IPC and Pipes 1: Introduction and Basic Usage

1. **Introduction and Basic Usage**
   - **IPC mechanisms**
   - **pipes and pipe() call**
   - **pipe characteristics**
   - **standard usage pattern**
2. Usage Details
3. FIFOs

Interprocess Communication

Interprocess Communication (**IPC**) refers to mechanisms for sending data from one process to another.

Linux/UNIX supports a number of IPC mechanisms:

- pipes and FIFOs
- sockets
- shared memory
- message queues
- signals

(Process **synchronization** mechanisms like **semaphores** can also be considered as IPC in a very broad sense.)

IPC (contd.)

IPC mechanisms can differ in a number of dimensions:

- data transfer direction:
  unidirectional vs. **bidirectional**
- data formatting:
  stream (of bytes) vs. separate **messages**
- process relationship requirements:
  related (*share a common ancestor*) vs. unrelated
- machine(s) hosting the processes:
  same machine only vs. different, **networked** machines
- data read order:
  always in write order vs. allow reading "**out-of-band**" (prioritized) data first
- able to transmit special information:
  e.g., file descriptor link

Pipes

Pipes are the oldest and simplest IPC mechanism, but also the most used IPC mechanism.

A pipe is a **unidirectional**, **stream-oriented** IPC mechanism that can be used between **related processes** that are running on the **same machine**.

Most users are familiar with pipes from their use in shell **pipelines**.

Consider a command like: `grep printf lab1.c | wc -l`

This will cause the shell to create a pipe and two subprocesses, with the `grep` process sending its output to the `wc` process via the pipe.

pipe() System Call

`pipe()` is the system call to create a pipe:
`int pipe(int pipefd[2])`

- `pipefd` is a two-element `int` array into which *file descriptors* for the *read and write ends* of the pipe will be placed
- `pipefd[0]` is the *read end* FD
- `pipefd[1]` is the *write end* FD
- these FDs can be used with `read()` and `write()` to send and receive information between processes
- returns 0 on success, -1 on error

Related Processes on Same Machine

Logically, using a pipe to send data from one process to another is equivalent to having one process write data to a *file* and the other read data from that file.

A pipe is a **kernel buffer**, however, so pipe I/O is much faster than file I/O.

Unlike a file, however, basic, *unnamed* pipes do *not* have an entry in the filesystem.

This means that the only way for a second process to get access to a pipe created by another process is by *inheriting the pipe's file descriptors* (e.g., across a `fork()` call).

Thus, pipes are limited to IPC among processes that are *"related"* (*share a common ancestor*), such as parent-child or siblings.

Standard Usage Pattern

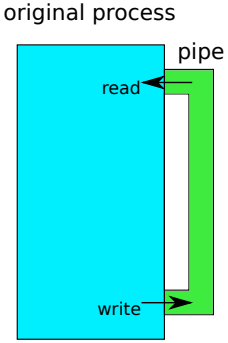
Becuase pipe file descriptors must be inherited, this virtually always means that `pipe()` will be called and only later will one or more `fork()` calls be made.

The child created by `fork()` will *inherit* the file descriptors for the read and write ends of the pipe (since it is a copy of its parent).

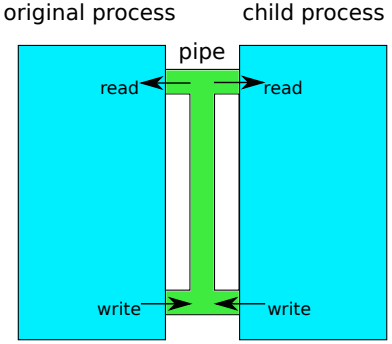
This will allow the parent and child processes to both have access to both ends of the pipe.

Pipe Illustration

After `pipe()` call:



After later `fork()` call:



Unidirectional Communication

Pipes are primarily intended for **unidirectional** communication between exactly *two* processes.

Consider trying to use a pipe for bidirectional communication: if P1 and then P2 each write data intended for the other into the pipe, should P1 read from the pipe first, it would get the message it had written, which was intended for P2.

Pipes are virtually always used with *only a single process reading from the pipe*, because otherwise there would be no way to control which process received the next "message."

Unidirectional Communication (contd.)

Pipes are usually used with a *single writer*, but sometimes have multiple processes write into the pipe, e.g., several clients sending requests to a single server process.

In this case, the communication *protocol* may have to enable the server to distinguish which client each "message" originated from (e.g., by including client PID or "name" in a message *header*).

