Inter-process communication using pipes

Suntae Hwang

Kookmin University

Introduction

- How processes communicate with each other : Interprocess Communication (IPC)
  - So far we learn that we can do IPC
    - by passing open files across a fork or an exec or
    - through the file system
  - Other techniques
    - pipes (half duplex)
    - FIFOs (named pipes)
    - stream pipes (full duplex), named stream pipes
    - message queues, semaphores, shared memory
    - sockets, streams
  - only some of the techniques are supported.

Pipes

- The oldest form of UNIX IPC and provided by all Unix systems.

- Two limitations
  - Half-duplex : data flows only in one direction
  - Can be used only between processes that have a common ancestor. (Usually a pipe is created by a process, that process calls fork, and the pipe is used between the parent and child.)

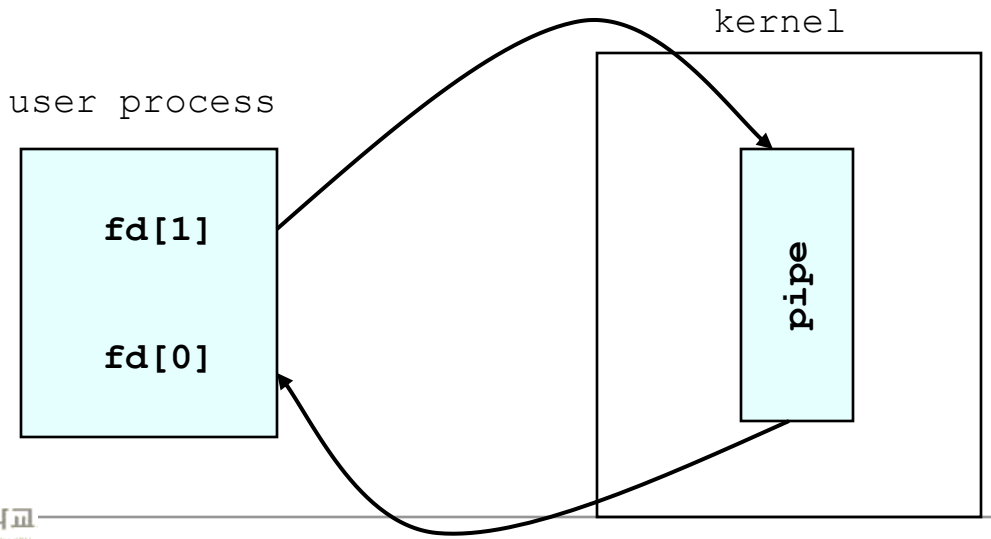
Pipes

```
#include <unistd.h>

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

Returns: 0 if OK, -1 on error

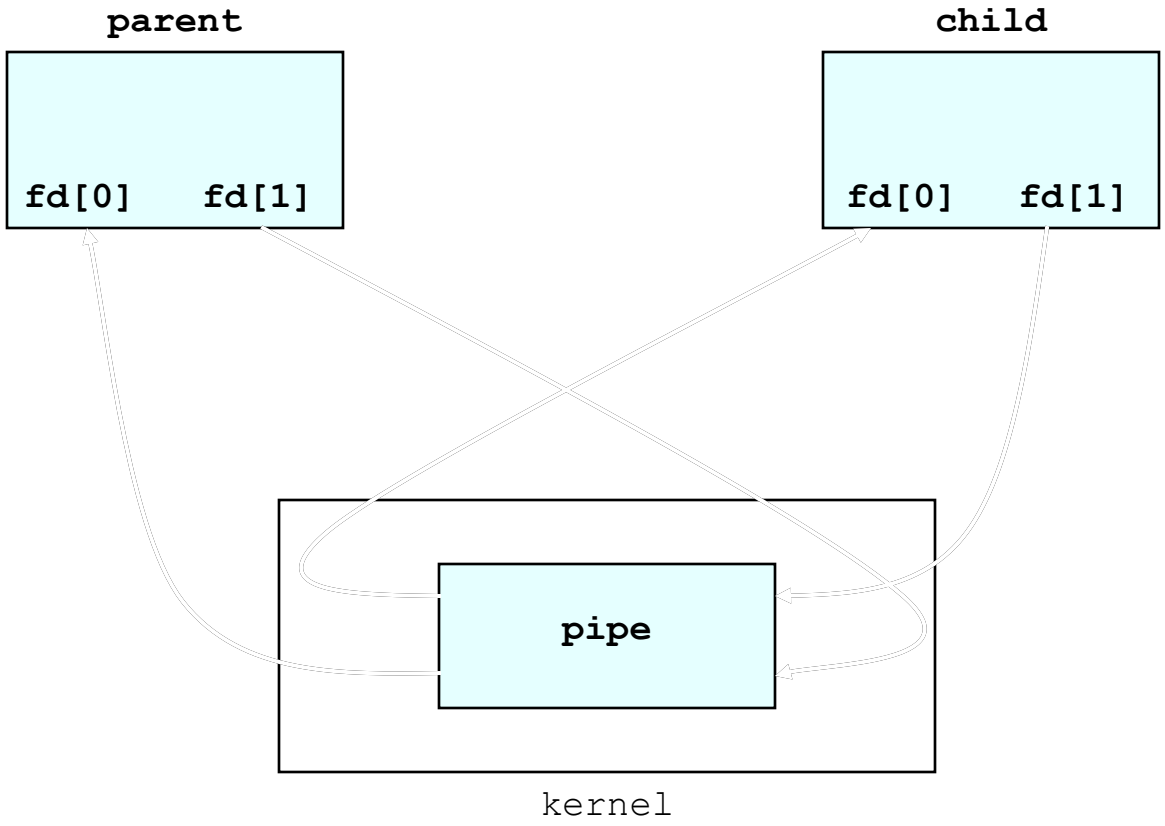
- Two file descriptors are returned through the *filedes* argument.
  - *filedes[0]* : open for reading.
  - *filedes[1]* : open for writing.
  - The output of *filedes[1]* is the input for *filedes[0]*.



Pipes – parent and child processes

