POSIX and UNIX and has led to the development of modern technologies such as:

- Remote procedure calls (Java RMI, JSON-RPC, SOAP, .net remote)
- Distributed object models (CORBA, sessions)

and many more.

Interprocess Communication

Pipes are the oldest Unix IPC method.

A pipe is a communication device that permits **unidirectional** communication. Data written to the “write end” of the pipe is read back from the “read end.”

Pipes are serial devices; the data is always read from the pipe in the same order it was written.

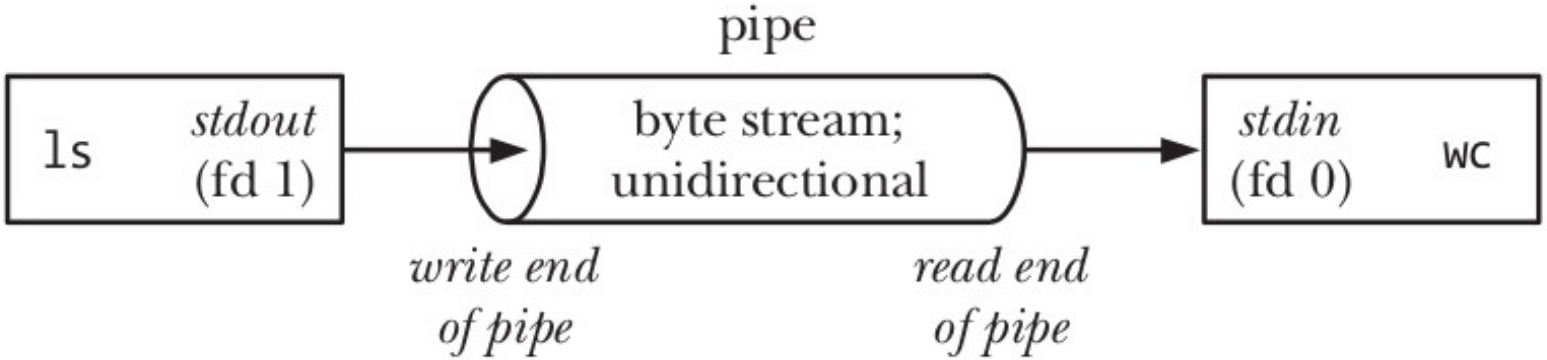
Typically, a pipe is used to communicate between two threads in a single process or between parent and child processes.

Interprocess Communication

In a shell, the symbol | creates a pipe. For example, this shell command causes the shell to produce two child processes, one for `ls` and one for `wc`:

```
% ls | wc -l
```

The shell also creates a pipe connecting the standard output of the `ls` subprocess with the standard input of the `wc` process. The filenames listed by `ls` are sent to `wc` in exactly the same order as if they were sent directly to the terminal.



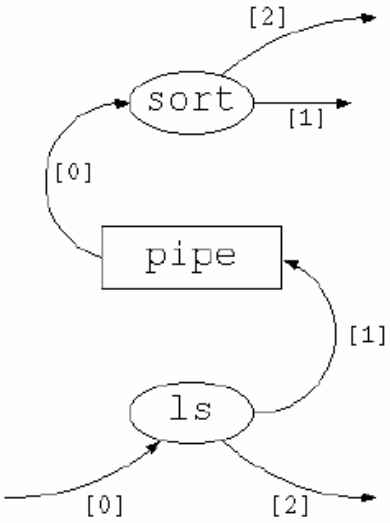
Interprocess Communication

- The following command line inputs have the same effect

```
#> ls -l > my.file
#> sort -n < my.file
```

The output of `ln -l` is written to myfile
`sort` command uses myfile as an input

```
#> ls -l | sort -n
```



`sort`

file descriptor table

[0]	`pipe` *read*
[1]	*standard output*
[2]	*standard error*

`ls`

file descriptor table

[0]	*standard input*
[1]	`pipe` *write*
[2]	*standard error*

Interprocess Communication

A pipe is simply a buffer maintained in kernel memory. This buffer has a finite and limited capacity (often 64KB).

If the writer process writes faster than the reader process consumes the data, and if the pipe cannot store more data, the writer process **blocks** until more capacity becomes available.

If the reader tries to read but no data is available, it **blocks** until data becomes available. Thus, the pipe automatically synchronizes the two processes.

Interprocess Communication

The `pipe()` system call creates a new pipe.

```
#include <unistd.h>
int pipe(int filedes [2]);
```

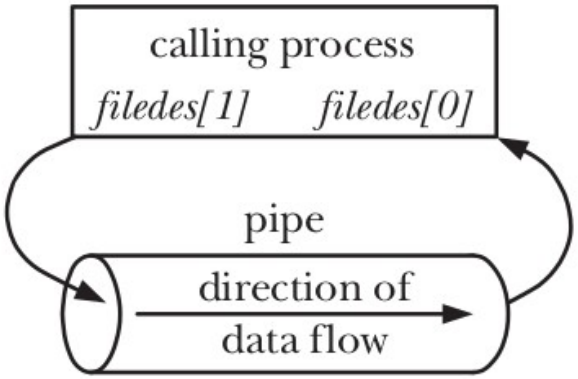


Figure 44-2: Process file descriptors after creating a pipe

Returns 0 on success, or –1 on error.

