UNIX Systems Programming

Interprocess communication

Overview

1. What is a Pipe
2. Unix System Review
3. Processes (review)
4. Pipes
5. FIFOs

1. Pipes

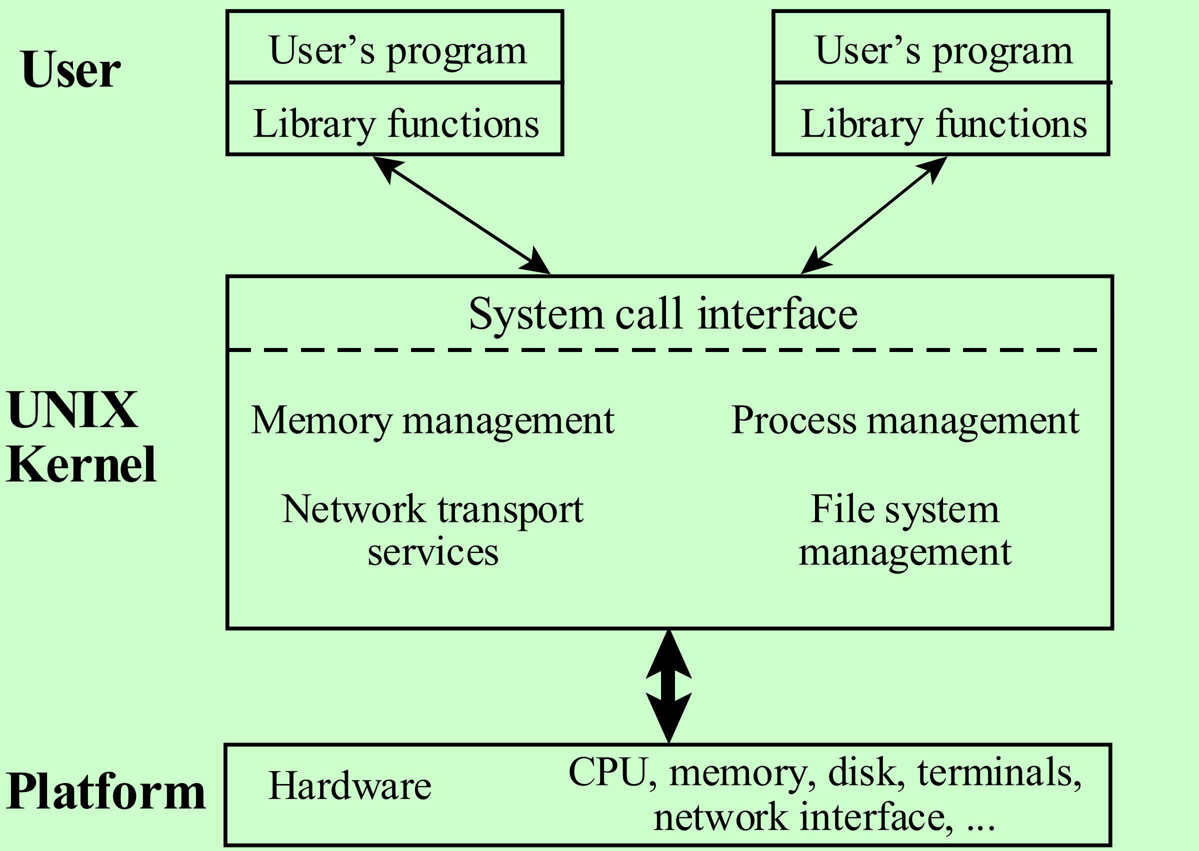
- A form of interprocess communication between processes that have a common ancestor
- It is a one-way (half duplex) communication channel which can be used to link processes
- A pipe is a generalization of the file idea
  - Can use I/O functions like `read()` and `write()` to receive and send data
- Typical use:
  - Pipe created by a process
  - Process calls fork()
  - Pipe used between parent and child

Differences between versions

- All systems support half-duplex
  - Data flows in only one direction
- Many newer systems support full duplex
  - Data flows in two directions
- For portability, assume only half-duplex

- Pipes at the UNIX shell level
  - who | wc -1
  - gives a count of the number of users logged on

A UNIX System



Review: fork()