```
parent => child:
parent closes fd[0];
child closes  fd[1];
```

```
parent <= child:
parent closes fd[1];
child closes  fd[0];
```



Pipes

- When one end of a pipe is closed,
  - reading from a pipe whose write end has been closed returns an end of file.
  - writing to a pipe whose read end has been close causes SIGPIPE is generated and the write returns an error (EPIPE).
  - A write of PIPE_BUF (kernel's pipe buffer size) bytes or less will not be interleaved with the writes from other processes.
  - fstat function returns a file type of FIFO for the pipe file descriptors. (can be tested by S_ISFIFO macro)

Pipes

```
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);
}
```

```
$ a.out
$ hello world
```

Pipes

```
int main(int argc, char *argv[]){
        int n, fin, fd[2];
        pid_t   pid;  char    line[MAXLINE];
         if (pipe(fd) < 0) err_sys("pipe error");

         fin = open(argv[1], O_RDONLY);

         if ( (pid = fork()) < 0) err_sys("fork error");
         else if (pid > 0) {   /* parent */
                 close(fd[0]);
                 while((n=read(fin,line,MAXLINE)))
                         write(fd[1], line, n);
        }
         else {     /* child */
                 close(fd[1]);

                 close(0);       /* close stdin */
                 dup(fd[0]);     /* duplicate stdin (redirection) */
                 execl("/bin/more", "more", (char*) 0);
        }
        exit(0);
}
```

Pipe Example – pager(1/3)

```
#define  DEF_PAGER        "/usr/bin/more"          /* default pager program */
int
main(int argc, char *argv[])
{
  int             n, fd[2];
  pid_t  pid;
  char   line[MAXLINE], *pager, *argv0;
  FILE   *fp;
  if (argc != 2)
         err_quit("usage: a.out <pathname>");
  if ( (fp = fopen(argv[1], "r")) == NULL)
         err_sys("can't open %s", argv[1]);
  if (pipe(fd) < 0)
         err_sys("pipe error");
  if ( (pid = fork()) < 0)
         err_sys("fork error");
```

Pipe Example – pager(2/3)