First supply an integer array of size 2. Then the call to `pipe()` stores the reading file descriptor in array position 0 and the writing file descriptor in position 1.

Interprocess Communication

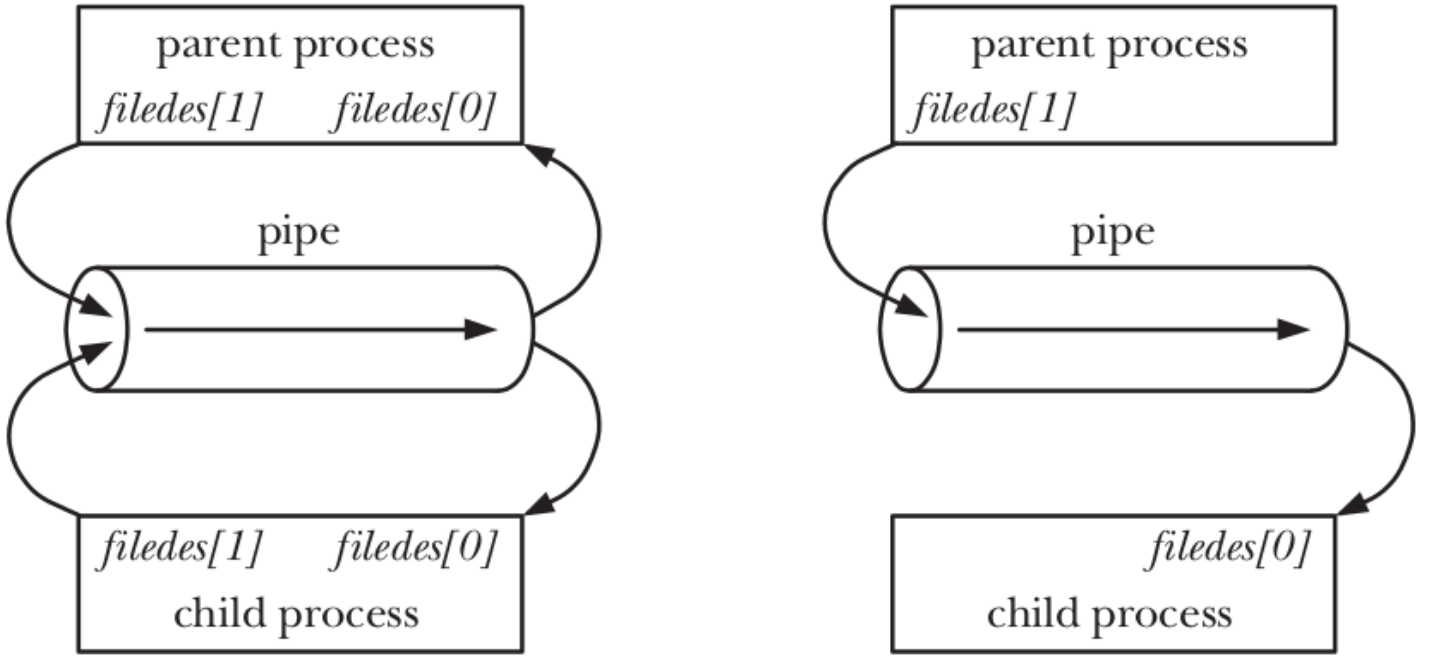
Example:

```
int pipe_fds[2];
int read_fd;
int write_fd;
pipe (pipe_fds);        // no error control ?! -40
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

Data written to the file descriptor `read_fd` can be read back from `write_fd`.

Interprocess Communication

A process's file descriptors cannot be passed to unrelated processes; however, when the process calls `fork`, file descriptors are copied to the new child process. Thus, *pipes can connect only related processes*.



a) After *fork()*

b) After closing unused descriptors

Figure 44-3: Setting up a pipe to transfer data from a parent to a child

Interprocess Communication

Listing 44-1: Steps in creating a pipe to transfer data from a parent to a child

```
int filedes[2];

if (pipe(filedes) == -1)                    /* Create the pipe */
    errExit("pipe");

switch (fork()) {                           /* Create a child process */
case -1:
    errExit("fork");

case 0:  /* Child */
    if (close(filedes[1]) == -1)            /* Close unused write end */
        errExit("close");

    /* Child now reads from pipe */
    break;

default: /* Parent */
    if (close(filedes[0]) == -1)            /* Close unused read end */
        errExit("close");

    /* Parent now writes to pipe */
    break;
}
```

Interprocess Communication

Example (part 1/2)

Listing 5.7 (*pipe.c*) Using a Pipe to Communicate with a Child Process

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

/* Write COUNT copies of MESSAGE to STREAM, pausing for a second
   between each.  */

void writer (const char* message, int count, FILE* stream)
{
  for (; count > 0; --count) {
    /* Write the message to the stream, and send it off immediately.  */
    fprintf (stream, "%s\n", message);
    fflush (stream);
    /* Snooze a while.  */
    sleep (1);
  }
}

/* Read random strings from the stream as long as possible.  */

void reader (FILE* stream)
{
  char buffer[1024];
  /* Read until we hit the end of the stream.  fgets reads until
     either a newline or the end-of-file.  */
  while (!feof (stream)
         && !ferror (stream)
         && fgets (buffer, sizeof (buffer), stream) != NULL)
    fputs (buffer, stdout);
}
```