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork( void );
```

- Creates a child process by making a *copy* of the parent process
- Both the child *and* the parent continue running

Context used by child and exec()

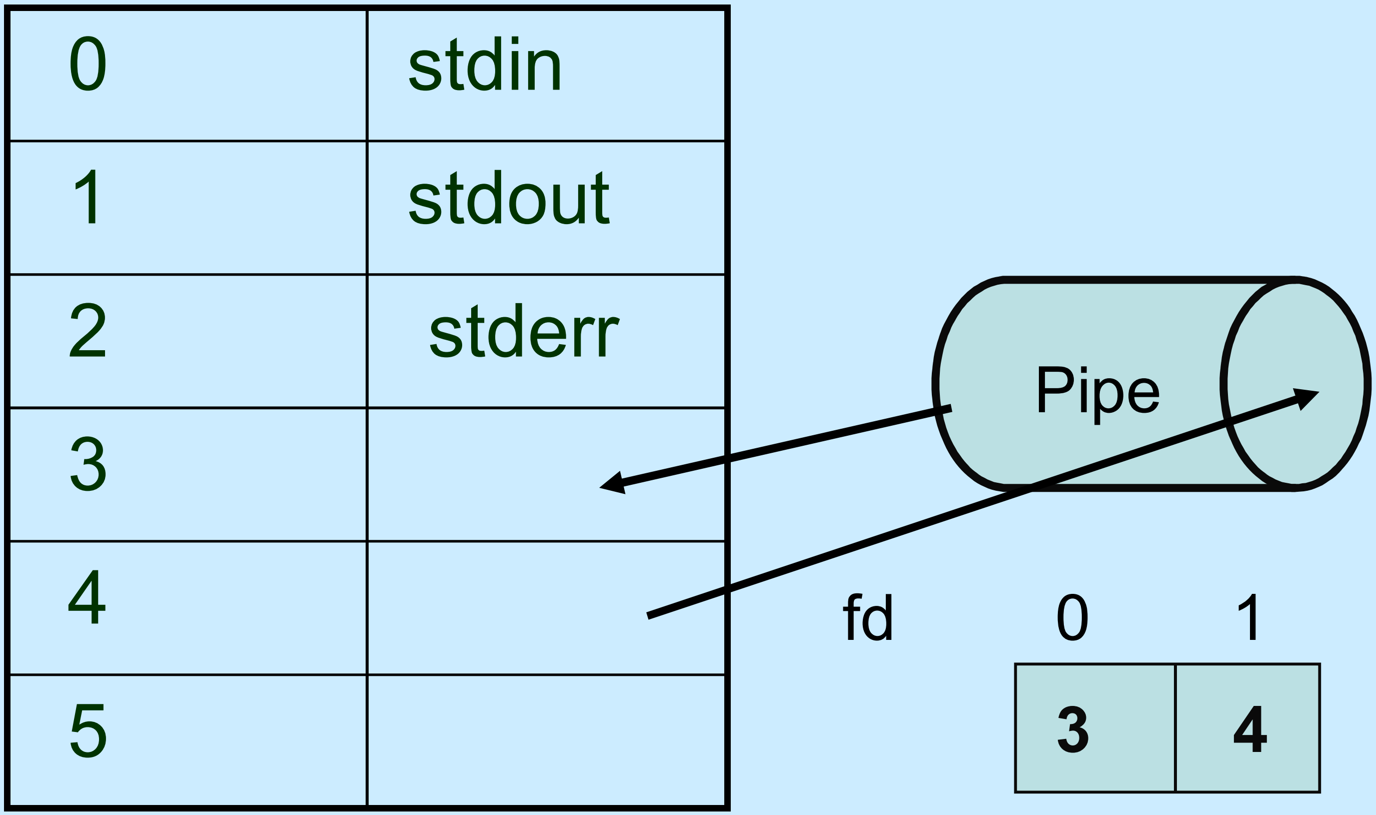
Attribute	Inherited by child	Retained in exec()
PID	No	Yes
Real PID	Yes	Yes
Effective PID	Yes	Depends on *setuid* bit
Data	Copied	No
Stack	Copied	No
Heap	Copied	No
Program Code	Shared	No
File Descriptors	Copied (file ptr is shared)	Usually
Environment	Yes	Depends on *exec()*
Current Directory	Yes	Yes
Signal	Copied	Partially

Programming with Pipes

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

- Returns 0 if ok, -1 on error
- Pipe() binds fd[] with two file descriptors
  - fd[0] is open for reading
  - fd[1] is open for writing
  - Output of fd[1] is input to fd[0]

After the pipe() call



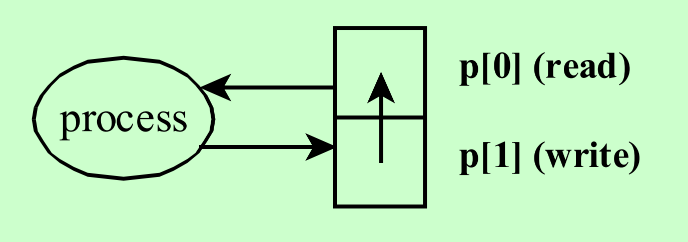
0	stdin
1	stdout
2	stderr
3	
4	
5	

fd	0	1
	3	4

Example: pipe-yourself.c

```
#include <stdio.h>
#include <unistd.h>
#define MSGSIZE 16 /* null */
char *msg1=“hello, world #1”;
char *msg2=“hello, world #2”;
char *msg3=“hello, world #3”;
int main()
    {
    char inbuf[MSGSIZE];
    int p[2], i;
    if( pipe( p ) < 0 )
        { /* open pipe */
        perror( “pipe” );
        exit( 1 ); }
    write( p[1], msg1, MSGSIZE );
    write( p[1], msg2, MSGSIZE );
    write( p[1], msg3, MSGSIZE );
    for( i=0; i < 3; i++ )
        { /* read pipe */
        read( p[0], inbuf, MSGSIZE );
        printf( “%s\n”, inbuf ); }
    return 0;
    }
```

```
$ a.out
hello, world #1
hello, world #2
hello, world #3
$
```



Things to Note

- Pipes uses FIFO ordering: *first-in first-out*.
- Read/write amounts do not need to be the same, but then text will be split differently.
- Pipes are most useful with `fork()` which creates an IPC connection between the parent and the child (or between the parents children)

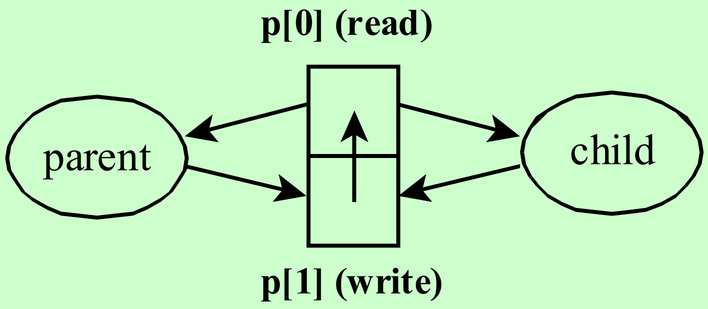
Example: pipe_fork.c

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#define MSGSIZE 16
char *msg1=“hello, world #1”;
char *msg2=“hello, world #2”;
char *msg3=“hello, world #3”;
int main()
   {
   char inbuf[MSGSIZE];
   int p[2], i, pid;
   if( pipe( p ) < 0 )
        { /* open pipe */
        perror( “pipe” );
        exit( 1 );
        }
   if( (pid = fork()) < 0 )
        {
        perror( “fork” );
        exit( 2 );
        }
```