```
else if (pid > 0) {
        /* parent */
        close(fd[0]);              /* close read end */
                /* parent copies argv[1] to pipe */
        while (fgets(line, MAXLINE, fp) != NULL) {
                n = strlen(line);
                if (write(fd[1], line, n) != n)
                        err_sys("write error to pipe");
        }
        if (ferror(fp))
                err_sys("fgets error");
        close(fd[1]);    /* close write end of pipe for reader */
        if (waitpid(pid, NULL, 0) < 0)
                err_sys("waitpid error");
        exit(0);
```

Pipe Example – pager(3/3)

```
  } else {
        /* child */
        close(fd[1]);    /* close write end */
        if (fd[0] != STDIN_FILENO) {
              if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                    err_sys("dup2 error to stdin");
              close(fd[0]);    /* don't need this after dup2 */
        }
              /* get arguments for execl() */
        if ( (pager = getenv("PAGER")) == NULL)
              pager = DEF_PAGER;
        if ( (argv0 = strrchr(pager, '/')) != NULL)
              argv0++;         /* step past rightmost slash */
        else
              argv0 = pager;   /* no slash in pager */
        if (execl(pager, argv0, (char *) 0) < 0)
              err_sys("execl error for %s", pager);
  }
}
```

Dup2

NAME

dup2 - duplicate an open file descriptor

SYNOPSIS

```
#include <unistd.h>
int dup2(int fildes, int fildes2);
```

MT-LEVEL

Unsafe

Async-Signal-Safe

DESCRIPTION

dup2() causes the file descriptor fildes2 to refer to the same file as fildes. fildes is a file descriptor referring to an open file, and fildes2 is a non-negative integer less than the current value for the maximum number of open file descriptors allowed the calling process (see getrlimit(2)). If fildes2 already referred to an open file, not fildes, it is closed first. If fildes2 refers to fildes, or if fildes is not a valid open file descriptor, fildes2 will not be closed first.

RETURN VALUES

Upon successful completion a non-negative integer, namely, the file descriptor, is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

`popen` and `pclose` Functions

```
#include <stdio.h>

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

Returns: file pointer if OK, `NULL` on error

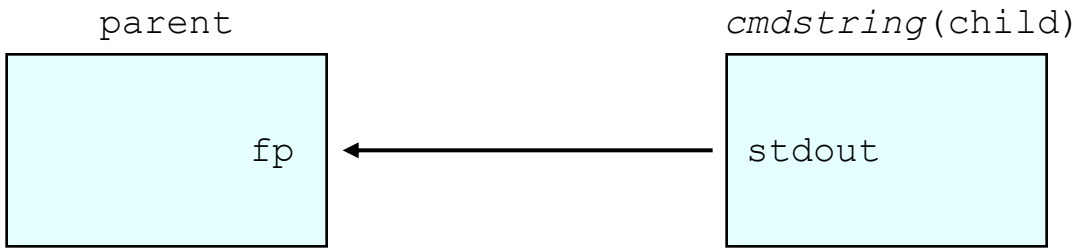
```
int pclose(FILE *fp);
```

Returns: termination status of *cmdstring*, or –1 on error

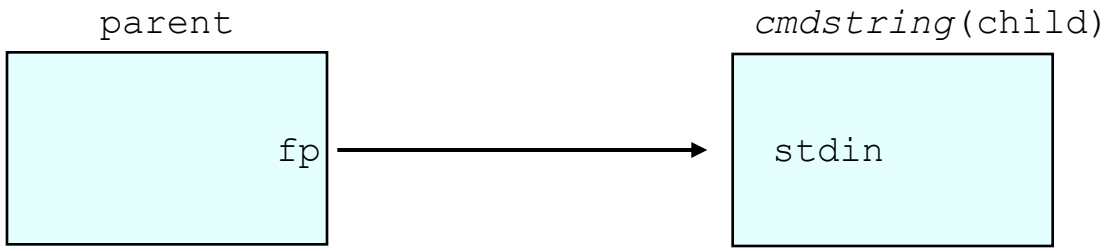
- Standard I/O library provides pipe functions: `popen` and `pclose`
  - handle all the work :
    - the creation of a pipe, the *fork* of a child, closing the unused ends of the pipe, *exec*ing a *shell* to execute the command, and waiting for the command to terminate.
- **`popen`** does a fork and exec to execute the *cmdstring* and returns a file pointer.
  - The file pointer is connected to the standard output(input) of *cmdstring* if type is “r”(“w”)
- **`pclose`** closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell.