Interprocess Communication

Example (part 2/2)

```
int main ()
{
  int fds[2];
  pid_t pid;

  /* Create a pipe.  File descriptors for the two ends of the pipe are
     placed in fds.  */
  pipe (fds);
  /* Fork a child process.  */
  pid = fork ();
  if (pid == (pid_t) 0) {
    FILE* stream;
    /* This is the child process.  Close our copy of the write end of
       the file descriptor.  */
    close (fds[1]);
    /* Convert the read file descriptor to a FILE object, and read
       from it.  */
    stream = fdopen (fds[0], "r");
    reader (stream);
    close (fds[0]);
  }
  else {
    /* This is the parent process.  */
    FILE* stream;
    /* Close our copy of the read end of the file descriptor.  */
    close (fds[0]);
    /* Convert the write file descriptor to a FILE object, and write
       to it.  */
    stream = fdopen (fds[1], "w");
    writer ("Hello, world.", 5, stream);
    close (fds[1]);
  }

  return 0;
}
```

Interprocess Communication

When a pipe is created, the file descriptors used for the two ends of the pipe are the next lowest-numbered descriptors available. Since, in normal circumstances, descriptors 0, 1, and 2 are already in use for a process, some higher-numbered descriptors will be allocated for the pipe.

So how do we bring about the situation where two programs that read from *stdin* and write to *stdout* are connected using a pipe, such that the standard output of one program is directed into the pipe and the standard input of the other is taken from the pipe?

By duplicating file descriptors!

Interprocess Communication

```
int pfd[2];
pipe(pfd);
/* Allocates (say) file descriptors 3 and 4 for pipe */
/* Other steps here, e.g., fork() */
close(STDOUT_FILENO); /* Free file descriptor 1 */
dup(pfd[1]); /* dup uses lowest free file descriptor, i.e., fd 1 */
```

The end result of the above steps is that the process's standard output is bound to the write end of the pipe. A corresponding set of calls can be used to bind a process's standard input to the read end of the pipe.

Interprocess Communication

However `dup` assumes that 0, 1 and 2 are already open.

If however 0 was closed for some reason, `pfd[1]` would become its clone instead of 1.

That's why it's a better idea to use `dup2()` instead of `close()` and `dup()`

```
/* Close fd 1, and reopen it as the write end of pipe */
dup2(pfd[1], STDOUT_FILENO);
```

Interprocess Communication

After duplicating `pfd[1]`, we now have two file descriptors referring to the write end of the pipe: descriptor 1 and `pfd[1]`. Since unused pipe file descriptors should be closed, after the `dup2()` call, we close the superfluous descriptor:

```
if (pfd[1] != STDOUT_FILENO) { // just in case
        dup2(pfd[1], STDOUT_FILENO);
        close(pfd[1]);
}
```

Interprocess Communication

Example (part 1/2)

Listing 44-4: Using a pipe to connect *ls* and *wc*

pipes/pipe_ls_wc.c

```
#include <sys/wait.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int pfd[2];                                 /* Pipe file descriptors */

    if (pipe(pfd) == -1)                        /* Create pipe */
        errExit("pipe");

    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:             /* First child: exec 'ls' to write to pipe */
        if (close(pfd[0]) == -1)                /* Read end is unused */
            errExit("close 1");

        /* Duplicate stdout on write end of pipe; close duplicated descriptor */

        if (pfd[1] != STDOUT_FILENO) {          /* Defensive check */
            if (dup2(pfd[1], STDOUT_FILENO) == -1)
                errExit("dup2 1");
            if (close(pfd[1]) == -1)
                errExit("close 2");
        }

        execlp("ls", "ls", (char *) NULL);      /* Writes to pipe */
        errExit("execlp ls");

    default:            /* Parent falls through */
        break;
    }
```

Interprocess Communication

Example (part 2/2)

```
    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:             /* Second child: exec 'wc' to read from pipe */
        if (close(pfd[1]) == -1)                    /* Write end is unused */
            errExit("close 3");

        /* Duplicate stdin on read end of pipe; close duplicated descriptor */

        if (pfd[0] != STDIN_FILENO) {               /* Defensive check */
            if (dup2(pfd[0], STDIN_FILENO) == -1)
                errExit("dup2 2");
            if (close(pfd[0]) == -1)
                errExit("close 4");
        }

        execlp("wc", "wc", "-l", (char *) NULL);    /* Reads from pipe */
        errExit("execlp wc");

    default:            /* Parent falls through */
        break;
    }

    /* Parent closes unused file descriptors for pipe, and waits for children */

    if (close(pfd[0]) == -1)
        errExit("close 5");
    if (close(pfd[1]) == -1)
        errExit("close 6");
    if (wait(NULL) == -1)
        errExit("wait 1");
    if (wait(NULL) == -1)
        errExit("wait 2");

    exit(EXIT_SUCCESS);
}
```

pipes/pipe_ls_wc.c

Interprocess Communication

The inherent synchronization mechanism of pipes can be exploited for general purpose synchronization as well.