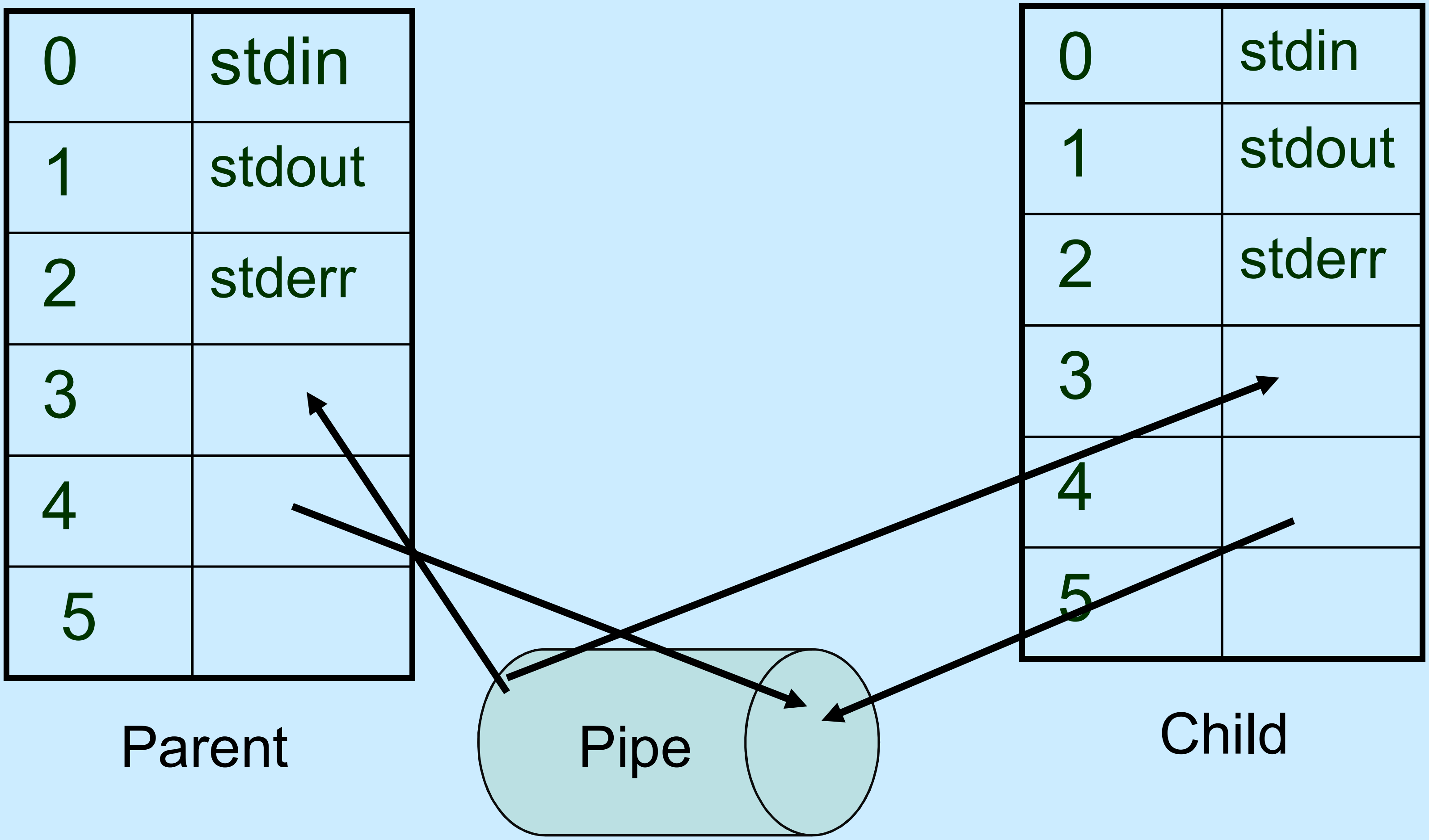
Cont’d

```
else if( pid > 0 ) /* parent */
     {
     write( p[1], msg1, MSGSIZE );
     write( p[1], msg2, MSGSIZE );
     write( p[1], msg3, MSGSIZE );
     wait( (int *) 0 );
     }
else if( pid == 0 ) /* child */
     {
     for( i=0; i < 3; i++ )
            {
            read( p[0], inbuf, MSGSIZE );
            printf( “%s\n”, inbuf );
            }
     }
return 0;
}
```

```
$ a.out
hello, world #1
hello, world #2
hello, world #3
$
```



Another look



0	stdin
1	stdout
2	stderr
3	
4	
5	

Parent

Pipe

0	stdin
1	stdout
2	stderr
3	
4	
5	

Child

Things to Note

- Notice that both parent and child can read/write to the pipe
- Possible to have multiple readers/writers attached to a pipe
  - Can causes confusion
- Best style is to close links you do not need
  - i.e, we close the read end in one process and the write end in the other process
  - For our example, the read end of the parent and the write end of the child