popen and pclose Functions

```
fp = popen(command, "r" );
```




```
fp = popen(command, "w" );
```



popen and pclose Functions

```
#include <sys/wait.h>
#include "ourhdr.h"
#define  PAGER   "${PAGER:-more}" /* environment variable, or default */

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

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

        if ( (fpout = popen(PAGER, "w")) == NULL)
                err_sys("popen error");

                /* copy argv[1] to pager */
        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);
}
```

Our version of `popen` and `pclose`

- The next program shows our version of popen and pclose. Although the core of popen is similar to the code we've used so far, there are many details that we need to take care of . First, each time popen is called we have to remember the process ID of the child that we create and either its file descriptor or FILE pointer. We choose to save the child's process ID in the array childpid, which we index by the file descriptor. This way, when pclose is called, with the FILE pointer as its argument, we call the standard I/O function fileno to get the file descriptor, and then have the child process ID for the call to waitpid. Since it's possible for a given process to call popen more than once, we dynamically allocate the childpid array (the first time popen is called), with room for as many children as there are file descriptors.
- What happens if the caller of pclose has established a signal handler for SIGCHLD? Waitpid would return an error of EINTR. Since the caller is allowed to catch this signal (or any other signal that might interrupt the call to waitpid) we just call waitpid again if it is interrupted by a caught signal.

Our version of `popen` and `pclose` (cont'd)

- Popen
  - Create a pipe by `pipe()`
  - Create a process by `fork()`
  - Remember child process PID
  - Reopen the pipe by `fdopen()`

  In the child process
  - Redirect standard IO by `dup2()`
  - Execute a command by `exec()` with "`sh -c command`"

- Pclose
  - Remove the pipe by `close()`
  - Remove the child process by `waitpid()`

Our popen(1/3)

```
static pid_t *childpid = NULL;     /* ptr to array allocated at run-time */
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() */
```

Our popen(2/3)

```
else if (pid == 0) {    /* child */
        if (*type == 'r') {
                close(pfd[0]);
                if (pfd[1] != STDOUT_FILENO) {
                        dup2(pfd[1], STDOUT_FILENO);
                        close(pfd[1]);
                }
        } else {
                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);
}
```

Our popen(3/3)

```
        /* 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);
}
```

Our `pclose`

```
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 */
}
```

Filter

- A UNIX filter is a program that reads from standard input and writes to standard output.
- Filters are normally connected linearly in shell pipelines.

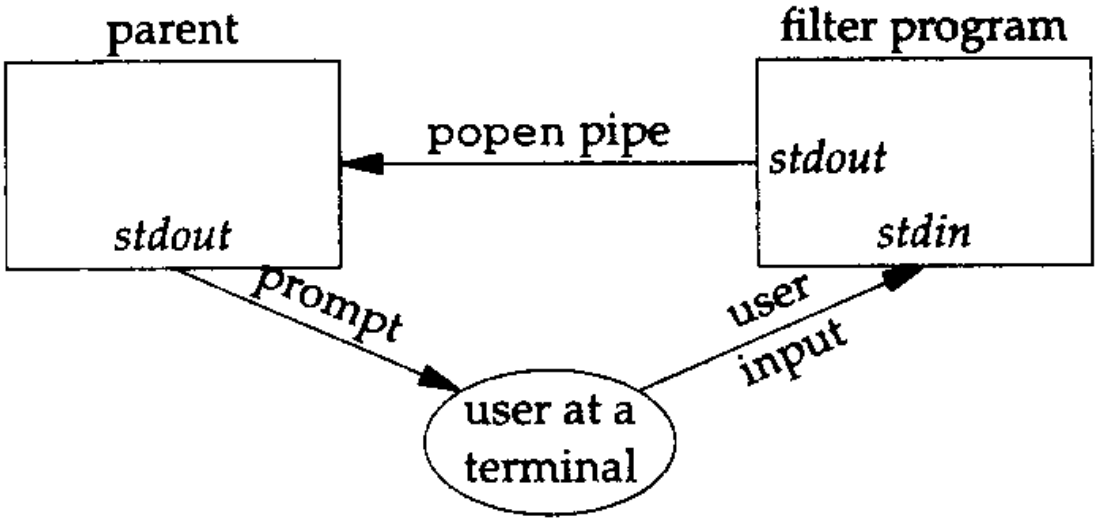


Figure 14.8 Transforming input using popen.

