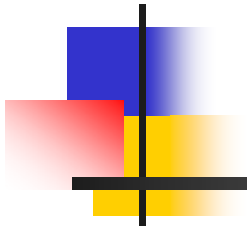
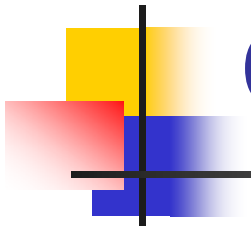


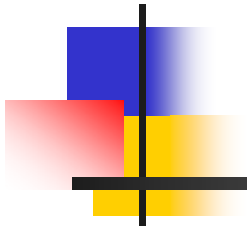
프로세스 간 통신 (Interprocess communication)

숙명여대 창병모

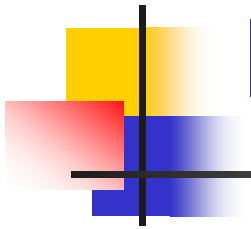


Contents

1. Pipes
2. FIFOs

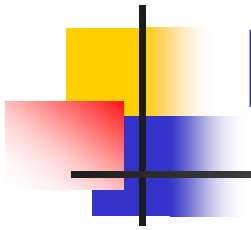


파이프(Pipe)



IPC using Pipes

- IPC using regular files
  - unrelated processes can share
  - fixed size
  - life-time
  - lack of synchronization
- IPC using pipes
  - for transmitting data between related processes
  - can transmit an unlimited amount of data
  - automatic synchronization on open()



Pipes

- In a UNIX shell,
  - the pipe symbol | (the vertical bar)
- In a shell, UNIX pipes look like:

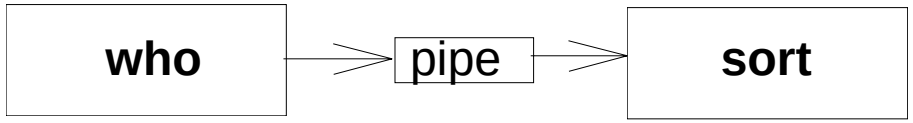
```
$ ls -alg | more
$ command 1 | command 2
```

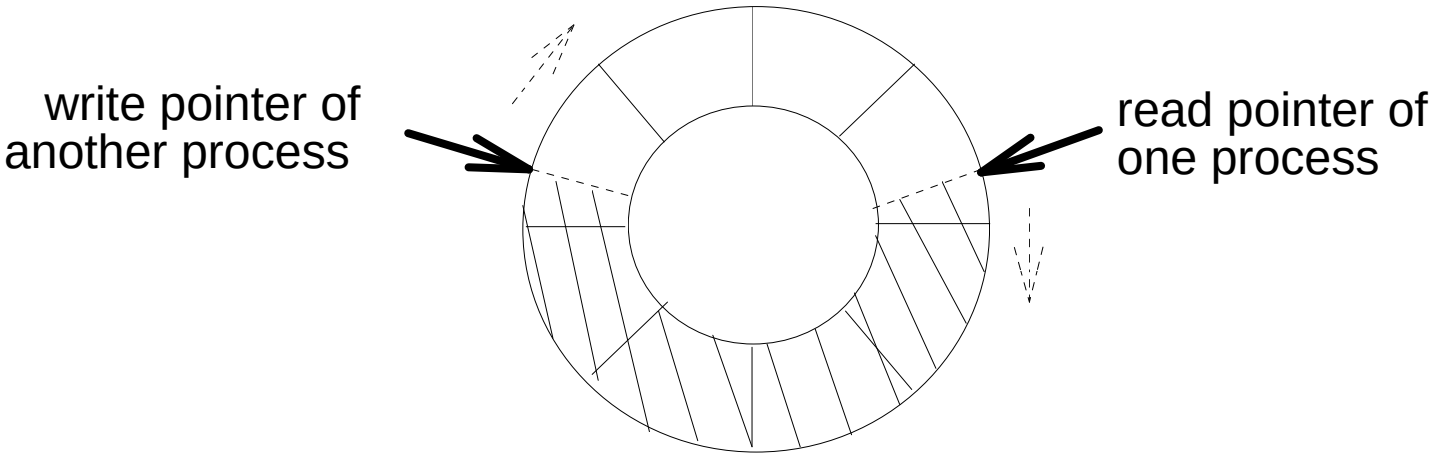
  - the standard output of command 1 becomes the standard input of command2
- We can have longer pipes:

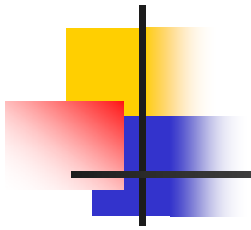
```
$ pic paper.ms | tbl | eqn | ditroff -ms
```

Example

- ***%who | sort***

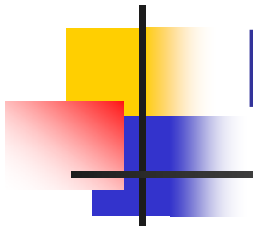






IPC using Pipes

- Data transmitting
  - data is written into pipes using the write( ) system call
  - data is read from a pipe using the read( ) system call
  - automatic blocking when full or empty
- Types of pipes
  - (unnamed) pipes
  - named pipes



Pipes

- In UNIX, pipes are the oldest form of IPC
- Limitations of Pipes
  - Half duplex (data flows in one direction)
  - Can only be used between processes that have a common ancestor

    (Usually used between the parent and child processes)
  - A child process inherits pipes of its parent process

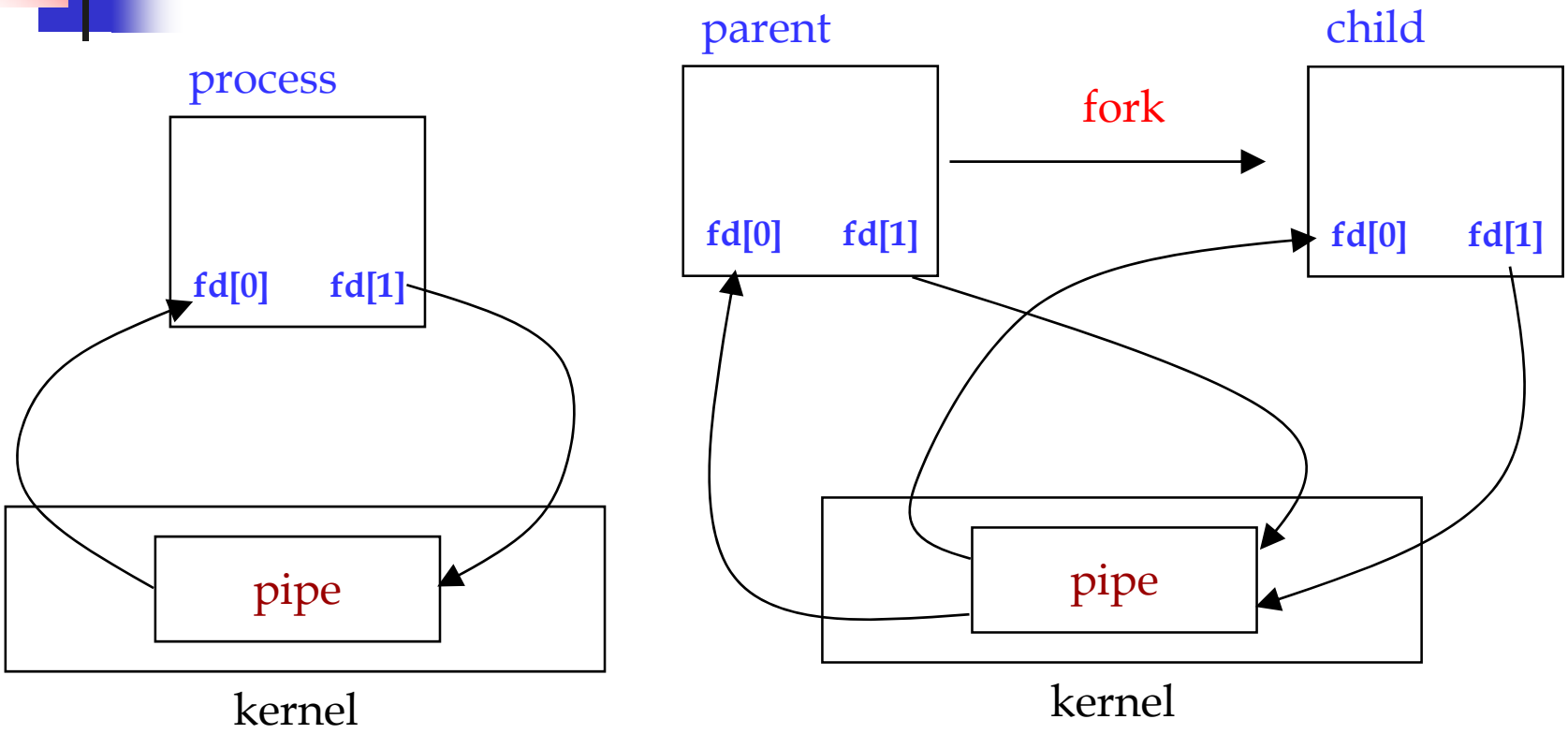
Pipes

```
#include <unistd.h>

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

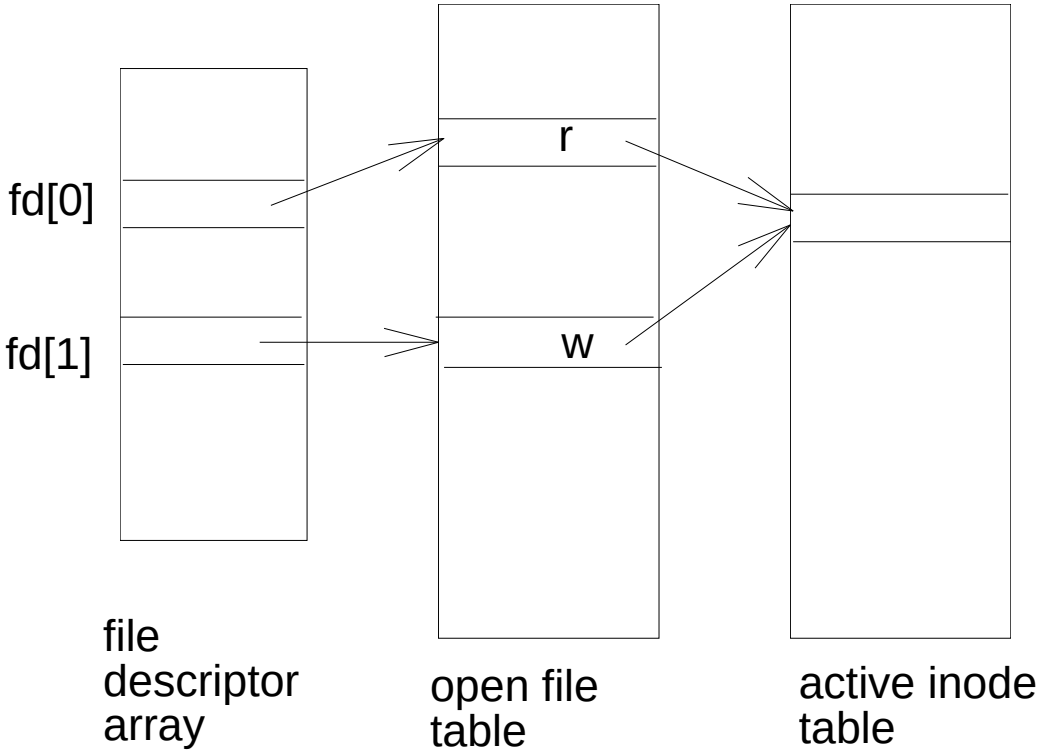
Returns: 0 if OK, -1 on error

- two file descriptors
  - *fd*[0] : read file descriptor for the pipe
  - *fd*[1] : write file descriptor for the pipe
- Anything that is written on *fd*[1] may be read by *fd*[0]
  - This is of no use in a single process.
  - A method of communication between processes.
  - A way to communicate with parent-child processes.

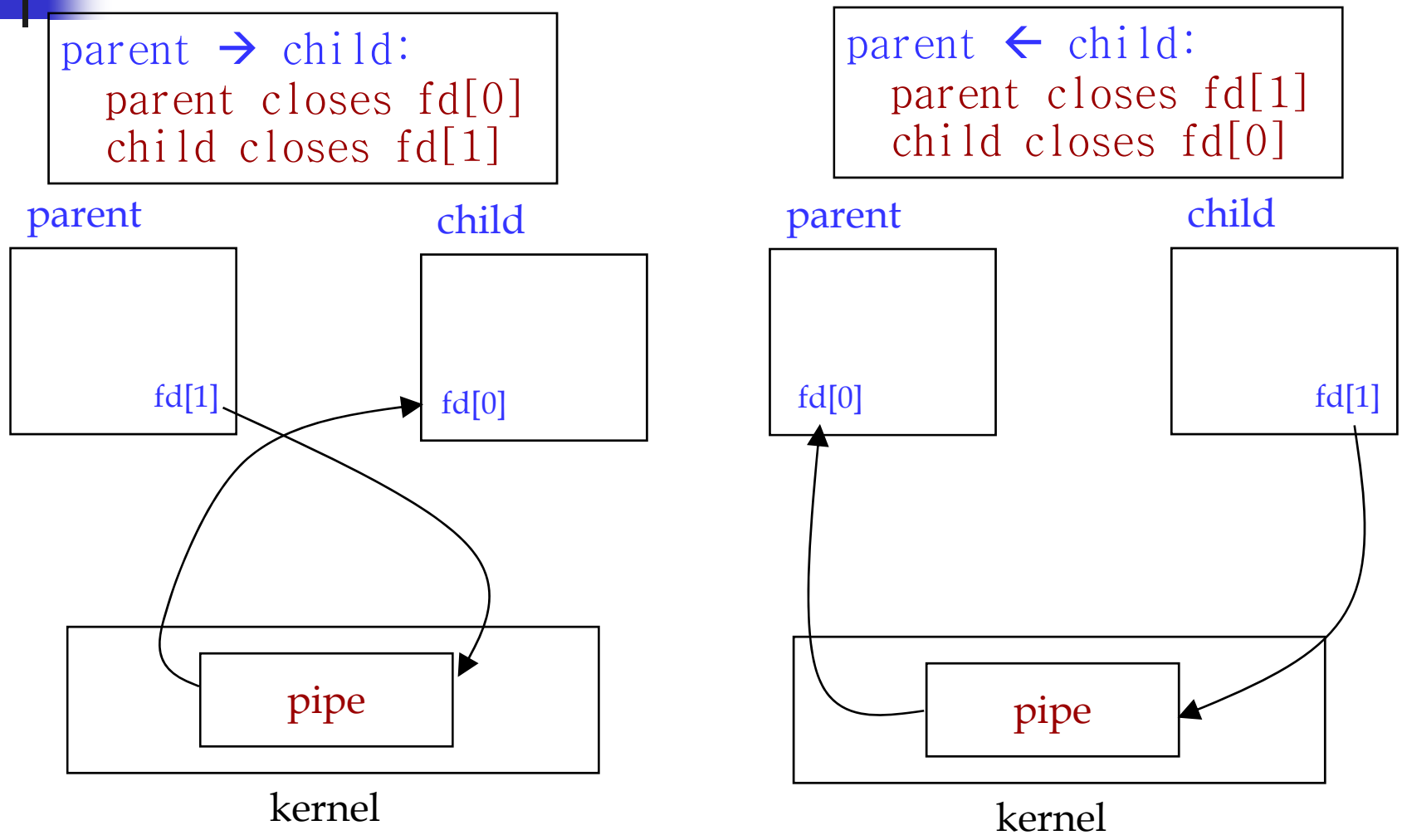
Pipes

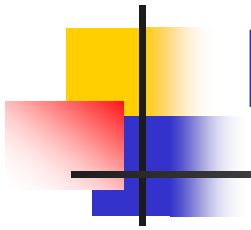

process
fd[0]
fd[1]
pipe
kernel
parent
child
fork
fd[0]
fd[1]
fd[0]
fd[1]
pipe
kernel

How to implement pipes



Pipes


parent → child:
parent closes fd[0]
child closes fd[1]
parent
child
fd[1]
fd[0]
pipe
kernel
parent ← child:
parent closes fd[1]
child closes fd[0]
parent
child
fd[0]
fd[1]
pipe
kernel

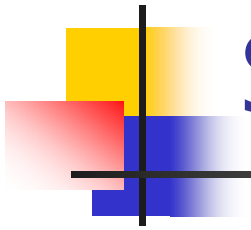


Pipes

- Read from a pipe with write end closed
  - returns 0 to indicate `EOF`
- Write to a pipe with read end closed
  - `SIGPIPE` generated,
  - write() returns error (errno == `EPIPE`)
- Atomic write
  - A write of `PIPE_BUF` (kernel's pipe buffer size) bytes or less will not be interleaved with the writes from other processes

Sending messages from parent to child