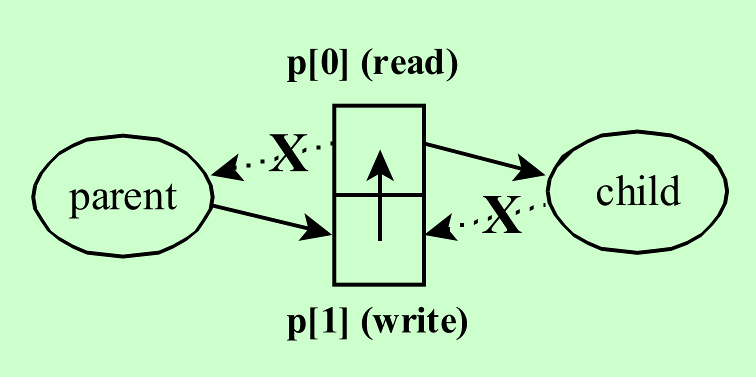
Example: pipe_fork_close.c

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#define MSGSIZE 16
char *msg1=“hello, world #1”;
char *msg2=“hello, world #2”;
char *msg3=“hello, world #3”;
int main()
   {
   char inbuf[MSGSIZE];
   int p[2], i, pid;
   if( pipe( p ) < 0 )
        { /* open pipe */
        perror( “pipe” );
        exit( 1 );
        }
   if( (pid = fork()) < 0 )
        {
        perror( “fork” );
        exit( 2 );
        }
```

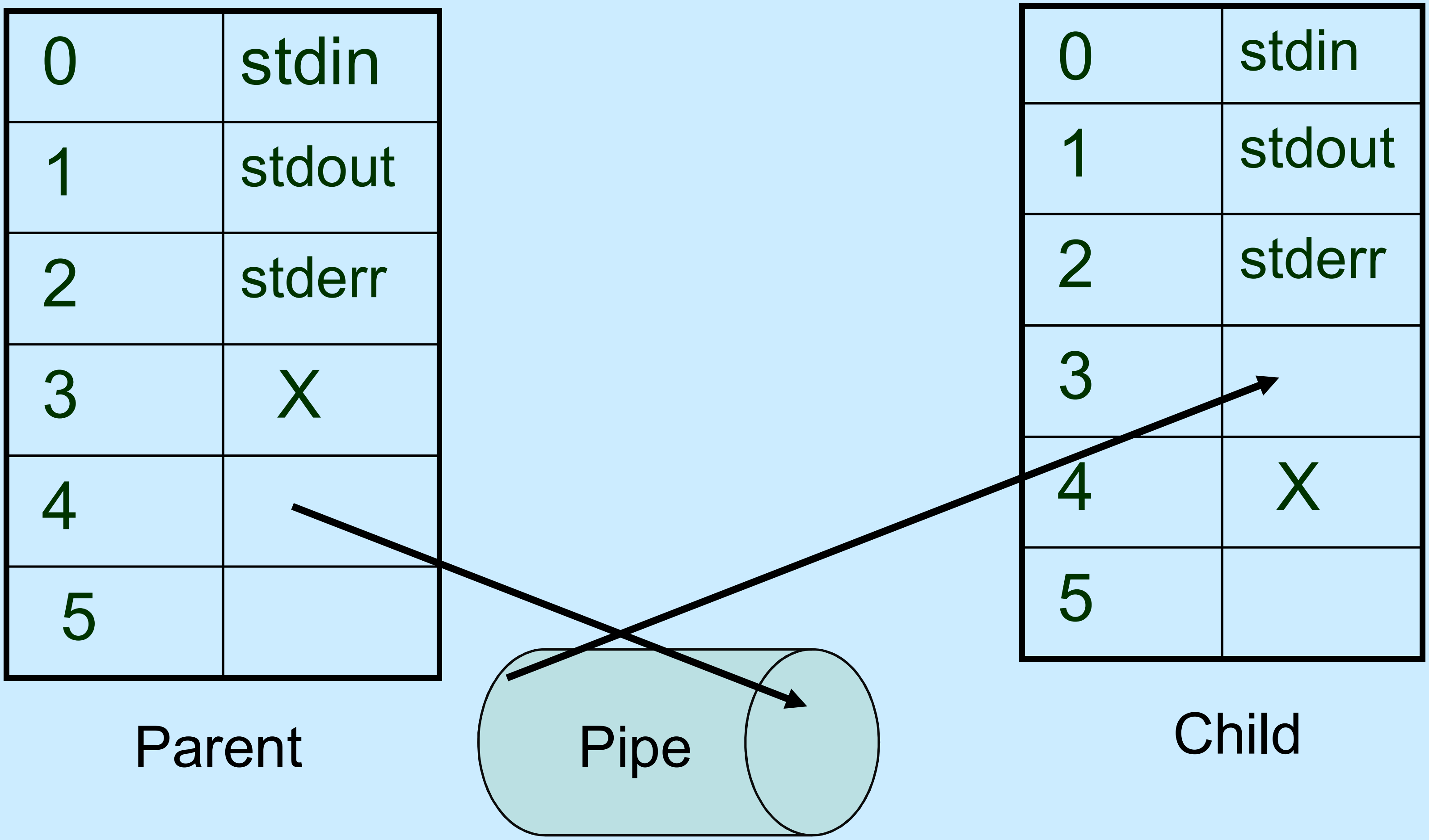
Cont’d

```
else if( pid > 0 ) /* parent */
     {
     close( p[0] ); /* read link */
     write( p[1], msg1, MSGSIZE );
     write( p[1], msg2, MSGSIZE );
     write( p[1], msg3, MSGSIZE );
     wait( (int *) 0 );
     }
else if( pid == 0 ) /* child */
     {
     close( p[1] ); /* write link */
     for( i=0; i < 3; i++ )
             {
             read( p[0], inbuf, MSGSIZE );
             printf( “%s\n”, inbuf );
             }
     }
return 0;
}
```