myuclc.c

```
ipc/myuclc.c

#include <ctype.h>
#include "ourhdr.h"
int
main(void)
{
  int             c;
  while ( (c = getchar()) != EOF) {
          if (isupper(c))
                  c = tolower(c);
          if (putchar(c) == EOF)
                  err_sys("output error");
          if (c == '\n')
                  fflush(stdout);
  }
  exit(0);
}
```

popen1.c

```
Int main(void)
{
  char    line[MAXLINE];
  FILE    *fpin;
  if ( (fpin = popen("myuclc", "r")) == NULL)
          err_sys("popen error");
  for ( ; ; ) {
          fputs("prompt> ", stdout);
          fflush(stdout);
          if (fgets(line, MAXLINE, fpin) == NULL)    /* read from pipe */
                  break;
          if (fputs(line, stdout) == EOF)
                  err_sys("fputs error to pipe");
  }
  if (pclose(fpin) == -1)
          err_sys("pclose error");
  putchar('\n');
  exit(0);
}
```

```
$ popen1
prompt> PIPE open
pipe open
prompt>
```

Coprocesses

- A filter becomes a *coprocess* when the same program generates its input and reads its output.

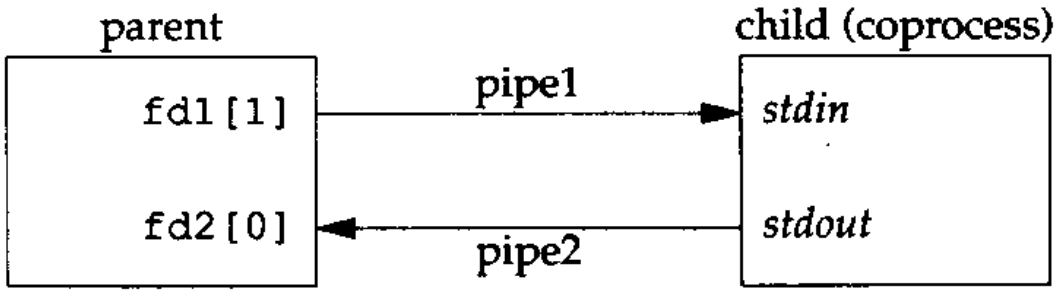


Figure 14.9 Driving a coprocess by writing its standard input and reading its standard output.

add2.c

```
#include "ourhdr.h"
int
main(void)
{
  int    n, int1, int2;
  char   line[MAXLINE];
  while ( (n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
         line[n] = 0;              /* null terminate */
         if (sscanf(line, "%d%d", &int1, &int2) == 2) {
                sprintf(line, "%d\n", int1 + int2);
                n = strlen(line);
                if (write(STDOUT_FILENO, line, n) != n)
                       err_sys("write error");
         } else {
                if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                       err_sys("write error");
         }
  }
  exit(0);
}
```

pipe4.c(1/3)

```
static void sig_pipe(int signo)
{
  printf("SIGPIPE caught\n");
  exit(1);
}

static void      sig_pipe(int);            /* our signal handler */
int
main(void)
{
  int            n, fd1[2], fd2[2];
  pid_t  pid;
  char   line[MAXLINE];
  if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
         err_sys("signal error");
  if (pipe(fd1) < 0 || pipe(fd2) < 0)
         err_sys("pipe error");
  if ( (pid = fork()) < 0)
         err_sys("fork error");
```

pipe4.c(2/3)

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

pipe4.c(3/3)

```
  } 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");
  }
}
```

add2stdio.c

- What happens if standard I/Os are used instead read/write ?

```
int
main(void)
{
  int             int1, int2;
  char    line[MAXLINE];
  while (fgets(line, MAXLINE, stdin) != NULL) {
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
                if (printf("%d\n", int1 + int2) == EOF)
                        err_sys("printf error");
        } else {
                if (printf("invalid args\n") == EOF)
                        err_sys("printf error");
        }
  }
  exit(0);
}
```