Imagine a parent process that wants to wait/block until all of its children have accomplished their respective tasks (a.k.a. *synchronization barrier*); could you solve this through signals?

Interprocess Communication

It's solvable through pipes as well: the parent builds a pipe before creating the child processes.

Each child inherits a file descriptor for the write end of the pipe and closes this descriptor once it has completed its action.

After all of the children have closed their file descriptors for the write end of the pipe, the parent's `read()` from the pipe will complete, returning end-of-file (0). At this point, the parent is free to carry on to do other work.

(Note that closing the unused write end of the pipe in the parent is essential for the correct operation of this technique; otherwise, the parent would block forever when trying to read from the pipe.)

Interprocess Communication

Example (part 1/3)

The following is an example of what we see when we use the program in Listing 44-3 to create three children that sleep for 4, 2, and 6 seconds:

```
$ ./pipe_sync 4 2 6
08:22:16  Parent started
08:22:18  Child 2 (PID=2445) closing pipe
08:22:20  Child 1 (PID=2444) closing pipe
08:22:22  Child 3 (PID=2446) closing pipe
08:22:22  Parent ready to go
```

Listing 44-3: Using a pipe to synchronize multiple processes

pipes/pipe_sync.c

```
#include "curr_time.h"                  /* Declaration of currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int pfd[2];                         /* Process synchronization pipe */
    int j, dummy;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sleep-time...\n", argv[0]);

    setbuf(stdout, NULL);               /* Make stdout unbuffered, since we
                                           terminate child with _exit() */
    printf("%s  Parent started\n", currTime("%T"));

    if (pipe(pfd) == -1)
        errExit("pipe");
```

Example (part 2/3)

```
for (j = 1; j < argc; j++) {
    switch (fork()) {
    case -1:
        errExit("fork %d", j);

    case 0: /* Child */
        if (close(pfd[0]) == -1)        /* Read end is unused */
            errExit("close");

        /* Child does some work, and lets parent know it's done */

        sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
                                        /* Simulate processing */
        printf("%s  Child %d (PID=%ld) closing pipe\n",
                currTime("%T"), j, (long) getpid());
        if (close(pfd[1]) == -1)
            errExit("close");

        /* Child now carries on to do other things... */

        _exit(EXIT_SUCCESS);

    default: /* Parent loops to create next child */
        break;
    }
}
```

Interprocess Communication

Example (part 3/3)

```
    /* Parent comes here; close write end of pipe so we can see EOF */

    if (close(pfd[1]) == -1)                /* Write end is unused */
        errExit("close");

    /* Parent may do other work, then synchronizes with children */

    if (read(pfd[0], &dummy, 1) != 0)
        fatal("parent didn't get EOF");
    printf("%s  Parent ready to go\n", currTime("%T"));

    /* Parent can now carry on to do other things... */

    exit(EXIT_SUCCESS);
}
```

Interprocess Communication

A common use for pipes is to execute a shell command and either read its output or send it some input. The `popen()` and `pclose()` functions are provided to simplify this task.

```
#include <stdio.h>
FILE *popen(const char * command , const char * mode);
```

Returns file stream, or NULL on error

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

Returns termination status of child process, or –1 on error

Interprocess Communication

The `popen()` function creates a pipe, and then forks a child process that execs a shell, which in turn creates a child process to execute the string given in command.

The mode argument is a string that determines whether the calling process will read from the pipe (mode is `r`) or write to it (mode is `w`).

