Pipes

- Overview
- Half-Duplex Pipes
- A Pipe within a Single Process
- A Pipe with 2 Processes
- Closing Pipe Ends
- Sending Data from Child to Parent
- Sending Data from Child to Parent Continued
- Pipe to Connect Filters
- Pipe to Connect Filters Continued
- Pipe to Connect Filters Continued
- Named Pipes or FIFOs
- FIFO Example
- Opening a FIFO
- Using FIFOs for Non-Linear Communication
- FIFOs for Client-Server Communication
- FIFOs for Client-Server Communication Continued
- `popen()`
- `popen()` Output Filter
- A `popen()` Implementation
- A `popen()` Implementation Continued
- A `pclose()` Implementation
- Coprocesses
- Coprocess Implementation
- Coprocess Implementation Continued
- References

Overview

- Cover material for next project.
- Calls for using anonymous pipes for IPC.
- Half-duplex named pipes (FIFOs).
- `popen()`.
- Co-processes.

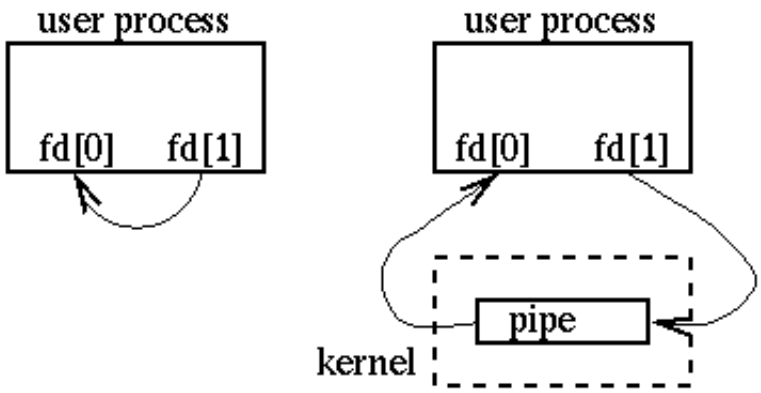
Half-Duplex Pipes

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

- Returns two file descriptors in `filedes[]`. `filedes[0]` is open for reading; `filedes[1]` is open for writing.
- **Half-Duplex**: Data flows in only one direction.
- Can be used only within processes having a common ancestor. Typically, a parent creates a pipe, forks a child and then uses the pipe to communicate with the child.

