

UNIX System Programming

Pipes/FIFOs

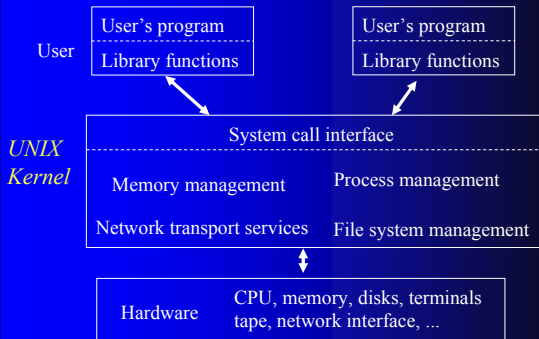
❖ Objectives

- Look at UNIX support for interprocess communication (IPC) on a single machine
- Review processes
- pipes, FIFOs

Overview

1. A UNIX System (review)
2. Processes (review)
3. Pipes
4. FIFOs

1. A UNIX System



2. Processes (review)

- ❖ *Definition:* A process is the *context* (the information) maintained for an executing program.

2.1. Context

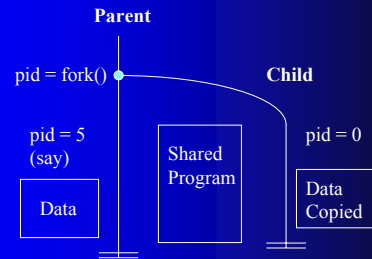
- Process ID (pid) unique integer
- Parent process ID ($ppid$)
- Real User ID ID of user/process which started this one
- Effective User ID ID of user who wrote the program
- Current Directory
- File Descriptor table
- Environment VAR=VALUE pairs

- Program code
- Data Memory for global vars
- Stack Memory for local vars
- Heap Malloc'd memory
- Execution priority
- Signal Information
- umask value

2.2. fork() Reminder

- ❖ `#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.

fork() in Pictures



fork() Example

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int i;
    if( fork() > 0 )
    { /* parent */
        for (i=0; i < 1000; i++)
            printf("\t\t\tPARENT %d\n", i);
    }
}
```

```
else
{ /* child */
    for (i=0; i<1000; i++)
        printf("CHILD %d\n", i);
}
return 0;
}
```

Possible Output

```
CHILD 0
CHILD 1
CHILD 2
PARENT 0
PARENT 1
PARENT 2
PARENT 3
CHILD 3
CHILD 4
PARENT 4
:
```

Things to Note

- ❖ `i` is copied between parent and child.
- ❖ The switching between the parent and child depends on many factors:
 - machine load, system process scheduling
- ❖ I/O buffering effects amount of output shown.
- ❖ Output interleaving is *nondeterministic*
 - cannot determine output by looking at code

2.4. exec() Reminder

- ❖ Family of functions for replacing process's program with the one inside the `exec()` call.

- ❖ e.g.

```
int execlp(char *file, char *arg0,
           char *arg1, ..., (char *)0);

execlp("sort", "sort", "-n",
       "foobar", (char *)0);
```

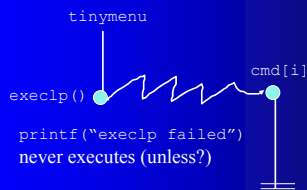
tinymenu.c

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

void main()
{
    char *cmd[] = {"who", "ls", "date"};
    int i;
    printf("\0=who 1=ls 2=date : ");
    scanf("%d", &i);

    execlp(cmd[i], cmd[i], (char *)0);
    printf("execlp failed\n");
}
```

Execution



2.5. wait() Reminder

- ❖ `#include <sys/types.h>`
`#include <sys/wait.h>`
`pid_t wait(int *statloc);`
- ❖ Used by parent to wait for any of its children to terminate.
- ❖ Possible to extract `pid` of terminated child and its `exit` status.