Pseudo terminal

- 위의 프로그램을 coprocess로 사용하면 deadlock이 발생한다. 왜?
  - 위의 coprocess의 stdin과 stdout은 pipe로 통해 있으므로 terminal driver가 아니므로 fully buffered가 된다. 즉 위의 add2stdio.c는 standard input에서 읽기 위해 blocking되고, pipe4.c 역시 pipe에서 읽을 때 block된다.
- 이를 해결하기 위해 stdin과 stdout을 강제로 lined buffer로 바꿀 수 있다.

```
setvbuf(stdin, NULL, _IOLBF, 0);
setvbuf(stdout, NULL, _IOLBF, 0);
```

NULL means automatically allocated buffer

- 그러나 coprocess의 source file을 갖고 있지 않는 경우에는 어떻게 하나?
  - The solution for this general problem is to make the coprocess being invoked think that its standard input and output are connected to a terminal. That causes the standard I/O routines in the coprocess to line buffer these two I/O streams, similar to what did with the explicit calls to setvbuf previously. We use pseudo terminal to do this.

FIFOs

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

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

Returns: 0 if OK, -1 on error

- Named pipes
- Unrelated processes can exchange data, whereas pipes can be used only between related processes.
- FIFO is a type of file : FIFO type (S_ISFIFO macro)
- Once a FIFO created, the normal file I/O functions (open,close, read write, unlink etc) all work with FIFO

FIFOs

- Opening a FIFO:
  - Normal cases (w/o O_NONBLOCK)
    - An open for read(write)-only blocks until some other process opens the FIFO for writing(reading).
  - Nonblocking (with O_NONBLOCK)
    - An open for read-only returns immediately if no process has the FIFO open for writing
    - An open for write-only returns an error (errno=ENXIO) if no process has the FIFO open for reading