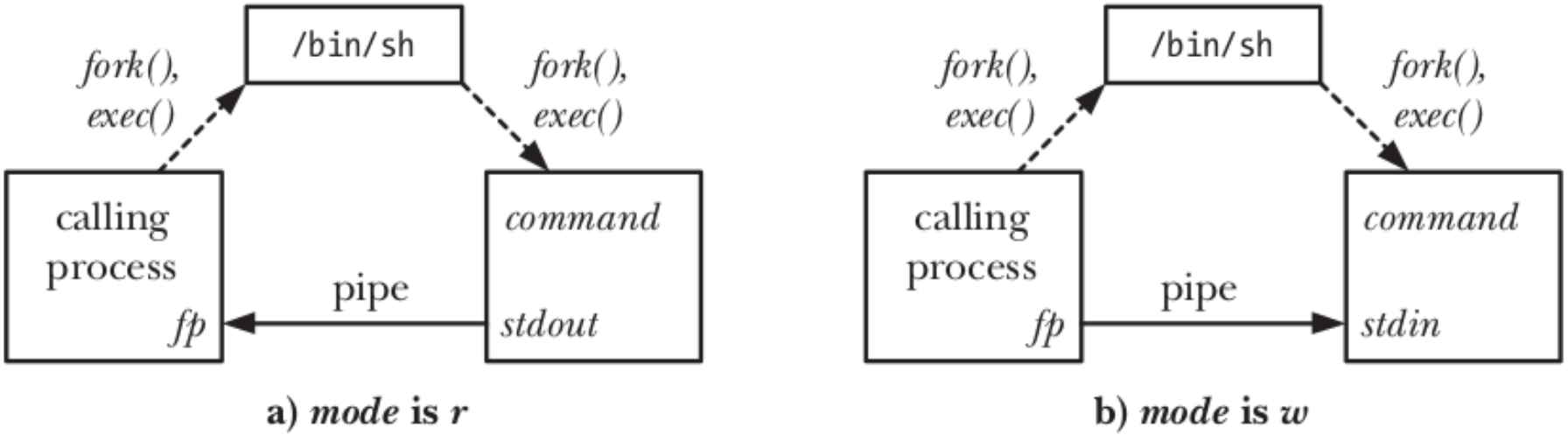


Figure 44-4: Overview of process relationships and pipe usage for *popen()*

Interprocess Communication

Listing 5.9 **(*popen.c*) Example Using *popen***

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

int main ()
{
  FILE* stream = popen ("sort", "w");
  fprintf (stream, "This is a test.\n");
  fprintf (stream, "Hello, world.\n");
  fprintf (stream, "My dog has fleas.\n");
  fprintf (stream, "This program is great.\n");
  fprintf (stream, "One fish, two fish.\n");
  return pclose (stream);
}
```

Eliminates the need for `dup2`, `pipe`, `fork`, `exec`, `fdopen`.

Security-wise, `popen()` is risky (like `system()`) as it involves executing a shell; you can find various exploit examples online for both, usually involving executing commands with a high-privilege user account.

Interprocess Communication

Our second IPC tool is **FIFO**.

A FIFO is simply a pipe that has a name in the filesystem.

Any process can open or close the FIFO; **the processes on either end of the pipe need not be related to each other**. FIFOs are also called *named pipes*.

Just as with pipes, a FIFO has a write end and a read end, and data is read from the pipe in the same order as it is written. This fact gives FIFOs their name: *first-in, first-out*.

Interprocess Communication

We can create a FIFO from the shell using the `mkfifo` command. Specify the path to the FIFO on the command line. For example, create a FIFO in /tmp/fifo by invoking this:

```
$ mkfifo /tmp/fifo
$ ls -l /tmp/fifo
prw-rw-rw-    1 exprof  users  0 Jan 18 14:04   /tmp/fifo
```

When applied to a FIFO (or pipe), `fstat()` and `stat()` return a file type of `S_IFIFO` in the `st_mode` field of the stat structure. When listed with `ls -l`, a FIFO is shown with the type p in the first column.

Interprocess Communication

In one window, read from the FIFO by invoking the following:

```
$ cat < /tmp/fifo
```

In a second window, write to the FIFO by invoking this:

```
$ cat > /tmp/fifo
```

Then type in some lines of text. Each time you press Enter, the line of text is sent through the FIFO and appears in the first window. Close the FIFO by pressing Ctrl+D in the second window.

Remove the FIFO with this line:

```
$ rm /tmp/fifo
```

Interprocess Communication

The `mkfifo()` call creates a new FIFO with the given pathname.

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

Returns 0 on success, or –1 on error

The `mode` argument specifies the permissions for the new FIFO. These permissions are specified by OR'ing the desired combination of constants: S_I(R|W|X)(USR|GRP|OTH)

Interprocess Communication

Access a FIFO just like an ordinary file. To communicate through a FIFO, one process must open it for writing, and another process must open it for reading. Either low-level I/O functions (`open`, `write`, `read`, `close`) or C library I/O functions (`fopen`, `fprintf`, `fscanf`, `fclose`) may be used.

For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:

```
int fd = open (fifo_path, O_WRONLY);
write (fd, data, data_length);
close (fd);
```

Interprocess Communication

To read a string from the FIFO using C library I/O functions, you could use this code:

```
FILE* fifo = fopen (fifo_path, “r”);
fscanf (fifo, “%s”, buffer);
fclose (fifo);
```

A FIFO can have multiple readers or multiple writers. Bytes from each writer are written atomically up to a maximum size of PIPE_BUF (4KB on Linux). Chunks from simultaneous writers can be interleaved. Similar rules apply to simultaneous reads.

The FIFO must be opened on both ends before data can be passed. Normally, opening the FIFO blocks until the other end is opened also (use the O_NONBLOCK flag in `open()` if you don’t want this).

Interprocess Communication

The parent reads what its child has written to a named pipe

Example (part 1/3)

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/wait.h>
#define BUFSIZE 256
#define FIFO_PERM  (S_IRUSR | S_IWUSR)
/* function predefinitions */
int dofifochild(const char *fifoname, const char *idstring);
int dofifoparent(const char *fifoname);

int main (int argc, char *argv[]) {
   pid_t childpid;

   if (argc != 2) {                          /* command line has pipe name */
      fprintf(stderr, "Usage: %s pipename\n", argv[0]);
      return 1;
   }
   if (mkfifo(argv[1], FIFO_PERM) == -1) {          /* create a named pipe */
      if (errno != EEXIST) {
         fprintf(stderr, "[%ld]:failed to create named pipe %s: %s\n",
              (long)getpid(), argv[1], strerror(errno));
         return 1;
      }
   }
   if ((childpid = fork()) == -1){
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0)                                  /* The child writes */
      return dofifochild(argv[1], "this was written by the child");
   else
      return dofifoparent(argv[1]);
}
```