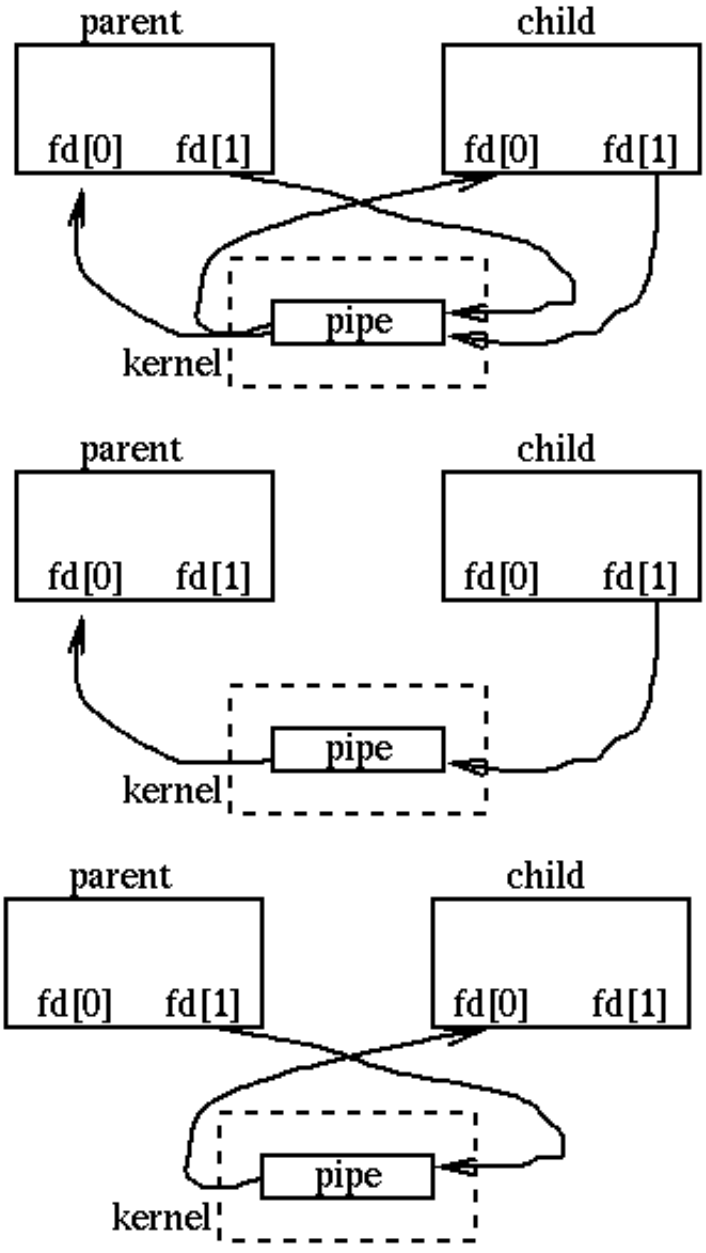
A Pipe within a Single Process

- Not of much use.
- Two ways of looking at it. 2nd picture emphasizes that pipes traverse the kernel.



- `fstat()` on file descriptor has a file type for which `S_ISFIFO` macro returns true.

A Pipe with 2 Processes



Closing Pipe Ends

- If we read from a pipe whose write end has been closed by all processes, then `read()` returns 0 after all the data has been read.
- If we write to a pipe whose read end has been closed by all processes, then the `SIGPIPE` signal is generated.
- A write to a pipe of fewer than `PIPE_BUF` bytes is guaranteed to be contiguous; i.e., it will not be interspersed with data from writes from other writers.

Sending Data from Child to Parent

```
#define MSG "Hello, world\n"

int
main(void)
{
  int pid;
  int fd[2];
  if (pipe(fd) < 0) {
    perror("pipe open"); exit(1);
  }
  if ((pid = fork()) < 0) {
    perror("fork"); exit(1);
  }
  else if (pid == 0) { /* child */
    close(fd[0]);
    write(fd[1], MSG, strlen(MSG));
  }
```

Sending Data from Child to Parent Continued

```
  else { /* parent */
    enum { LEN = 40 };
    int line[LEN];
    int n;
    close(fd[1]);
    n = read(fd[0], line, LEN);
    write(1, line, n);
  }
  return 0;
}
```

Pipe to Connect Filters

```
$ ls | wc
     52      52    1072
$ ./programs/do-pipe ls wc
     52      52    1072
$
```

Program do-pipe.c:

```
int
main(int argc, const char *argv[])
{
  if (argc != 3) {
    fprintf(stderr, "usage: %s PRG1 PRG2\n",
            argv[0]); exit(1);
  }
  doPipe(argv[1], argv[2]);
  return 0;
}
```

Pipe to Connect Filters Continued

```
static void
doPipe(const char *srcProgram, const char *destProgram)
{
  int fd[2];
  if (pipe(fd) < 0) {
    perror("cannot create pipe"); exit(1);
  }
  pid_t pid = fork();
  if (pid < 0) {
    perror("cannot fork"); exit(1);
  }
```

Pipe to Connect Filters Continued

```
  /* child which runs srcProgram with stdout redirected */
  else if (pid == 0) {
    close(fd[0]);
    if (fd[1] != STDOUT_FILENO) {
      if (dup2(fd[1], STDOUT_FILENO) < 0) {
        perror("child dup2"); exit(1);
      }
      close(fd[1]);
    }
    execlp(srcProgram, srcProgram, NULL);
  }
  /* parent which runs destProgram with stdin redirected */
  else {
    close(fd[1]);
    if (fd[0] != STDIN_FILENO) {
      if (dup2(fd[0], STDIN_FILENO) < 0) {
        perror("parent dup2"); exit(1);
      }
      close(fd[0]);
    }
    execlp(destProgram, destProgram, NULL);
  }
}
```

Named Pipes or FIFOs

- Pipes which are in the file system.
- Can be used between unrelated processes.
- Created using system call

```
int mkfifo(const char *pathname, mode_t mode);
```

- Corresponding `mkfifo` command at shell level.
- File I/O API can be used with fifos; `open()`, `read()`, `write()`, `close()`, `unlink()`, as well as `stdio` API.

FIFO Example

```
$ mkfifo fifo
$ perl -ne 'print "***$_";' <fifo &
[4] 6352
$ echo "Hello world" >fifo
$ ***Hello world

[4]-  Done  perl -ne 'print "***$_";' <fifo
$
```