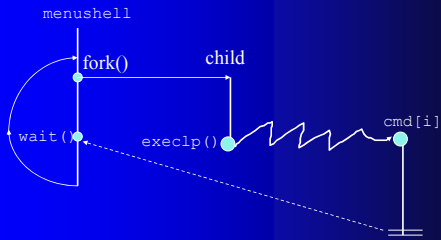
menushell.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void main()
{
    char *cmd[] = {"who", "ls", "date"};
    int i;
    while(1)
    {
        printf("\0=who 1=ls 2=date : ");
        scanf("%d", &i);
        ;
    }
}
```

```
if (fork() == 0)
{ /* child */
    execlp(cmd[i], cmd[i], (char *)0);
    printf("execlp failed\n");
    exit(1);
}
else
    /* parent */
    wait((int *)0);
} /* while */
```

Execution



2.6. Context used by child & exec()

Attribute	Inherited by child?	Retained in exec()
- pid	No	Yes
- real uid	Yes	Yes
- effective uid	Yes	Depends on setuid bit
- Data	Copied	No
- Stack	Copied	No
- Heap	Copied	No
- Program Code	Shared	No

Attribute	Inherited by child?	Retained in exec()
- File Descriptors	Copied (but file ptrs shared)	Usually
- Environment	Yes	Depends on exec()
- Current Dir	Yes	Yes
- signal	Copied	Partially

3. Pipes

- ❖ A pipe 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
- ❖ Pipes at UNIX level:
 - `who | wc -l`

3.1. Programming with Pipes

- ❖ `#include <unistd.h>`
`int pipe(int fds[2]);`
- ❖ Returns 0 if ok, -1 on error.
- ❖ `pipe()` binds `fds[]` with two file descriptors:
 - `fds[0]` used to read from pipe
 - `fds[1]` used to write to pipe

3.2. Pipe to Yourself

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

#define MSGSIZE 16 /* text + 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;
    :
```

```

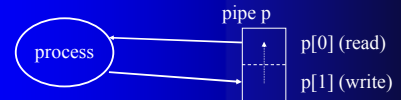
if (pipe(p) < 0)
    { /* open pipe */
    perror( "pipe" );
    exit( 1 );
    }

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;
}

```

Execution



- Output:


```

hello, world #1
hello, world #2
hello, world #3

```

Things to Note

- ❖ 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()`.

3.3. pipe() and fork()

```

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

#define MSGSIZE 16 /* text + 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, pid;
    :
}

```

```

if( pipe(p) < 0 )
    { /* open pipe */
    perror( "pipe" );
    exit(1);
    }
if( (pid = fork()) < 0 )
    {
    perror( "fork" );
    exit(2);
    }
if(pid > 0) { /* parent */
    write( p[1], msg1, MSGSIZE );
    write( p[1], msg2, MSGSIZE );
    write( p[1], msg3, MSGSIZE );
    wait( (int *)0);
    }
:

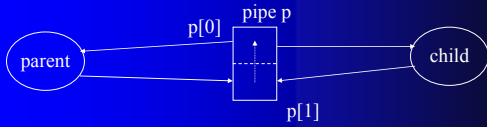
```

```

if( pid == 0 )
    { /* child */
    for( i=0; i < 3; i++ )
        {
        read( p[0], inbuf, MSGSIZE );
        printf( "%s\n", inbuf );
        }
    }
return 0;
}

```

Diagram



Things to Note

- ❖ Perfectly possible to have multiple readers / writers attached to a pipe
 - can cause confusion
- ❖ Best style is for a process to close the links it does not need. Also avoids problems (see below).

3.4. pipe() and fork() Again

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

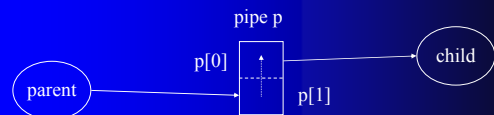
#define MSGSIZE 16 /* text + null */
char *msg1="hello, world #1";
char *msg2="hello, world #2";
char *msg3="hello, world #3";

int main()
{
    char inbuf[MSGSIZE];
    int p[2], i, pid;
    :
```

```
    if (pipe(p) < 0)
        { /* open pipe */
            perror("pipe");
            exit(1);
        }
    if ((pid = fork()) < 0)
        {
            perror("fork");
            exit(2);
        }
    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);
        }
```

