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*.