Pipes, Part 2: Pipe programming secrets

Edit New Page

jakebailey edited this page on Dec 16, 2014 · 7 revisions

Pipe Gotchas (1)

Here's a complete example that doesn't work! The child reads one byte at a time from the pipe and prints it out - but we never see the message! Can you see why?

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

int main() {
    int fd[2];
    pipe(fd);
    //You must read from fd[0] and write from fd[1]
    printf("Reading from %d, writing to %d\n", fd[0], fd[1]);

    pid_t p = fork();
    if (p > 0) {
        /* I have a child therefore I am the parent*/
        write(fd[1],"Hi Child!",9);

        /*don't forget your child*/
        wait(NULL);
    } else {
        char buf;
        int bytesread;
        // read one byte at a time.
        while ((bytesread = read(fd[0], &buf, 1)) > 0) {
            putchar(buf);
        }
    }
    return 0;
}
```

The parent sends the bytes `H,i,(space),C...!` into the pipe (this may block if the pipe is full). The child starts reading the pipe one byte at a time. In the above case, the child process will read and print each character. However it never leaves the while loop! When there are no characters left to read it simply blocks and waits for more. The call `putchar` writes the characters out but we never flush the buffer.

To see the message we could flush the buffer (e.g. fflush(stdout) or printf("\n")) or better, let's look for the end of message '!'

```
        while ((bytesread = read(fd[0], &buf, 1)) > 0) {
            putchar(buf);
            if (buf == '!') break; /* End of message */
        }
```

And the message will be flushed to the terminal when the child process exits.

Want to use pipes with printf and scanf? Use fdopen!

POSIX file descriptors are simple integers 0,1,2,3... At the C library level, C wraps these with a buffer and useful functions like printf and scanf, so we that we can easily print or parse integers, strings etc. If you already have a file descriptor then you can 'wrap' it yourself into a FILE pointer using `fdopen` :

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

int main() {
    char *name="Fred";
    int score = 123;
    int filedes = open("mydata.txt", "w", O_CREAT, S_IWUSR | S_IRUSR);

    FILE *f = fdopen(filedes, "w");
    fprintf(f, "Name:%s Score:%d\n", name, score);
    fclose(f);
```

For writing to files this is unnecessary - just use `fopen` which does the same as `open` and `fdopen` However for pipes, we already have a file descriptor - so this is great time to use `fdopen` !

Here's a complete example using pipes that almost works! Can you spot the error? Hint: The parent never prints anything!

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

int main() {
    int fh[2];
    pipe(fh);
    FILE *reader = fdopen(fh[0], "r");
    FILE *writer = fdopen(fh[1], "w");
    pid_t p = fork();
    if (p > 0) {
        int score;
        fscanf(reader, "Score %d", &score);
        printf("The child says the score is %d\n", score);
    } else {
        fprintf(writer, "Score %d", 10 + 10);
        fflush(writer);
    }
    return 0;
}
```

Note the (unnamed) pipe resource will disappear once both the child and parent have exited. In the above example the child will send the bytes and the parent will receive the bytes from the pipe. However, no end-of-line character is ever sent, so `fscanf` will

continue to ask for bytes because it is waiting for the end of the line i.e. it will wait forever! The fix is to ensure we send a newline character, so that `fscanf` will return.

```
change:   fprintf(writer, "Score %d", 10 + 10);
to:       fprintf(writer, "Score %d\n", 10 + 10);
```

So do we need to `fflush` too? Yes, if you want your bytes to be sent to the pipe immediately! At the beginning of this course we assumed that file streams are always *line buffered* i.e. the C library will flush its buffer everytime you send a newline character. Actually this is only true for terminal streams - for other filestreams the C library attempts to improve performance by only flushing when it's internal buffer is full or the file is closed.

When do I need two pipes?

If you need to send data to and from a child asynchronously, then two pipes are required (one for each direction). Otherwise the child would attempt to read its own data intended for the parent (and vice versa)!

Closing pipes gotchas

Processes receive the signal SIGPIPE when no process is listening! From the pipe(2) man page -

```
If all file descriptors referring to the read end of a pipe have been closed,
 then a write(2) will cause a SIGPIPE signal to be generated for the calling proc
```

Tip: Notice only the writer (not a reader) can use this signal. To inform the reader that a writer is closing their end of the pipe, you could write your own special byte (e.g. 0xff) or a message (`"Bye!"`)

Here's an example of catching this signal that does not work! Can you see why?

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

void no_one_listening(int signal) {
    write(1, "No one is listening!\n", 21);
}

int main() {
    signal(SIGPIPE, no_one_listening);
    int filedes[2];

    pipe(filedes);
    pid_t child = fork();
    if (child > 0) {
        /* I must be the parent. Close the listening end of the pipe */
        /* I'm not listening anymore!*/
        close(filedes[0]);
    } else {
```

```
        /* Child writes messages to the pipe */
        write(filedes[1], "One", 3);
        sleep(2);
        // Will this write generate SIGPIPE ?
        write(filedes[1], "Two", 3);
        write(1, "Done\n", 5);
    }
    return 0;
}
```

The mistake in above code is that there is still a reader for the pipe! The child still has the pipe's first file descriptor open and remember the specification? All readers must be closed.

When forking, *It is common practice* to close the unnecessary (unused) end of each pipe in the child and parent process. For example the parent might close the reading end and the child might close the writing end (and vice versa if you have two pipes)

The lifetime of pipes

Unnamed pipes (the kind we've seen up to this point) live in memory (do not take up any disk space) and are a simple and efficient form of inter-process communication (IPC) that is useful for streaming data and simple messages. Once all processes have closed, the pipe resources are freed.

An alternative to *unamed* pipes is *named* pipes created using `mkfifo` - more about these in a future lecture.