```
#include  <unistd.h>  /* pipe1.c */
#define MAXLINE 100

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

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

  if ( (pid = fork()) < 0)
     perror("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);
}
```



Shell Pipe 구현

- % command1 | command2
- IDEA
  - Child executes command1
  - Parent executes command2
  - Standard output of command1(child)
    → via pipe →
  - Standard input of command2(parent)

Example: Shell Pipe 구현

```
#include <stdio.h>
#define READ 0
#define WRITE 1
main(argc, argv )
int argc;  char* argv[];
{
    int fd[2];
    pipe(fd);
    if (fork() ==0) { // child
        close(fd[READ]);
        dup2(fd[WRITE],1);
        close(fd[WRITE]);
        execlp(argv[1], argv[1], NULL);
        perror("Connect");
    }
    else {          // parent
        close(fd[WRITE]);
        dup2(fd[READ],0);
        close(fd[READ]);
        execlp(argv[2], argv[2], NULL);
        perror("Connect");
    }
}
```

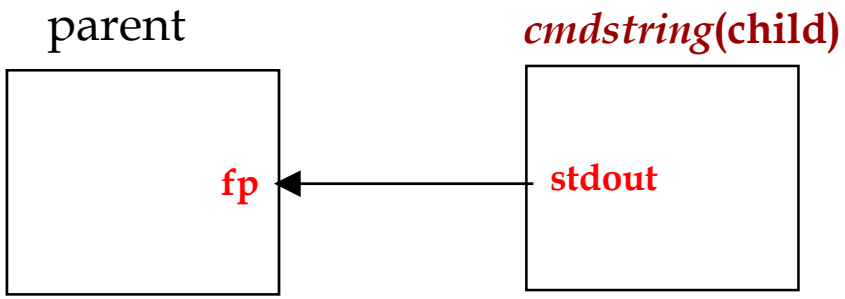
popen()/pclose()

```
#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
```

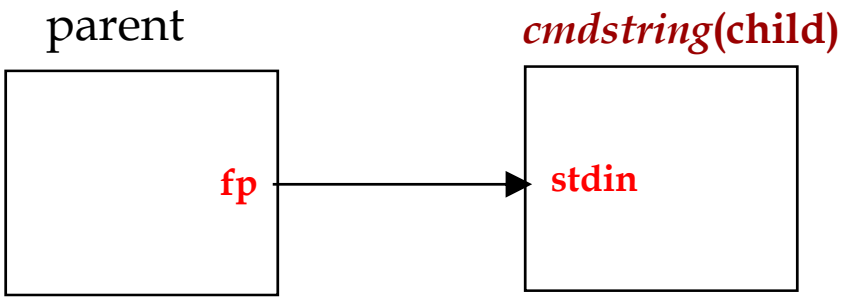
- handle all the dirty work
  - the creation of a pipe,
  - the fork of a child,
  - closing the unused ends of the pipe,
  - execing a shell to execute the command, and
  - waiting for the command to terminate
- popen does a fork and exec to execute the *cmdstring* and returns a file pointer.

popen() pclose()

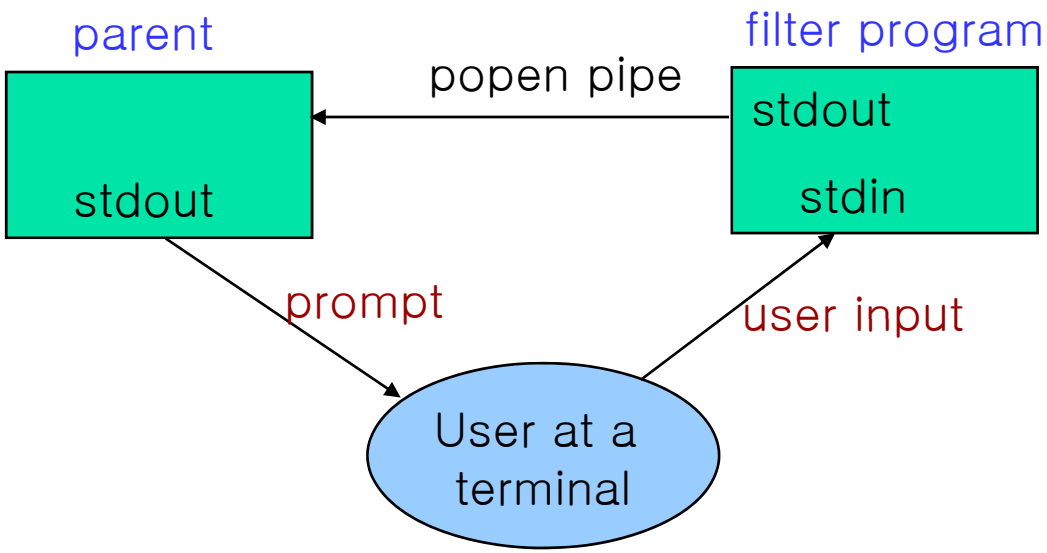
fp = popen(*cmdstring*, "r");



fp = popen(*cmdstring*, "w");

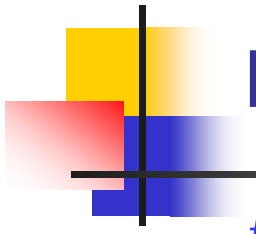


Example: popen1.c



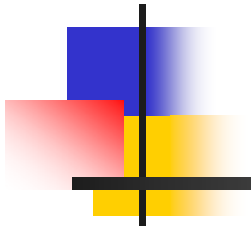
Example: popen1.c