Interprocess Communication

The child writes to the pipe and returns

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include "restart.h"
#define BUFSIZE 256

int dofifochild(const char *fifoname, const char *idstring) {
   char buf[BUFSIZE];
   int fd;
   int rval;
   ssize_t strsize;

   fprintf(stderr, "[%ld]:(child) about to open FIFO %s...\n",
          (long)getpid(), fifoname);
   while (((fd = open(fifoname, O_WRONLY)) == -1) && (errno == EINTR)) ;
   if (fd == -1) {
      fprintf(stderr, "[%ld]:failed to open named pipe %s for write: %s\n",
             (long)getpid(), fifoname, strerror(errno));
      return 1;
   }
   rval = snprintf(buf, BUFSIZE, "[%ld]:%s\n", (long)getpid(), idstring);
   if (rval < 0) {
      fprintf(stderr, "[%ld]:failed to make the string:\n", (long)getpid());
      return 1;
   }
   strsize = strlen(buf) + 1;
   fprintf(stderr, "[%ld]:about to write...\n", (long)getpid());
   rval = r_write(fd, buf, strsize);
   if (rval != strsize) {
      fprintf(stderr, "[%ld]:failed to write to pipe: %s\n",
             (long)getpid(), strerror(errno));
      return 1;
   }
   fprintf(stderr, "[%ld]:finishing...\n", (long)getpid());
   return 0;
}
```

`r_write` is the repeated form of `write`, in case of `EINTR`

Available at http://usp.cs.utsa.edu/usp/programs/chapter06/

Example (part 2/3)

Interprocess Communication

The parent reads what was written to a named pipe.

Example (part 3/3)

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include "restart.h"
#define BUFSIZE 256
#define FIFO_MODES O_RDONLY

int dofifoparent(const char *fifoname) {
   char buf[BUFSIZE];
   int fd;
   int rval;

   fprintf(stderr, "[%ld]:(parent) about to open FIFO %s...\n",
                      (long)getpid(), fifoname);
   while (((fd = open(fifoname, FIFO_MODES)) == -1) && (errno == EINTR))  ;
   if (fd == -1) {
      fprintf(stderr, "[%ld]:failed to open named pipe %s for read: %s\n",
             (long)getpid(), fifoname, strerror(errno));
      return 1;
   }
   fprintf(stderr, "[%ld]:about to read...\n", (long)getpid());
   rval = r_read(fd, buf, BUFSIZE);
   if (rval == -1) {
      fprintf(stderr, "[%ld]:failed to read from pipe: %s\n",
             (long)getpid(), strerror(errno));
      return 1;
   }
   fprintf(stderr, "[%ld]:read %.*s\n", (long)getpid(), rval, buf);
   return 0;
}
```

Pipes, fifos and the client-server model

The **client–server model** is a distributed application structure that partitions tasks between the providers of a resource or service, called **servers**, and service requesters, called **clients**.

e.g. http server and web browser

This is opposed to the **peer-to-peer** (p2p) model, where two or more programs/computers (or simply *peers*) pool their resources and communicate in a decentralized system (each peer can act as both client and server).

e.g. bittorrent based file sharing

Pipes, fifos and the client-server model

Client-server models have many flavors depending on the number of servers, number of clients, type of communication between them (simple request, request-reply), way of handling the clients (iterative, concurrent), etc.

Consider the common scenario of a single server with multiple clients using fifo based communication, where the server provides the (trivial) service of assigning unique sequential numbers to each client that requests them.

The first impulse is usually to create a fifo between the server and the clients...this is bad idea; why?

Pipes, fifos and the client-server model

The first impulse is usually to create a fifo between the server and the clients...this is bad idea; why?

Because multiple clients would race to read from the FIFO, and possibly read each other's response messages rather than their own.

Therefore, each client creates a unique FIFO that the server uses for delivering the response for that client, and the server needs to know how to find each client's FIFO; either with a pre-agreed name template or by sending the fifo name as part of the request message.