```
    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;
}
```

Diagram



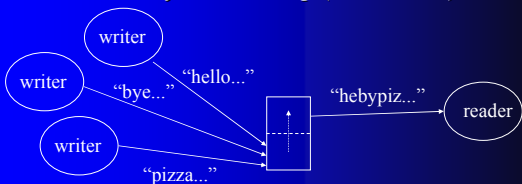
3.5. Size of a Pipe

- ❖ Every pipe has a size limit
 - POSIX minimum (512 bytes)
- ❖ `write()` will suspend (block) if there is not enough room in the pipe for its text.
 - may block in the middle of its write
- ❖ Writing to a pipe with no read links returns `-1`, and `errno` is set to `EPIPE`.

- ❖ `read()` will block if the pipe is empty and there is a write link open to that pipe.
- ❖ `read()` will return 0 if the pipe is empty and there are *no* write links to the pipe.
 - remember to close write links or `read()` calls will never return.

3.6. Several Writers

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



Avoid Interleaving

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

3.7. Non-blocking `read()` & `write()`

- ❖ Sometimes you want to avoid `read()` and `write()` from blocking.
- ❖ Examples:
 - want to return an error instead
 - want to poll several pipes in turn until one has data

3.7.1. 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 fd

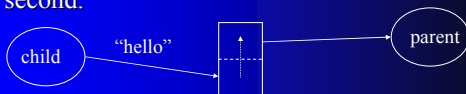
- ❖ `write()` call will not block if the pipe is full.
- ❖ `write()` will return -1 and `errno` be set to `EAGAIN`.
- ❖ In old UNIXs (mid 1980's), `write()` returns 0.

Non-blocking read fd

- ❖ `read()` call will not block if the pipe is empty.
- ❖ `read()` will return -1 and `errno` be set to `EAGAIN`.
- ❖ In old UNIXs (mid 1980's), `read()` returns 0:
 - cannot tell if pipe is empty or closed

3.7.2. Non-blocking with -1 error

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



```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

#define MSGSIZE 6 /* text + null */
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 );
        }

    /* make read link non-blocking */
    if( fcntl(pfd[0],F_SETFL, O_NONBLOCK) < 0 )
        {
            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(int p[])
{
    int nread;
    char buf[MSGSIZE];
    close(p[1]); /* write link */
    while(1)
    {
        nread = read(p[0], buf, MSGSIZE);
        switch(nread)
        {
            case -1:
                if (errno == EAGAIN)
                {
                    printf("(pipe empty)\n");
                    sleep(1);
                    break;
                }
                else
                {
                    perror("read");
                    exit(4);
                }
        }
    }
}

```

```

case 0:
    printf("End of conversation\n");
    close(p[0]); /* read link */
    exit(0);
default: /* text read */
    printf("MSG=%s\n", buf);
}
}
}

```

```

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 */
}

```

Output

```

(pipe empty)
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 of conversation

3.7.3. Non-blocking with 0 error

- ❖ If non-blocking `read()` does not distinguish between end-of-input and empty pipe then can use special message to mean end:
 - e.g. send “bye” as last message

3.8. Pipes and `exec()`

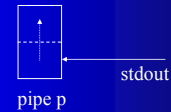
- ❖ How can we code `who | sort` ?
- ❖ Use `exec()` to start two processes which share a pipe.
- ❖ Connect the pipe to `stdin` and `stdout` using `dup2()`.

3.8.1. dup2() Reminder

- ❖ `#include <unistd.h>`
`int dup2(int old-fd, int new-fd);`
- ❖ Existing file descriptor, `old-fd`, is duplicated onto `new-fd`.
- ❖ If `new-fd` already exists, it will be closed first.

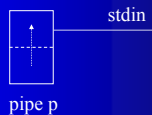
Write pipe fd + stdout

- ❖ Connect the write end of a pipe to `stdout`.
- ❖ `int p[2];`
`pipe(p);`
`dup2(p[1], 1);`



Read pipe fd + stdout

- ❖ Connect the read end of a pipe to `stdin`.
- ❖ `dup2(p[0], 0);`

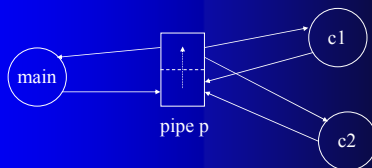


3.8.2. Four Stages to who | sort

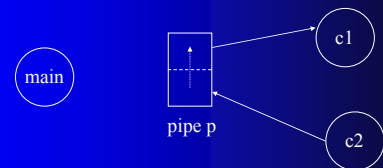
- 1) `main()` creates a pipe



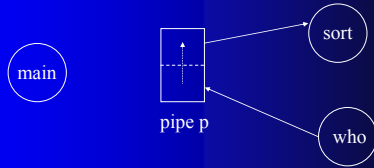
- 2) `main()` forks twice to make two children. They inherit the pipe's file descriptors.