```
#include  <sys/wait.h>  /* popen1.c */

int main(void) {
  char  line[MAXLINE];
  FILE  *fpin;
  if ( (fpin = popen("myuclc", "r")) == NULL)
          perror("popen error");

  for ( ; ; ) {
          fputs("prompt> ", stdout);
          fflush(stdout);
          if (fgets(line, MAXLINE, fpin) == NULL)  /* read from pipe */
                  break;
          if (fputs(line, stdout) == EOF)
                  perror("fputs error to pipe");
  }
  if (pclose(fpin) == -1)
          perror("pclose error");
  putchar('\n');
  exit(0);
}
```

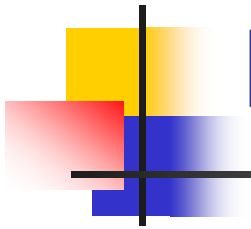


myuclc.c

```
#include <ctype.h>      /* myuclc.c */
#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);
}
```

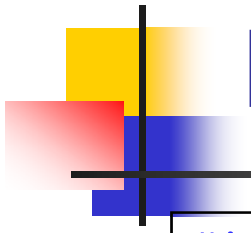


FIFO(Name Pipe)



FIFOs

- Pipes can be used only between related processes
- FIFOs are "named pipes"
  - can be used between unrelated processes
- A type of file
  - stat.st_mode == FIFO
  - Test with `S_ISFIFO` macro

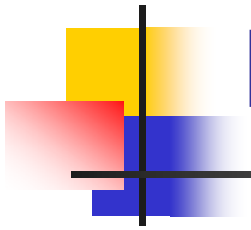


FIFOs

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

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

Returns: 0 if OK, -1 on error

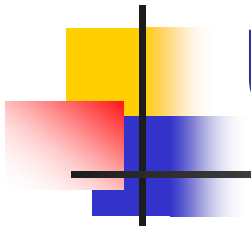
- Creating FIFOs is similar to creating a file
  - *pathname* : filename
  - *mode*: permissons, same as for open( ) function
- Using a FIFO is similar to using a file