Pipes, fifos and the client-server model

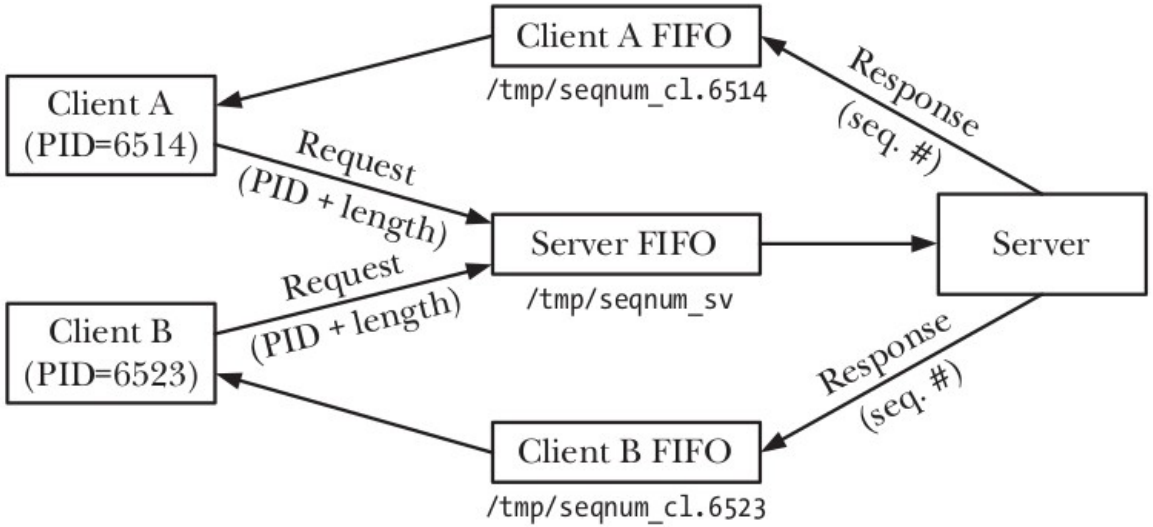


Figure 44-6: Using FIFOs in a single-server, multiple-client application

For instance the names of the client fifos can consist of the pids of the clients, thus easily ensuring uniqueness.

Pipes, fifos and the client-server model

Another practical issue to consider that **the data in pipes and FIFOs is a byte stream**; boundaries between multiple messages are not preserved. This means that when multiple messages are being delivered to a single process, such as the server in our example, then the sender and receiver must agree on some convention for separating the messages; e.g.

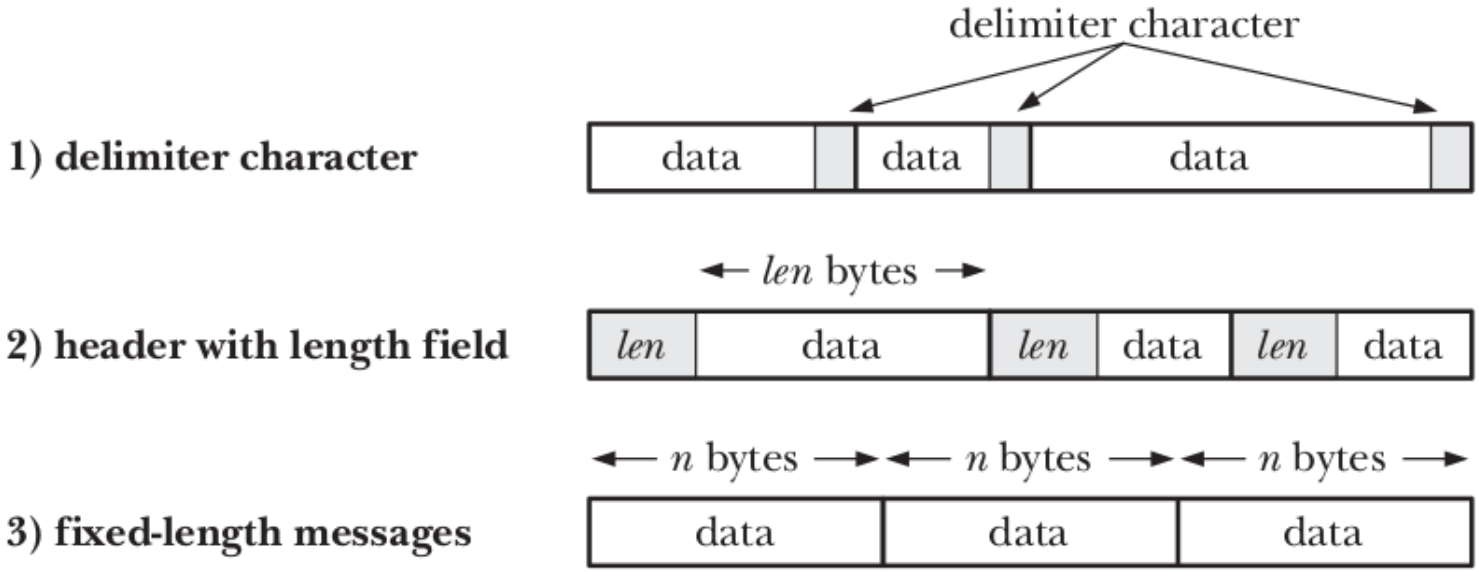


Figure 44-7: Separating messages in a byte stream

Pipes, fifos and the client-server model

Listing 44-6: Header file for `fifo_seqnum_server.c` and `fifo_seqnum_client.c`

`pipes/fifo_seqnum.h`

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#define SERVER_FIFO "/tmp/seqnum_sv"
                                /* Well-known name for server's FIFO */
#define CLIENT_FIFO_TEMPLATE "/tmp/seqnum_cl.%ld"
                                /* Template for building client FIFO name */
#define CLIENT_FIFO_NAME_LEN (sizeof(CLIENT_FIFO_TEMPLATE) + 20)
                                /* Space required for client FIFO pathname
                                   (+20 as a generous allowance for the PID) */

struct request {                /* Request (client --> server) */
    pid_t pid;                  /* PID of client */
    int seqLen;                 /* Length of desired sequence */
};

struct response {               /* Response (server --> client) */
    int seqNum;                 /* Start of sequence */
};
```

Pipes, fifos and the client-server model

