Assignment 2: Pipes

Recitation for ECES 338
February 8, 1999

Unnamed Pipes

- `int pipe (int filedes[2])`
  - This function creates two file descriptors, stored in an array. These descriptors are connected such that writes to the first one will be readable by the second descriptor, and vice-versa. This function is typically used in a `fork()` environment, as that is the only way to pass the open file descriptors between processes.

    Example:

```
int pdes[2];

if (pipe (pdes) == -1)
        {
        fprintf (stderr, "Error creating pipe.\n");
        perror ("Pipe Create");
        exit (1);
        }
```

- `ssize_t read (int filedes, void *message, size_t num_bytes)`
  - This function (documented in section two of the manual) reads up to `num_bytes` from the given file descriptor, and places the data into the given buffer. It then returns the number of bytes that it read. If you wish to use variable-length messages, you can use one of two methods. The first is to just bound the size of your messages, and rely on `read()` to quit reading when it is done. Alternatively, you could develop some sort of encoding scheme whereby you first `read()` a number, which tells you the size of the following message.

    Example:

```
read (pdes[0], consumed_msg, BUFSIZE)
```

- `ssize_t write (int filedes, const void *message, size_t num_bytes)`
  - This function (documented in section two of the manual) is similar to `read()`, in that it will write `num_bytes` from the given buffer into the given file descriptor. If said file descriptor is busy, or cannot hold the requisite amount of data, `write()` will block. It will return the number of bytes actually written.

    Example:

```
write (pdes[1], prod_msg, strlen (prod_msg) + 1)
```

- `close (int filedes)`
  - This function simply closes a file descriptor, flushing all of the buffered data pending.

Named Pipes

- `int mknod (const char *path, mode_t mode, dev_t dev)`
  - The `path` should be a fully qualified filename on the filesystem to which you have write privileges. For named pipes, the `mode` should be `S_IFIFO` bitwise-or'd with the filesystem permissions that you wish for your pipe to have. The `dev` parameter should be left zero.

- `int mkfifo (const char *path, mode_t mode)`
  - Identical to `mknode`, except that the `mode` only needs to specify the filesystem parameters.

- `int open (const char *path, int oflag)`
  - This opens a file, which in this case will be a pipe created by either `mknod()` or `mkfifo()`. The flag parameter specifies if the stream is to be opened for reading, writing, or both operations.

- `int unlink (const char *path)`
  - This function removes a given filename from the filesystem, similar to `rm`.

Miscellaneous Functions

- `strlen` – calculates the length of a given string, be sure to add one in order to account for the string terminator.

- `sprintf` – is useful for "building" strings, the `printf` way. Basically, instead of printing to `stdout`, the output will be written to a buffer, given as the first parameter.

- `memset` – a really efficient way to wipe out a given area of memory with some new constant (usually zero).

Miscellaneous Topics

- The monkey book has a lot of paranoia about deadlock, and the `O_NONBLOCK` and `O_NDELAY` flags. Basically, you can code yourself into situations whereby all (or some) of your processes deadlock, so that they are all waiting for some event, which will never occur. For example, the default behavior of the `write()` system call is to block (sleep) if the file descriptor is busy. Thus, if you have two processes attempting to write to the same file descriptor, neither will ever awake. I don't think that you'll have to worry much about this, for this assignment.

- To implement fully, I believe that part one of the latest assignment is going to be the hardest. I'd suggest that you use relatively simple messages (such as one word), as

that will simplify the file `I/O`. It will also be much easier if you create a `consumer()` and a `producer()` function, and pass them values so that they know that they're `C1` or `P2`, for example. Also, be sure to call `srand()` once in the parent before you call `rand()`.

- Part 4 is rather ambiguous. Tekin writes that you should have your parent `execl()` into an `ls`, but I think that you might want to do it in a child process, so that you can show multiple directory listings over the course of your run.

- Some hints for this assignment (since I haven't gotten assignment graded…):
  - Be sure to make your code readable. Use a word processor if you have to. Also, modularity (functions) is going to be a **must** for this assignment.
  - Be sure to comment your code, especially the variables that you use. I also find it handy to comment the closing "curly braces", in order to keep track of them.
  - Be sure to include details about your compilation environment.
  - Be sure to include your output, as readable and detailed as you can make it.
  - Be sure to do error checking! Even though it is optional, it can make up for lost points if the grader doesn't like what you did elsewhere in your assignment. It's as simple as wrapping system calls up in an `if` statement, and using the `perror()` and/or `fprintf (stderr, …)` functions.
  - Finally, be sure to check all of the homepages associated with this course! Also, if you are at all confused about something, feel free to e-mail your recitation leader, or come to our office hours. That's what we're here for!