```
$ a.out
hello, world #1
hello, world #2
hello, world #3
$
```



Another look



0	stdin
1	stdout
2	stderr
3	X
4	
5	

Parent

0	stdin
1	stdout
2	stderr
3	
4	X
5	

Child

Rules of Pipes

- **Every pipe has a size limit**
  - **POSIX minimum is 512 bytes (most systems makes this figure larger … for Solaris it is 5120 bytes)**

- **`read()` blocks if pipe is empty *and* there is a write link open to that pipe**
  - **Close write links or read() will never return**

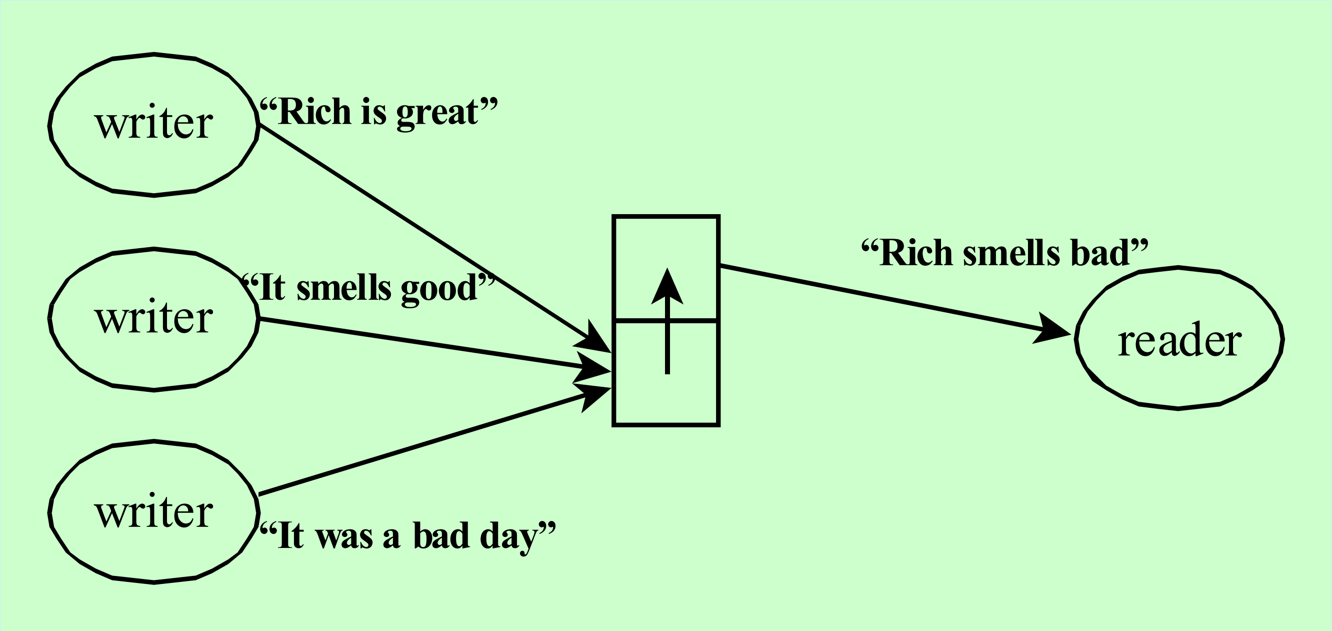
- **`read()` from a pipe whose `write()` end is closed *and* is empty returns 0 (indicates EOF)**

- **`write()` to a pipe with no read() ends returns -1 and generates SIGPIPE and `errno` is set to EPIPE**

- **`write()` blocks if the pipe is full or there is not enough room to support the `write()`.**
  - **May block in the middle of a write()**

Several Writers

- Since a `write()` can suspend in the middle of its output then output from multiple writers may be mixed up (*interleaved*).



- In `limits.h`, the constant `PIPE_BUF (512-4096)` gives the maximum number of bytes that can be output by a `write()` without any chance of interleaving
- Use `PIPE_BUF` if there are to be multiple writers in your code

Non-blocking read() & write()

- **Problem:**
  - Sometimes you want to prevent `read()` and `write()` from blocking.
- **Goals:**
  - want to return an error code instead
  - want to poll several pipes in turn until one has data
- **Approaches:**
  - Use fstat() on the pipe to get the number of characters in pipe (caveat: multiple readers may give a race condition)
  - Use fcntl() on the pipe and set it to O_NONBLOCK

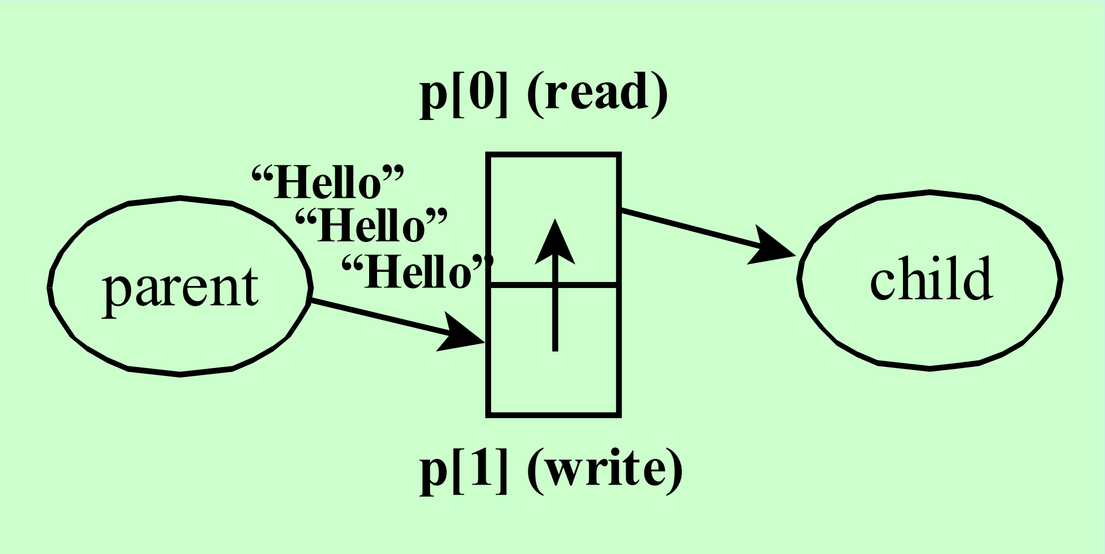
Using `fcntl()`

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
      :
if( fcntl( fd, F_SETFL, O_NONBLOCK ) < 0 )
  perror(“fcntl”);
      :
```