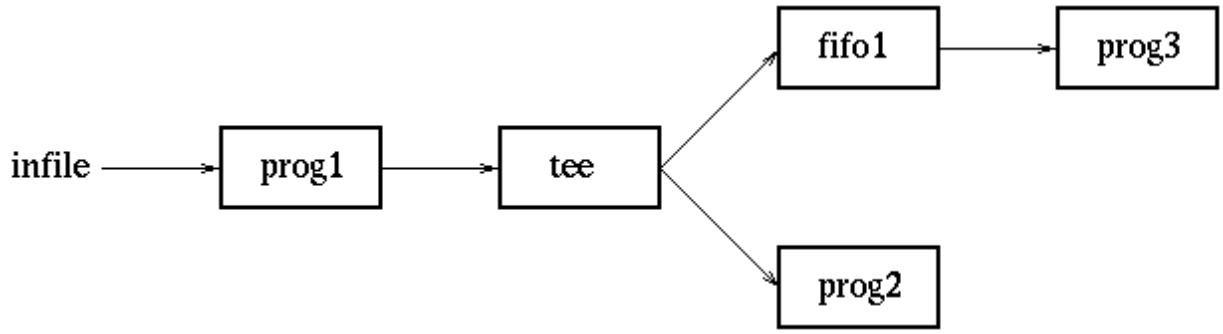
Opening a FIFO

We can `open()` a FIFO like any other file. However, with a FIFO, the `O_NONBLOCK` flag affects what happens:

- Without `O_NONBLOCK`, a `open()` of a FIFO for read-only blocks until some process opens the pipe for writing.
- With `O_NONBLOCK`, a `open()` of a FIFO for read-only returns immediately.
- Without `O_NONBLOCK`, a `open()` of a FIFO for write-only blocks until some process opens the pipe for reading.
- With `O_NONBLOCK`, a `open()` of a FIFO for write-only returns an error (`ENXIO`) if no process has the FIFO open for reading.

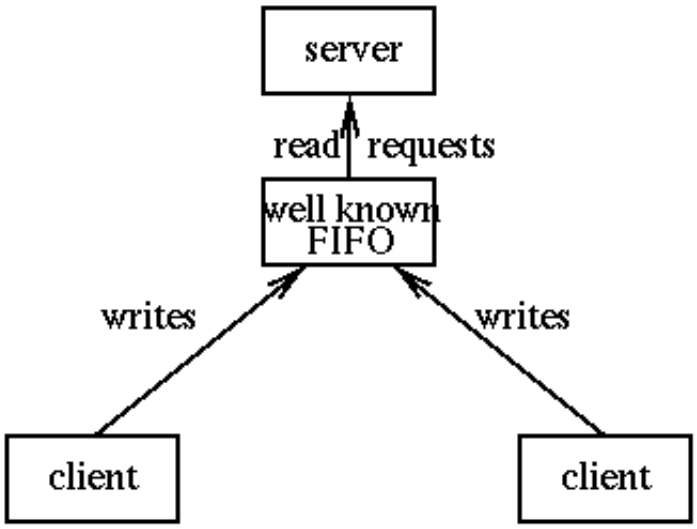
Using FIFOs for Non-Linear Communication

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```



FIFOs for Client-Server Communication

Clients send requests to FIFO over a *well-known* FIFO: i.e. a FIFO whose name is known to all clients.



FIFOs for Client-Server Communication Continued

Server sends responses to clients over client-specific FIFOs. Typically, the name of a client specific FIFO includes client PID. Hence initial server request should include client PID.

If server opens request FIFO only for reading, then each time the number of clients on the request FIFO drops from 1 to 0, then the server will receive an `EOF` on the request FIFO. Common trick is to have server open request FIFO read-write, so that it doesn't need to handle `EOF`.

popen()

- A common situation is where a program needs to `exec()` a child and connect either to its `stdin` or `stdout`.
- `FILE *popen(const char *program, char *mode)` does a `fork()`/`exec()` of `program` and connects a pipe to it.
- The return value of `popen()` is a `FILE` pointer to the parent end of the pipe.
- If the `mode` is `"r"`, then the parent can *read* the pipe.
- If the `mode` is `"w"`, then the parent can *write* the pipe.
- `int pclose(FILE *fp)` is used to close a previously `popen()`'d `FILE` pointer `fp`. Returns terminations status of `program`, or -1 on error.

popen( ) Output Filter

```
$ ./outfil 'tr a-z A-Z'
She sells sea shells on the sea shore.
SHE SELLS SEA SHELLS ON THE SEA SHORE.
$

int
main(int argc, char *argv[])
{
  if (argc != 2) {
    fprintf(stderr, "usage: outfil FILTER\n"); exit(1);
  }
  else {
    const char *filter = argv[1];
    FILE *fpout = popen(filter, "w");
    int c;
    if (!fpout) {
      perror("popen"); exit(1);
    }
    setvbuf(fpout, NULL, _IOLBF, 0);
    while ((c = getchar()) != EOF) fputc(c, fpout);
    pclose(fpout);
  }
  return 0;
}
```