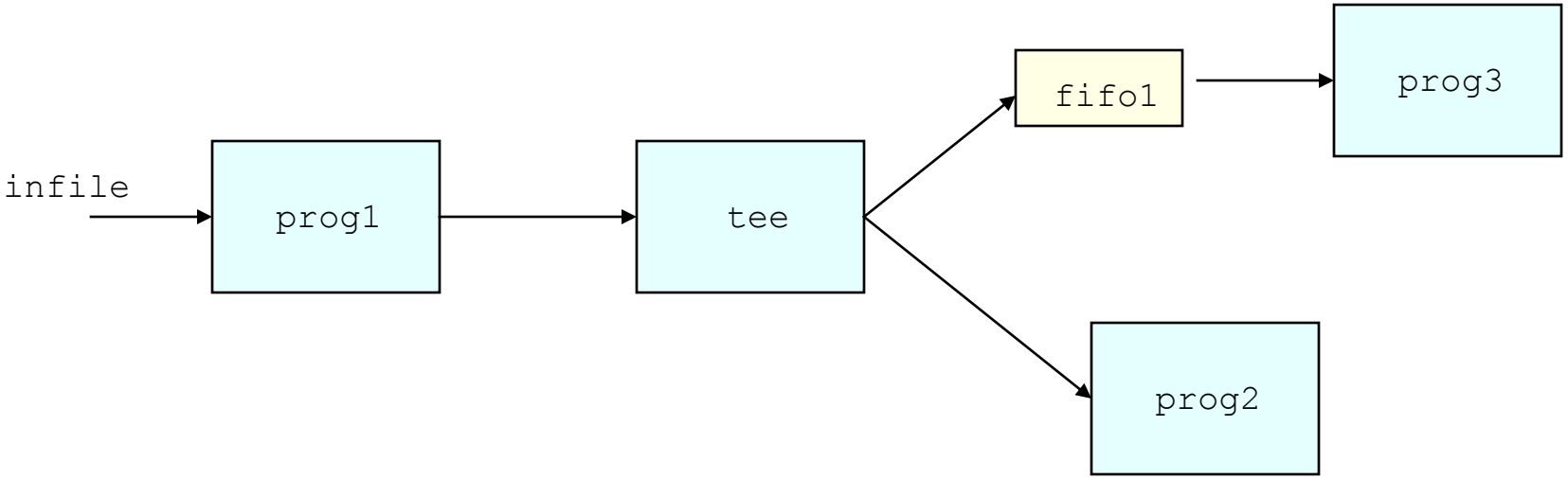
FIFOs

- Read and Writes
  - Writing to a FIFO that no process has open for reading causes SIGPIPE to generate.
  - When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO.
  - PIPE_BUF : the maximum amount of data that can be written atomically to a FIFO (without being interleaved among multiple writers).
- Use of FIFO
  - shell commands to pass data from one shell pipeline to another without creating intermediate temporary files
  - A client-server application to pass data between the clients and server.

FIFOs Examples

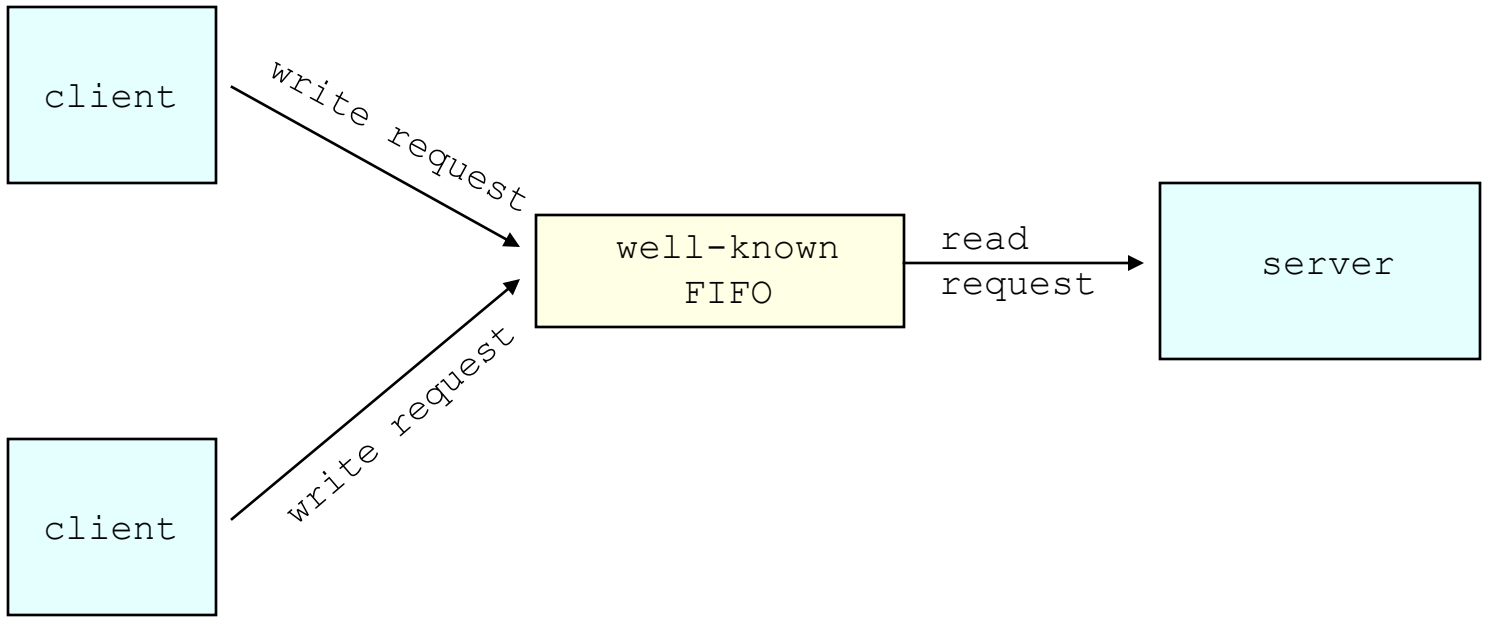
- Using FIFOs to Duplicate Output Stream
  - tee(1) – copies its standard input to both its standard output and to the file named on its command line