- **Non-blocking write**: On a write-only file descriptor, `fd`, future `writes` will never block
  - Instead return immediately with a -1 and set errno to `EAGAIN`
- **Non-blocking read**: On a read-only file descriptor, `fd`, future `reads` will never block
  - return -1 and set errno to `EAGAIN` or return 0 if pipe is empty (or closed)

Example: Non-blocking with -1 return

- **Child writes “hello” to parent every 3 seconds (3 times).**
- **Parent does a non-blocking read each second.**



Example: pipe_nonblocking.c

```
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#define MSGSIZE 6
char *msg1=“hello”;

void parent_read( int p[] );
void child_write( int p[] );

int main()
   {
   int pfd[2];
   if( pipe( pfd ) < 0 )
       { /* open pipe */
       perror( “pipe” );
       exit( 1 );
   }
```

main Cont’d

```
if( fcntl( pfd[0], F_SETFL, O_NONBLOCK ) < 0 )
    { /* read non-blocking */
    perror( “fcntl” );
    exit( 2 );
    }
switch( fork() )
    {
    case -1: /* error */
           perror(“fork” );
           exit(3);
    case 0: /* child */
           child_write( pfd );
           break;
    default: /* parent */
           parent_read( pfd );
           break;
    }
return 0;
}
```

void parent_read()

```
void parent_read( int p[] )
   {
   int nread, done = 0;
   char buf[MSGSIZE];
   close( p[1] ); /* write link */
   while( !done )
       {
       nread = read( p[0], buf, MSGSIZE );
       switch( nread )
              {
              case -1:
                     if( errno == EAGAIN )
                            {
                            printf(“(pipe empty)\n”);
                            sleep( 1 );
                            break;
                            }
```

Cont’d

```
                 else
                        {
                        perror( “read” );
                        exit(4);
                        }
           case 0:
                 /* pipe has been closed */
                 printf( "End conversation\n" );
                 close( p[0] ); /* read fd */
                 exit(0);
           default: /* text read */
                 printf( "MSG=%s\n", buf );
           } /* switch */
    } /* while */
} /* parent_read */
```

void child_write()

```
void child_write( int p[] )
   {
   int i;
   close( p[0] ); /* read link */
   for( i = 0; i < 3; i++ )
        {
        write( p[1], msg1, MSGSIZE );
        sleep( 3 );
        }
   close( p[1] ); /* write link */
   }
```

```
$ a.out
MSG=hello
(pipe empty)
(pipe empty)
(pipe empty)
MSG=hello
(pipe empty)
(pipe empty)
(pipe empty)
MSG=hello
(pipe empty)
(pipe empty)
(pipe empty)
End conversation
$
```

Limitations of Pipes

- Processes using a pipe must come from a common ancestor:
  - e.g. parent and child
  - cannot create general servers like print spoolers or network control servers since unrelated processes cannot use it
- Pipes are not permanent
  - they disappear when the process terminates
- Pipes are sometimes one-way:
  - makes fancy communication harder to code
- Pipes do not work over a network

What are FIFOs/Named Pipes?

- Similar to pipes (as far as `read/write` are concerned, e.g. FIFO channels), but with some additional advantages:
  - Unrelated processes can use a FIFO.
  - A FIFO can be created separately from the processes that will use it.
  - FIFOs look like files:
    - have an owner, size, access permissions
    - open, close, delete like any other file
    - permanent until deleted with `rm`

Creating a FIFO

- **UNIX `mkfifo` command:**

```
$ mkfifo fifo1
```

- **On older UNIXs (original ATT UNIX), use `mknod`:**

```
$ mknod fifo1 p
```

- **Use `ls` to get information:**

```
$ ls -l fifo1
prw------- 1 rhurley staff 0 Jul 3 12:02 fifo1
```

Using FIFOs: FIFO Blocking

- FIFOs can be read and written using standard UNIX commands connected via “<“ and “>” (a commands input or output)

- If there are no writers then a read:
  - e.g. `cat < fifo1`

  will block until there is 1 or more writers.

- If there are no readers then a write:
  - e.g. `ls -l > fifo1`

  will block until there is 1 or more readers

Reader / Writer Example

```
$ cat < fifo1 &
[1] 22341
$ ls -l > fifo1; wait
total 17
prw-rw-r-- 1 rhurley staff 0 Jul 3 12:15 fifo1
[1] Done cat < fifo1
$
```

1. Output of `ls -l` is written down the FIFO
2. Waiting `cat` reads from the FIFO and display the output
3. `cat` exits since `read` returns 0 (the FIFO is not open for writing anymore and 0 is returned as EOF)

`wait` - causes the shell to wait until `cat` exits before redisplaying the prompt

Creating a FIFO in C

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

- **Returns 0 if OK, -1 on error.**
- **`mode` is the same as for `open()` - and is modifiable by the process' `umask` value**
- **Once created, a FIFO must be opened using `open()`**