```
#include <signal.h>
#include "fifo_seqnum.h"

int
main(int argc, char *argv[])
{
    int serverFd, dummyFd, clientFd;
    char clientFifo[CLIENT_FIFO_NAME_LEN];
    struct request req;
    struct response resp;
    int seqNum = 0;                     /* This is our "service" */

    /* Create well-known FIFO, and open it for reading */

    umask(0);                           /* So we get the permissions we want */
    if (mkfifo(SERVER_FIFO, S_IRUSR | S_IWUSR | S_IWGRP) == -1
            && errno != EEXIST)
        errExit("mkfifo %s", SERVER_FIFO);
    serverFd = open(SERVER_FIFO, O_RDONLY);
    if (serverFd == -1)
        errExit("open %s", SERVER_FIFO);

    /* Open an extra write descriptor, so that we never see EOF */

    dummyFd = open(SERVER_FIFO, O_WRONLY);
    if (dummyFd == -1)
        errExit("open %s", SERVER_FIFO);

    if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
        errExit("signal");
```

Iterative server (part 1/2)

- creates the fifo
- must be executed before clients
- open will block until a client opens as well
- second open ensures that the server doesn't see EOF if all clients close the write end of the FIFO
- SIGPIPE: occurs when writing without a reader; SIGPIPE kills by default; ignore it so as to receive EPIPE instead

Pipes, fifos and the client-server model

Iterative server (part 2/2)

```
    for (;;) {                          /* Read requests and send responses */
        if (read(serverFd, &req, sizeof(struct request))
                != sizeof(struct request)) {
            fprintf(stderr, "Error reading request; discarding\n");
            continue;                   /* Either partial read or error */
        }

        /* Open client FIFO (previously created by client) */

        snprintf(clientFifo, CLIENT_FIFO_NAME_LEN, CLIENT_FIFO_TEMPLATE,
                (long) req.pid);
        clientFd = open(clientFifo, O_WRONLY);
        if (clientFd == -1) {           /* Open failed, give up on client */
            errMsg("open %s", clientFifo);
            continue;
        }

        /* Send response and close FIFO */

        resp.seqNum = seqNum;
        if (write(clientFd, &resp, sizeof(struct response))
                != sizeof(struct response))
            fprintf(stderr, "Error writing to FIFO %s\n", clientFifo);
        if (close(clientFd) == -1)
            errMsg("close");

        seqNum += req.seqLen;           /* Update our sequence number */
    }
}
```

Pipes, fifos and the client-server model

Client (part 1/2)

```
#include "fifo_seqnum.h"

static char clientFifo[CLIENT_FIFO_NAME_LEN];

static void             /* Invoked on exit to delete client FIFO */
removeFifo(void)
{
    unlink(clientFifo);
}

int
main(int argc, char *argv[])
{
    int serverFd, clientFd;
    struct request req;
    struct response resp;
    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [seq-len...]\n", argv[0]);

    /* Create our FIFO (before sending request, to avoid a race) */

    umask(0);                   /* So we get the permissions we want */
    snprintf(clientFifo, CLIENT_FIFO_NAME_LEN, CLIENT_FIFO_TEMPLATE,
            (long) getpid());
    if (mkfifo(clientFifo, S_IRUSR | S_IWUSR | S_IWGRP) == -1
                && errno != EEXIST)
        errExit("mkfifo %s", clientFifo);
    if (atexit(removeFifo) != 0)
        errExit("atexit");
```

```
$ ./fifo_seqnum_server &
[1] 5066
$ ./fifo_seqnum_client 3
0
$ ./fifo_seqnum_client 2
3
$ ./fifo_seqnum_client
5
```

Pipes, fifos and the client-server model

Client (part 2/2)

```
    /* Construct request message, open server FIFO, and send request */

    req.pid = getpid();
    req.seqLen = (argc > 1) ? getInt(argv[1], GN_GT_0, "seq-len") : 1;

    serverFd = open(SERVER_FIFO, O_WRONLY);
    if (serverFd == -1)
        errExit("open %s", SERVER_FIFO);

    if (write(serverFd, &req, sizeof(struct request)) !=
            sizeof(struct request))
        fatal("Can't write to server");

    /* Open our FIFO, read and display response */

    clientFd = open(clientFifo, O_RDONLY);
    if (clientFd == -1)
        errExit("open %s", clientFifo);

    if (read(clientFd, &resp, sizeof(struct response))
            != sizeof(struct response))
        fatal("Can't read response from server");

    printf("%d\n", resp.seqNum);
    exit(EXIT_SUCCESS);
}
```