- 3) Close the pipe links which are not needed.



4) Replace children by programs using
`exec()`.



3.8.3. whosort.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

int main()
{
    int fds[2];

    pipe(fds); /* no error checks */
    ;
}
```

```
if (fork() == 0)
{ /* 1st child */
/* fds[0]/stdin --> sort */
dup2(fds[0], 0);
close(fds[1]);
execlp("sort", "sort", (char *)0);
}
else
{ /* parent */
if (fork() == 0)
{ /* 2nd child */
/* who --> fds[1]/stdout */
dup2(fds[1], 1);
close(fds[0]);
execlp("who", "who", (char *)0);
}
else { /* parent */
;
```

```
/* parent closes its links to the pipe */
close(fds[0]);
close(fds[1]);

/* wait for 2 children */
wait((int *)0);
wait((int *)0);
}
return 0;
}
```

3.9. 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 since unrelated processes cannot use it
- ❖ Pipes are not permanent
 - they disappear when the process terminates

- ❖ Pipes are one-way:
 - makes fancy communication harder to code
- ❖ Readers and writers do not know each other.
- ❖ Pipes do not work over a network.

4. FIFOs (Named Pipes)

- ❖ Similar to pipes, but with some 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
 - permanent until deleted with `rm`

4.1. Creating a FIFO

- ❖ UNIX `mkfifo` command:

```
$ mkfifo fifo1
```

Default mode is 0666 - umask value
- ❖ On older UNIXs, use `mknod`:

```
$ mknod fifo1 p
```

p means FIFO
- ❖ Use `ls` to get information:

```
$ ls -l fifo1
prw-rw-r-- 1 maria maria 0 Oct 23 11.45 fifo1
```

4.2. Using FIFOs

- ❖ Can read / write to FIFOs with UNIX commands:
 - e.g.

```
$ cat < fifo1 &
```
 - this will hang due to FIFO blocking

FIFO Blocking

- ❖ If there are no writers then a read will block until there is 1 or more writers.
 - e.g. `cat < fifo1`
- ❖ If there are no readers then a write will block until there is 1 or more readers.
 - e.g. `ls -l > fifo1`

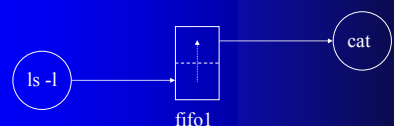
Reader / Writer Example

- ❖

```
$ cat < fifo1 &
[1] 22341

$ ls -l > fifo1; wait
total 17
prw-rw-r-- 1 maria usr 0 Oct 23 11.45 fifo1
:
```
- [1] + Done cat < fifo1
- \$

Diagram



4.3. Creating a FIFO in C

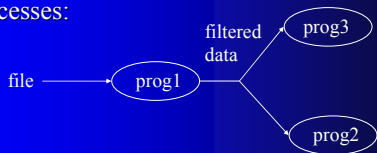
- ❖ `#include <sys/types.h>`
`#include <sys/stat.h>`
- ```
int mkfifo(const char *pathname,
 mode_t mode);
```
- ❖ Returns 0 if ok, -1 on error.
- ❖ `mode` is the same as for `open()`.

## 4.4. 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.

## 4.5. Shell Usage

- ❖ *Example:* process a filtered output stream twice – i.e. pass filtered data to two separate processes:

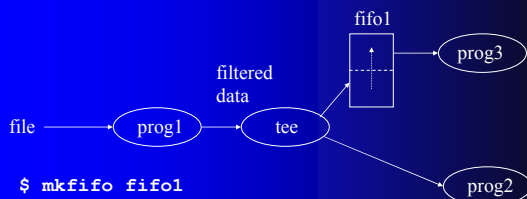


## Example

- ❖ First we must see how to 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.

## Coding

- Unix's `tee()` copies standard input to both its standard input and to the file named on its command line



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

## 4.6. Client-server Usage

- ❖ First we must see how to 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.

## 4.6.1. Opening FIFOs

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

- ❖ 

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

## Blocking open()

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

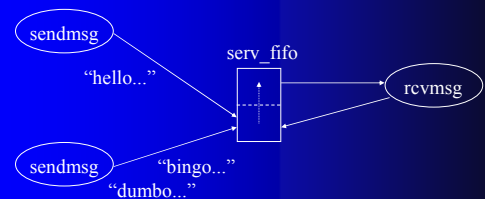
## Non-blocking open()

- ❖ 