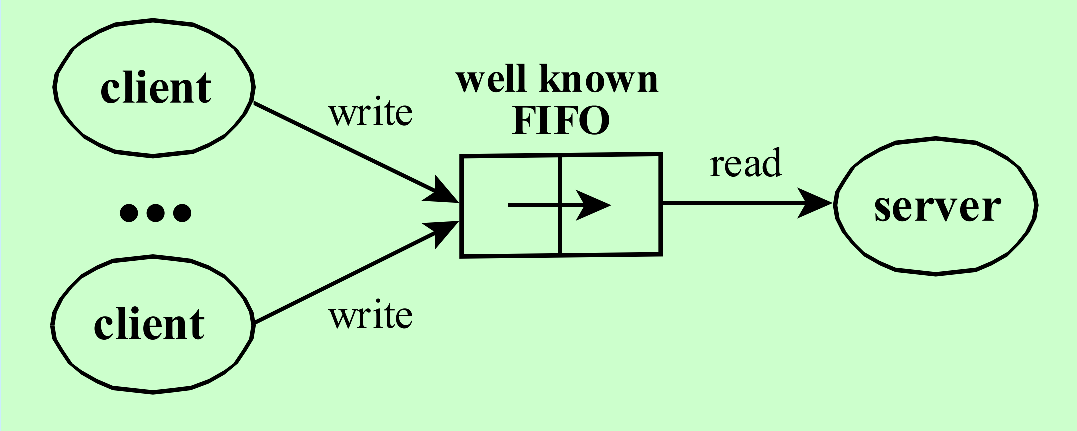
Two Main Uses of FIFOs

1. Used by shell commands to pass data from one shell pipeline to another without using temporary files.

2. Create client-server applications on a single machine.

Client-Server Applications

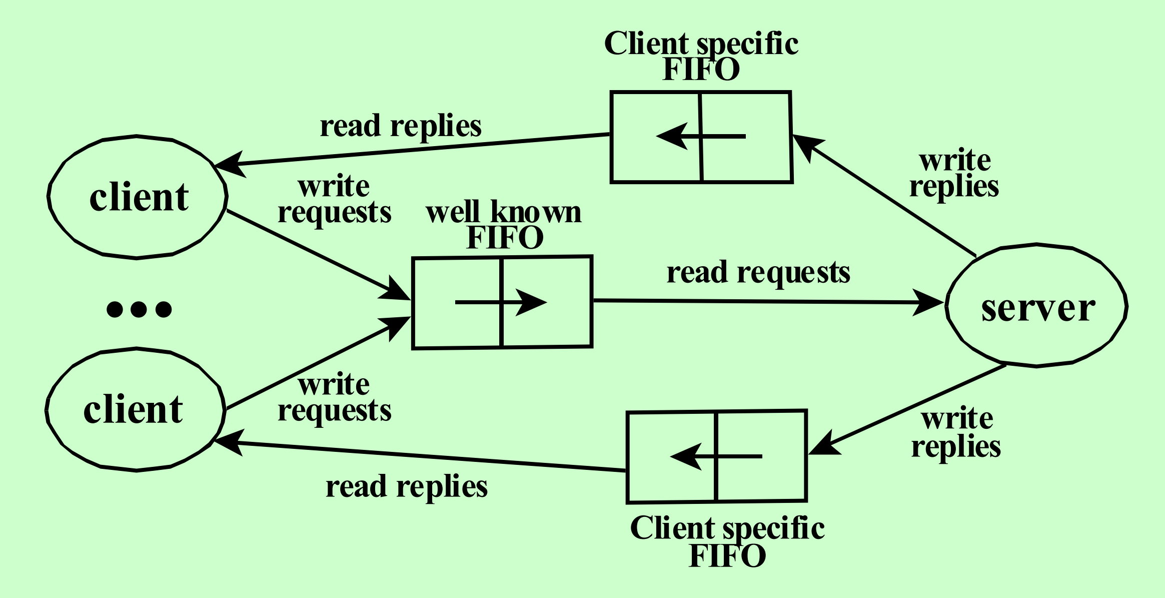
- Server contacted by numerous clients via a well-known FIFO



- How are replies from the server sent back to each client?

Client-Server FIFO Application

- **Problem:** A single FIFO (as before) is not enough.
- **Solution:** Each client send its PID as part of its message. Which it then uses to create a special ‘reply’ FIFO for each client
  - e.g. `/tmp/serv1.XXXX` where `XXXX` is replaced with the clients process ID



Problems

- **The server does not know if a client is still alive**
  - **may create FIFOs which are never used**
  - **client terminates before reading the response (leaving FIFO with one writer and no reader)**

- **Each time number of clients goes from 1 client to 0 the server reads an EOF on the well-known FIFO, if it is set to read-only.**
  - **Common trick is to have the server open the FIFO as read-write**

Programming Client-server Applications

- First we must see how to `create`, `open` and `read` a FIFO from within C.

- Clients will write in non-blocking mode, so they do not have to wait for the server process to start.

Creating a FIFO

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

- **Creates a FIFO file named by `pathname`**
- **The FIFO will be given `mode` permissions (0666)**
- **Can be modified using the process' `umask` value**