"Messages" must also be short enough to ensure they are written into the pipe with a single `write()`.

Sockets are generally a better choice for IPC in **client-server** systems, since they support separate connections for each client and bidirectional communication.

Standard Usage Pattern (contd.)

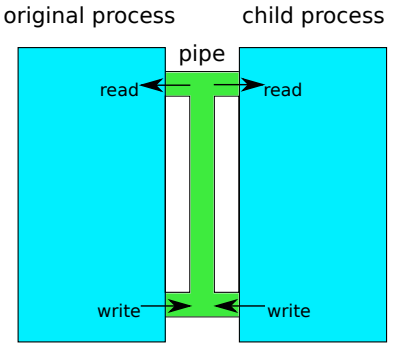
If pipes work best with one reader process and one writer process, how is this enforced after pipe FDs are inherited?

Generally, one pipe end is closed in each process, forming a unidirectional communication channel from one process to the other.

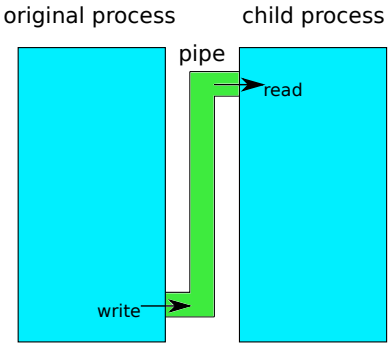
This also makes it possible for the reading process to determine when no more data will be coming—see below.

Pipe Illustration (contd.)

Two readers & writers can cause problems:



After closing pipe ends, have unidirectional channel:



pipe() Usage Example

Parent sends one message to child via pipe:

```
//Create pipe:  (must do before fork() so FDs inherited by child)
int pipefd[2];  //array to hold pipe FDs
pipe(pipefd);

//Create child and do its reading & printing functionality:
if (fork() == 0) {
  //In child:
  close(pipefd[1]);                //unused in child, close
  char buff[10];                   //storage for message
  read(pipefd[0],buff,10);         //read message from parent
  buff[11] = '\0';                 //turn buff into a string
  printf("pipe read: %s\n",buff);  //print message to stdout
  exit(EXIT_SUCCESS); }            //terminate child

//In parent (after fork()), so do its writing functionality:
close(pipefd[0]);                  //unused in parent, close
write(pipefd[1],"testing #1",10);  //write message to child
wait(NULL);                        //wait for child to terminate, collect
```

IPC and Pipes 2: Usage Details

1. Introduction and Basic Usage
2. **Usage Details**
   - **close unused pipe ends**
   - **stream oriented communication**
   - **finite size**
   - **I/O atomicity**
3. FIFOs

Close Unused Pipe Ends

Because a pipe will be used for unidirectional communication, each process requires access to only *one end* of the pipe.

Standard practice is to have each process that gains access to a pipe *immediately close the unused pipe end* file descriptor.

This allows a process to determine if the process(es) on the other end of the pipe has finished sending or receiving data:

- "If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read() from the pipe will see end-of-file (read() will return 0)."
- "If all file descriptors referring to the read end of a pipe have been closed, then a write() will cause a SIGPIPE signal to be generated for the calling process. If the calling process is ignoring this signal, then write() fails with the error EPIPE."

Close Unused Pipe Ends (contd.)

If the process *reading* from the pipe, itself has the write end of the pipe open, the process will fail to ever get an end-of-file return, even if all other processes have closed their write ends!

Being able to receive an end-of-file return when reading from a pipe will usually be required for proper program operation.

Failing to close unused pipe ends is a common cause of bugs when using pipes.

Consider the illustration situation, parent sending data to child: the child will know that no more data is forthcoming (and it should terminate) only when it receives an end-of-file return when trying to read from the pipe; this requires that *all* pipe write ends have been closed, including in the child process itself.

Close Unused Pipe Ends (contd.)

So, closing unused write ends is generally required for normal pipe usage.

On the other hand, *closing unused read ends* can be critical to detect *abnormal situations*.

Consider the illustration situation, parent sending data to child: should the child *unexpectedly/prematurely terminate*, the parent needs to know this so it doesn't go on writing data into the pipe.

The parent can find out the child has terminated by getting one of *two possible notifications* when it tries to write to the pipe.

In order to receive either of these notifications, *all read ends of the pipe must have been closed*, including in the writing (parent) process itself.

Close Unused Pipe Ends (contd.)