  - we can open, `close`, `read`, `write`, `unlink`, etc., to the FIFO



FIFOs

- If FIFO opened without O_NONBLOCK flag
  - an open for read-only blocks
    until some other process opens the FIFO for writing
  - an open for write-only blocks
    until some other process opens the FIFO for reading

- If O_NONBLOCK is specified (nonblocking)
  - 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



Using FIFOs

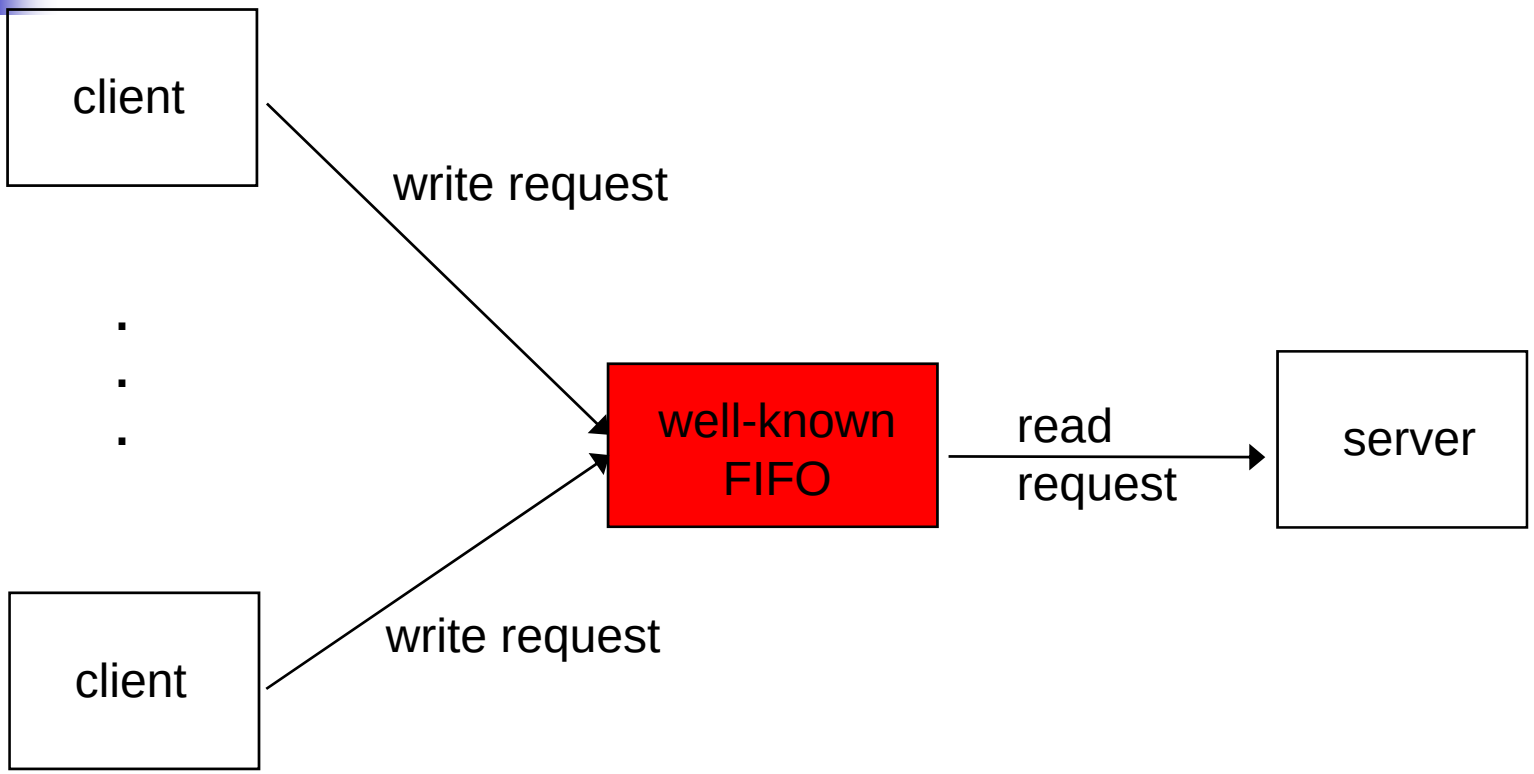
- Shell commands
  - pass data from one shell pipeline to another, without creating intermediate files
- Client-server application
  - pass data between clients and server

Client-Server Comm. Using a FIFO

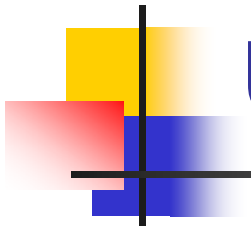
- Server
  - creates a "well-known" FIFO to communicate with clients
- Client
  - writes at most `PIPE_BUF` bytes at a time to avoid interleaving of client data,
- Problem

  Server can't reply clients using a single "well-known" FIFO

Client-Server Communication Using a FIFO

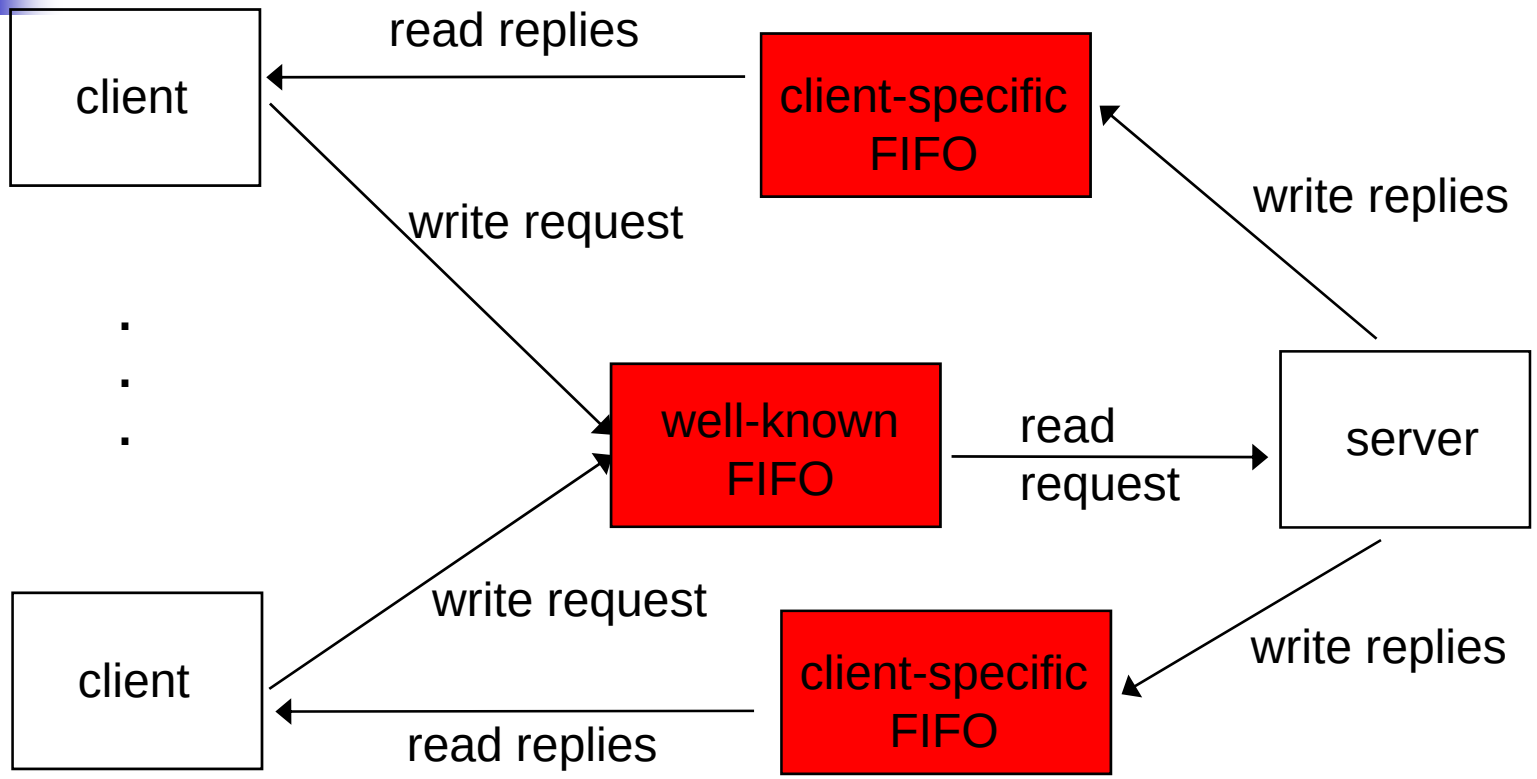


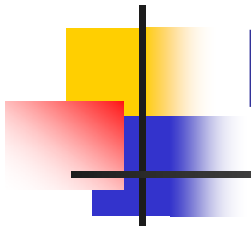
Client-Server Communication Using a FIFO



- Solution
  - Create a FIFO for each client such that server can reply
  - e.g.: /tmp/serv1.XXXX, where XXXXX is client's process ID
- What if a client has crashed ?
  - FIFOs left in system
  - (FIFO with 1 writer, no reader)