Opening FIFOs

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
:
fd = open( “fifo1”, O_WRONLY );
:
```

- A FIFO can be opened with `open()` (most I/O functions work with pipes).

Blocking `open()`

- An `open()` call for *writing* will block until another process opens the FIFO for *reading*.
  - this behavior is not suitable for a client who does not want to wait for a server process before sending data.

- An `open()` call for *reading* will block until another process opens the FIFO for *writing*.
  - this behavior is not suitable for a server which wants to poll the FIFO and continue if there are no readers at the moment.

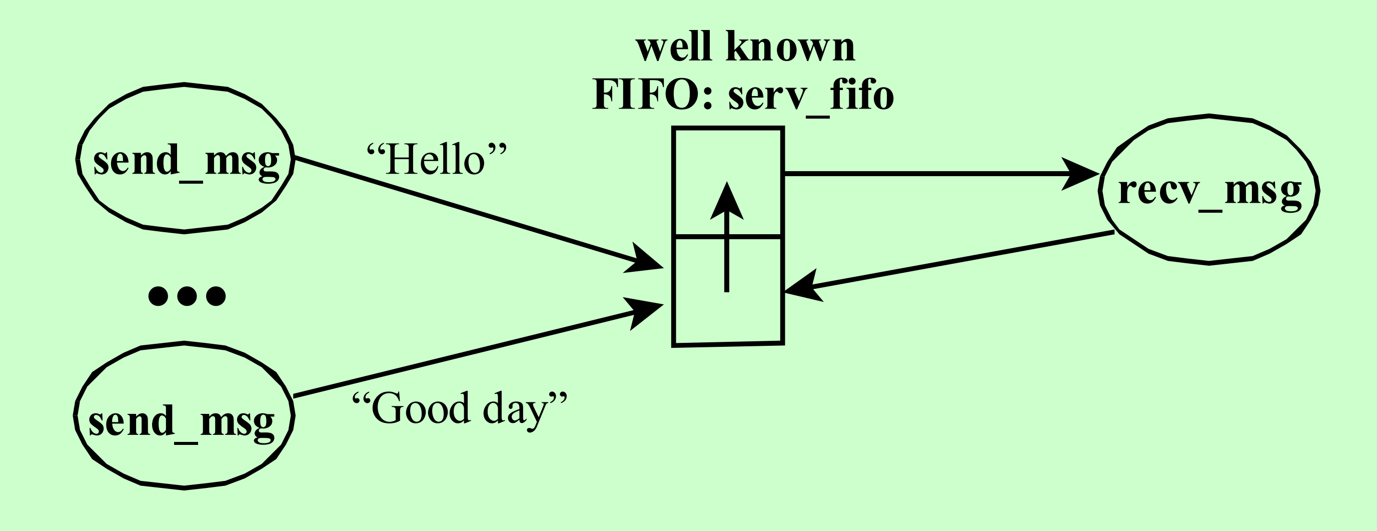
Non-blocking `open()`

```
if ( fd = open( “fifo1”, O_WRONLY | O_NONBLOCK)) < 0 )
    perror( “open FIFO” );
```

- `opens` the FIFO for writing
- returns -1 and `errno` is set to `ENXIO` if there are no readers, instead of blocking.
- Later `write()` calls will also not block.

Example: send_msg, recv_msg

- implement a message system
- exploits the fact that `reads/writes` to `pipes/FIFOs` are atomic
- if fixed-sized messages are passed, individual messages will stay intact even with concurrent senders



Notes:

- `recv_msg` can **read and write**;
  - otherwise the program would block at the open call
  - also avoids responding to reading a "return of 0" when the number of `send_msg` processes goes from 1 to 0 (and the FIFO is empty) O_RDWR - ensures that at least one process has the FIFO open for writing (i.e. `recv_msg` itself) so read will always block until data is written to the FIFO

- `send_msg` sends **fixed-size** messages of length `PIPE_BUF` to avoid interleaving problems with other `send_msg` calls. It uses non-blocking.

- `serv_fifo` is globally known, and previously created with `mkfifo`

Header for files

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>
#define SF "serv_fifo"
```

send_msg.c

```
void make_msg( char mb[], char input[]);
int main( int argc, char *argv[] )
   {
   int fd, i;
   char msgbuf[PIPE_BUF];
   if( argc < 2 )
        {
        printf( "Usage: send-msg msg...\n" );
        exit( 1 );
        }
   if( (fd = open( SF, O_WRONLY | O_NONBLOCK )) < 0)
        { perror( SF ); exit( 1 ); }
   for( i = 1; i < argc; i++ )
        {
        if( strlen( argv[i] ) > PIPE_BUF - 2 )
                printf( "Too long: %s\n", argv[i] );
        else
                {
                make_msg( msgbuf, argv[i] );
                write( fd, msgbuf, PIPE_BUF );
                }
        }
   close( fd );
   return 0;
   } /* end main */
```

send_msg.c cont’d

```
/* put input message into mb[] with '$` and padded with spaces */
void make_msg( char mb[], char input[])
   {
   int i;
   for( i = 1; i < PIPE_BUF-1; i++ )
        mb[i] = ' ';
   mb[i] = '\0';
   i = 0;
   while( input[i] != ‘\0’ )
        {
        mb[i] = input[i];
        i++;
        }
   mb[i] = '$';
   } /* make_msg */
```

recv_msg.c

```
void print_msg( char mb[] );
int main( int argc, char *argv[] )
   {
   int fd, I, done = 0;
   char msgbuf[PIPE_BUF];
   if(mkfifo(SF,0666) == -1)
        if(errno != EEXIST)
                { perror("receiver: mkfifo");
                exit( 1 ); }
   if( (fd = open( SF, O_RDWR )) < 0 )
        { perror( SF );
        exit( 1 ); }
   while( !done )
        {
        if( read( fd, msgbuf, PIPE_BUF ) < 0 )
                {
                perror( "read" );
                exit( 1 );
                }
        print_msg( msgbuf );
        }
   close( fd );
   return 0;
   } /* end main */
```

recv_msg.c cont’d

```
/* print mb[] up to the '$' marker */
void print_msg( char mb[] )
   {
   int i = 0;
   printf( "Msg: " );
   while( mb[i] != '$' )
        {
        putchar( mb[i] );
        i++;
        }
   putchar( '\n' );
   } /* make_msg */
```

```
$ send_msg "Hello"
serv_fifo: No such file or directory
$ recv_msg &
[1] 8323
$ send_msg "Hello"
$ Msg: Hello
send_msg "Nice to see you"
Msg: Nice to see you
$ send_msg "This" "is" "four" "messages"
Msg: This
Msg: is
Msg: four
Msg: messages
$ kill -9 %1
[1]    Killed              recv_msg
```