A popen( ) Implementation

```
static pid_t *childpid = NULL; /* ptr to dynamically allocated array */

static int maxfd;       /* from our open_max(), {Prog openmax} */

#define SHELL   "/bin/sh"

FILE *
popen(const char *cmdstring, const char *type)
{
  int i, pfd[2];
  pid_t pid;
  FILE  *fp;

  /* only allow "r" or "w" */
  if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
    errno = EINVAL; /* required by POSIX.2 */
    return(NULL);
  }
  if (childpid == NULL) { /* first time through */
    /* allocate zeroed out array for child pids */
    maxfd = open_max();
    if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
      return(NULL);
  }

  if (pipe(pfd) < 0) return(NULL); /* errno set by pipe() */

  if ( (pid = fork()) < 0)
    return(NULL);       /* errno set by fork() */
  else if (pid == 0) { /* child */
    if (*type == 'r') {
      close(pfd[0]);
      if (pfd[1] != STDOUT_FILENO) {
        dup2(pfd[1], STDOUT_FILENO);
        close(pfd[1]);
      }
    }
```

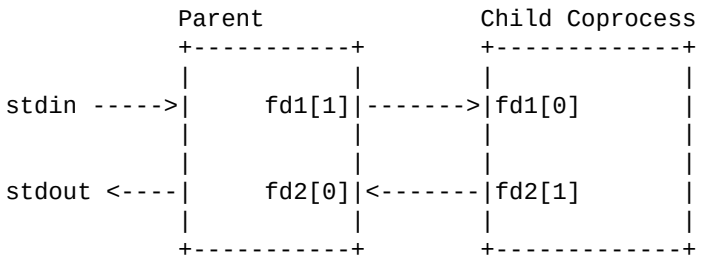
A popen( ) Implementation Continued

```
    else { /* *type == 'w' */
      close(pfd[1]);
      if (pfd[0] != STDIN_FILENO) {
        dup2(pfd[0], STDIN_FILENO);
        close(pfd[0]);
      }
    }
    /* close all descriptors in childpid[] */
    for (i = 0; i < maxfd; i++)
      if (childpid[i] > 0)
        close(i);

    execl(SHELL, "sh", "-c", cmdstring, (char *) 0);
    _exit(127);
  }
  /* parent */
  if (*type == 'r') {
    close(pfd[1]);
    if ( (fp = fdopen(pfd[0], type)) == NULL)
      return(NULL);
  }
  else {
    close(pfd[0]);
    if ( (fp = fdopen(pfd[1], type)) == NULL)
      return(NULL);
  }
  childpid[fileno(fp)] = pid;   /* remember child pid for this fd */
  return(fp);
}
```

A `pclose()` Implementation

```
int
pclose(FILE *fp)
{
  int           fd, stat;
  pid_t pid;

  if (childpid == NULL)
    return(-1);         /* popen() has never been called */

  fd = fileno(fp);
  if ((pid = childpid[fd]) == 0)
    return(-1);         /* fp wasn't opened by popen() */

  childpid[fd] = 0;
  if (fclose(fp) == EOF)
    return(-1);

  while (waitpid(pid, &stat, 0) < 0)
    if (errno != EINTR)
      return(-1);       /* error other than EINTR from waitpid() */

  return(stat); /* return child's termination status */
}
```

Coprocesses



Coprocess Implementation

```
int
main(void)
{
  int n, fd1[2], fd2[2];
  pid_t pid;
  char line[MAXLINE];

  if (pipe(fd1) < 0 || pipe(fd2) < 0)
    err_sys("pipe error");

  if ( (pid = fork()) < 0)
    err_sys("fork error");
  else if (pid > 0) { /* parent */
    close(fd1[0]);
    close(fd2[1]);
    while (fgets(line, MAXLINE, stdin) != NULL) {
      n = strlen(line);
      if (write(fd1[1], line, n) != n)
        err_sys("write error to pipe");
      if ( (n = read(fd2[0], line, MAXLINE)) < 0)
        err_sys("read error from pipe");
      if (n == 0) {
        err_msg("child closed pipe");
        break;
      }
      line[n] = 0;      /* null terminate */
      if (fputs(line, stdout) == EOF)
        err_sys("fputs error");
    }
```

Coprocess Implementation Continued

```
    if (ferror(stdin))
      err_sys("fgets error on stdin");
    exit(0);

  }
  else { /* child */
    close(fd1[1]);
    close(fd2[0]);
    if (fd1[0] != STDIN_FILENO) {
      if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
        err_sys("dup2 error to stdin");
      close(fd1[0]);
    }
    if (fd2[1] != STDOUT_FILENO) {
      if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
        err_sys("dup2 error to stdout");
      close(fd2[1]);
    }
    if (execl("./add2", "add2", (char *) 0) < 0)
      err_sys("execl error");
  }
}
```

References

Text, Ch. 44.