Unlike with end-of-file on `read()`, notification on `write()` requires additional code beyond simply closing unused pipe ends.

There are two possible "no pipe reader" notifcation methods when `write()` is called and all read ends of the pipe are closed:

1. a `SIGPIPE` signal is sent to the writer (the *default behavior*)
2. error return from `write()`, with `errno` set to `EPIPE`

By default, a process will receive a `SIGPIPE` signal if it writes to a pipe where all read ends have closed.

Since the **default disposition** action for `SIGPIPE` is *termination*, the writing process will *automatically terminate*.

Close Unused Pipe Ends (contd.)

It is *generally undesirable* to have a process terminate when it writes to a pipe just because the reading process(es) have all terminated prematurely/unexpectedly.

This means that we will generally have to include additional code in the writing process to set up `SIGPIPE` handling as desired.

In order to see the error return from `write()`, the process must have *explicitly* set `SIGPIPE` to be **ignored**.

This can be done with the call: `signal(SIGPIPE,SIG_IGN)`

An alternative is to have the writing process set `SIGPIPE` to be **caught**, and a **handler** function run instead.

pipe() Example using Closed Pipe Ends

Parent sends *unknown number* of messages to child via pipe:

```
  //Create pipe:
  int pipefd[2];
  pipe(pipefd);

  //Create child and do its reading & printing functionality:
  if (fork() == 0) {
    //In child:
    close(pipefd[1]);
    char buff[11];
    buff[10] = '\0';
    //Keep reading and printing messages until get end-of-file:
    while (read(pipefd[0],buff,10) != 0)
      printf("pipe read: %s\n",buff);
    exit(EXIT_SUCCESS);
  }

//...continued on next slide...
```


pipe() Example using Closed Pipe Ends (contd.)

```
//...continued from previous slide...

  //In parent (after fork()):
  close(pipefd[0]);
  //Set up SIGPIPE to be ignored, so write() will fail
  //if child prematurely terminates and so closes read end:
  signal(SIGPIPE,SIG_IGN);

  //Write N (<10) messages to the child:
  char msg[11];
  for (int i=1; i<=N; i++) {
    snprintf(msg,11,"testing #%1d",i);  //create message string
    //Write message but check if fails:
    if (write(pipefd[1],msg,10) < 0)
      //Child must have terminated prematurely, stop writing messages:
      break;
  }

  //Close pipe write end so child knows no more messages (gets end-of-file):
  close(pipefd[1]);
  wait(NULL);  //must still collect child
```

Stream Oriented Communication

Pipes support **stream-oriented** communication: bytes are read out of a pipe in the same order they were written into the pipe (i.e., **FIFO**).

It is not possible to determine how these bytes might have been written in groups, nor which process wrote them (no notion of "**messages**" or "sender").

If pipes are to be used to send "messages" then some **protocol** must be used to distinguish message boundaries:

- place a **delimiter** at the end of each message (e.g., `'\n'`)
- have each message include a fixed-size **header** containing the message *length*
- have all messages be the same size (possibly padded with spaces, null chars, etc.)

Finite Size, Blocking I/O

Since a pipe is implemented as a *kernel buffer*, it will have a *finite size*.

Linux pipes were once 4096 bytes, but now are 65536 bytes.

The pipe man page says this about pipe I/O and capacity:

- "Applications should not rely on a particular capacity: an application should be designed so that a reading process consumes data as soon as it is available."
- "If a process attempts to read from an empty pipe, then read() will **block** until data is available."
- "If a process attempts to write to a full pipe, then write() **blocks** until sufficient data has been read from the pipe to allow the write to complete."
- As with regular files, pipe I/O can be made **non-blocking**.

I/O Atomicity

Since pipes are going to be used in **multiprocess**, **concurrent** programs, the **atomicity** of I/O operations on pipes is important to understand.

The pipe man page says this about pipe I/O atomicity:

- "POSIX says that writes of less than PIPE_BUF bytes must be atomic."
- "Writes of more than PIPE_BUF bytes may be non-atomic: the kernel may interleave the data with data written by other processes."
- "POSIX requires PIPE_BUF to be at least 512 bytes."
- "On Linux, PIPE_BUF is 4096 bytes."

I/O Atomicity (contd.)

Consider requests to write n bytes, with f bytes of free space in the pipe buffer.