- Server must catch SIGPIPE
  - it's possible for a client to send a request and
  - terminate before reading the response,
  - leaving the FIFO with one writer (the server) and no reader

Client-Server Communication Using a FIFO



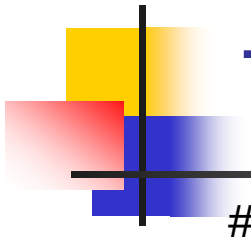


FIFO Log

- 서버(fifo_s) 프로그램은
  - fifo 를 만들고
  - fifo 에서 데이터를 읽어서
  - 파일에 기록한다
- 클라이언트(fifo_c) 프로그램은
  - fifo 에 문자열을 100 개 출력한다

fifo_s.c

```
#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>
#include  "util.h"
#define FIFO_NAME "fifo1"
#define FILE_NAME "fifo1.txt“

int main() {
  char s[100];
  int r, fd1, fd2;

  r = mkfifo(FIFO_NAME, S_IRUSR | S_IWUSR);
  if (r < 0) error("mkfifo error");

  fd1 = open(FIFO_NAME, O_RDONLY);
  if (fd1 < 0) error(FIFO_NAME " open error");

  fd2 = open(FILE_NAME, O_WRONLY | O_CREAT, S_IRUSR |S_IWUSR);
  if (fd2 < 0) error(FILE_NAME " open error");

  while ((r = read(fd1, s, 100)) > 0)
    write(fd2, s, r);
  close(fd1);
  close(fd2);
}
```



fifo_c.c

```
#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>
#include <string.h>
#include  "util.h"

#define FIFO_NAME "fifo1"

int main() {
  char s[100];
  int i, r, fd;

  fd = open(FIFO_NAME, O_WRONLY);
  if (fd < 0) error(FIFO_NAME " open error");

  for (i=0; i < 100; ++i) {
    sprintf(s, "%d %d\n", getpid(), i);
    write(fd, s, strlen(s));
  }
  close(fd);
}
```