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



FIFOs Examples

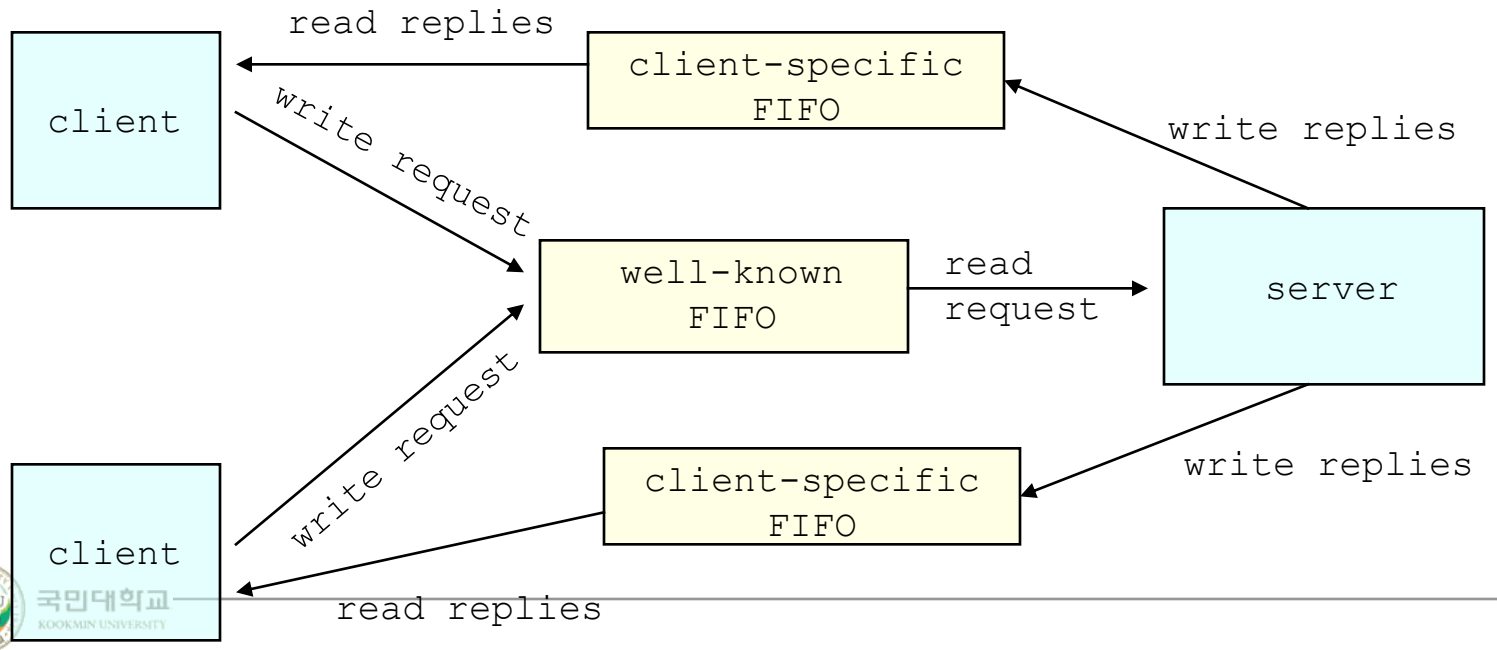
- Client-Server Communication Using a FIFO
  - Clients write to a “well-known” FIFO to send a request to the server (*The write needs to be less than PIPE_BUF bytes in size not to be interleaved*)



FIFOs Examples (cont'd)

- Client-Server Communication Using a FIFO
  - Problem with the previous slide: *no reply path* to clients
  - A single reply FIFO won't be enough because clients would never know when to read their response.
  - A solution: create a reply FIFO for each client.

/tmp/serv1.1234, where 1234 is replaced with the client's process ID



FIFOs Examples (cont'd)

- If a client crashes,
  - the client-specific FIFOs to be left in the file system since the server cannot tell this.
  - The server also must catch SIGPIPE, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.
- If the server open the well-known FIFO read-only,
  - each time the number of clients goes from 1 to 0 the server will read an end of file on the FIFO.
  - To prevent the server having to handle this case, a common trick is just to have the server open the well-known FIFO for read-write.