With `write()` we then have have these cases:

	$f \geq n$	$f < n$
$n \leq$ `PIPE_BUF`	immediate atomic write of n bytes	blocks until n bytes free then atomic write
$n >$ `PIPE_BUF`	write of n bytes but may not be atomic	blocks until n bytes written, but not atomic

Notice that *partial writes are not possible* with default, *blocking I/O* (they can, however, occur if **non-blocking I/O** is enabled).

I/O Atomicity (contd.)

Consider requests to read n bytes, with p bytes in pipe buffer.

With `read()` we then have have these cases:

$p = 0$	$p < n$	$p \geq n$
block until $p > 0$ then read $min(n, p)$ bytes	immediately read p bytes	immediately read n bytes

Notice that *partial reads can occur*.

IPC and Pipes 3: FIFOs

1. Introduction and Basic Usage
2. Usage Details
3. **FIFOs**
   - **FIFOs (named pipes)**
   - **mkfifo() call and mkfifo command**
   - **FIFO usage**
   - **popen() and pclose()**

FIFOs (Named Pipes)

One of the limitations of basic, unnamed pipes is that they can be used only among *related* processes.

A **FIFO** or **named pipe** is a pipe that has an *entry in the filesystem*.

This allows the FIFO to be opened and used by *unrelated processes*.

As a result, FIFOs are a better choice than basic pipes for clients to communicate with a server in a *client-server application*.

However, FIFOs are still *unidirectional*, so they are a reasonable choice for client-server architectures only if clients send simple requests to a server without requiring any responses.

mkfifo Call

The call to create a FIFO is the library function `mkfifo`:
`int mkfifo(const char *pathname, mode_t mode)`

- `pathname` is the filesystem path for the FIFO
- `mode` are the permissions for the entry
- returns 0 or -1 on error

The actual syscall that gets invoked to create a FIFO is `mknod`:
`int mknod(const char *pathname, mode_t mode, dev_t dev)`

Direct calls to `mknod` to create FIFOs are discouraged.

FIFOs may also be manually created with the `mkfifo` *command*:
`mkfifo [OPTION] NAME...`

FIFO Usage

Once a FIFO exists (in the filesystem) a process gains access to it by calling `open()` (a FIFO is a type of **special file**).

Typically, a process will either want to read from the FIFO or write to it but not both, and one process will do one and another process do the opposite.

Thus, each process will open the file either `O_RDONLY` or `O_WRONLY`.

Since a FIFO will initially be opened only for reading or for writing (unlike a plain pipe), the semantics of `open()` for FIFOs is that `open()` will *block* until the same FIFO is opened by a different process for the opposite purpose.

FIFO Example

Process #1:

```
//Create FIFO:
mkfifo("/tmp/myfifo",0600);

//Open FIFO to read from:
int fifo_in = open("/tmp/myfifo",O_RDONLY);

char buff[11];
int nread = read(fifo_in,buff,10);
buff[nread] = '\0';  //Turn buff into a string
printf("FIFO read: %s\n",buff);

close(fifo_in);
```

Process #2:

```
//Open FIFO to write to:
int fifo_out = open("/tmp/myfifo",O_WRONLY);

write(fifo_out,"testing",7);

close(fifo_out);
```


popen() and pclose()

C provides a couple of special functions that allow one to simplify running another program in a subprocess with a pipe to the parent:

```
FILE *popen(const char *command, const char *type)
int pclose(FILE *stream)
```

- `command` is a command line one could give the shell
- `type` is `"r"` or `"w"`, depending on whether the parent wants to read from the pipe or write to it

`popen()` creates a pipe, forks off a subprocess, closes unused pipe ends, and exec's the shell in it, passing `command` to the shell.

`pclose()` waits for the subprocess to terminate and returns the exit status of the command.

popen() and pclose() (contd.)

Example executing `mychild` program using `popen()`:

```
//Start mychild program with pipe to send info to it:
FILE *progfp = popen("mychild norman 110","w");
...
fprintf(progfp,"info to send");
...
pclose(progfp);
```


popen() and pclose() (contd.)

Example executing `mychild` program using syscalls:

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

switch(fork()) {
  case -1:
    perror("fork failed");
    exit(EXIT_FAILURE);
  case 0:
    close(pipefd[1]);
    execlp(mychild,"mychild","norman","110",(char*)NULL);
    perror("execlp failed")
    exit(EXIT_FAILURE);
  default:
    close(pipefd[0]);
    ...
    write(pipefd[1],"info to send",12);
    ...
    wait(NULL);
  }
```