```
:
if (fd = open("fifo1", O_RDONLY | O_NONBLOCK)) < 0)
 perror("open FIFO");
```

- ❖ Will return `-1` and `errno` is set to `EAGAIN` if there are no readers.
- ❖ Later `read()` calls will also not block.

## 4.6.2. Example: sendmsg, rcvmsg



## Some Points

- ❖ `rcvmsg` can read and write; the reason is explained later.
- ❖ `sendmsg` sends fixed-size messages of length `PIPE_BUF` to avoid interleaving problems with other `sendmsg` calls. It uses non-blocking.
- ❖ `serv_fifo` is globally known, and previously created with `mkfifo`

## sendmsg.c

```
#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"
void make_msg(char mb[], char input[]);

int main(int argc, cgar *argv[])
{ int fd, i;
 char msgbuf[PIPE_BUF];
 :
```

```

if (argc < 2)
{
 printf("Usage: sendmsg 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[1]);
 write(fd, msgbuf, PIPE_BUF);
 }
close(fd);
return 0;
}

```

*continued*

```

void make_msg(char mb[], char input[])
/* Put input msg into mb[] with '$'
 * and padded with spaces */
{
 int i;
 for (i=0; i < PIPE_BUF-1; i++)
 mb[i] = ' ';
 mb[i] = '\0';

 i = 0;
 while (input[i] != '\0')
 {
 mb[i] = input[i];
 i++;
 }
 mb[i] = '$';
}

```

## rcvmsg.c

```

#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"
void print_msg(char mb[]);

int main()
{
 int fd, i;
 char msgbuf[PIPE_BUF];
 :
}

```

*continued*

```

if ((fd = open(SF, O_RDWR)) < 0)
{
 perror(SF); exit(1);
}

while(1)
{
 if (read(fd, msgbuf, PIPE_BUF) < 0)
 {
 perror("read");
 exit(1);
 }
 /* do something with msg */
 print_msg(msgbuf);
}

close(fd);
return 0;
}

```

*continued*

```

void print_msg(char mb[])
/* Print mb[] up to the '$' */
{
 int i = 0;

 printf("Msg: ");
 while (mb[i] != '$')
 {
 putchar(mb[i]);
 i++;
 }
 putchar('\n');
}

```

## Things to Note about rcvmsg

- ❖ `open()` is blocking, so `read()` calls will block when the pipe is empty
- ❖ `open()` uses `O_RDWR` not `O_RDONLY`
  - this means there is a write link to the FIFO even when there are no `sendmsg` processes
  - this means that a `read()` call will block even when there are no `sendmsg` processes

## Usage

```
$ mkfifo serv_fifo

$ rcvmsg &
[1] 25129

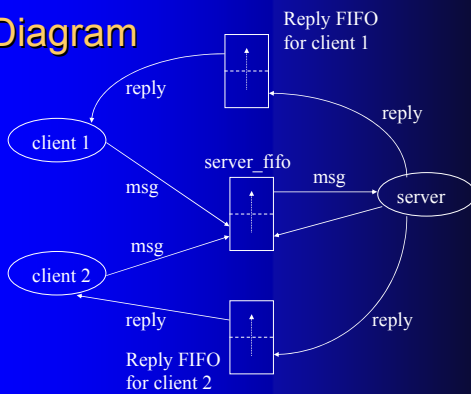
$ sendmsg "message 1" "message 2" &
$ sendmsg "message A" &

Msg: message 1
Msg: message A
Msg: message 2
```

## 4.7. Fancier Client-Servers

- ❖ Often you want to have a server reply to a client's message. How?
- ❖ A single FIFO (as before) is not enough.
- ❖ One solution is to have each client send its PID as part of its message.
- ❖ The server uses the PIDs to create a 'reply' FIFO for each client.

## Diagram



## Problem

- ❖ The server does not know if a client is still alive
  - may create FIFOs which are never used