## 1 UNIX Pipes

We will start off this guide by talking about UNIX pipes. Pipes are used by the operating system in order to transfer data from from one process to another during execution. We can imagine that a pipe acts like a file, where one process can read from the file, and the other process can write to the file. Pipes tend to follow a FIFO structure. Here are some common features of a UNIX pipe:

1. Pipes are limited to 10 logical blocks, with each block containing 512 bytes of data.
2. Generally, one process will read from the pipe while the other will write.
3. Data is written to one end of the pipe by a process and read from the other end of the pipe by another process.
4. There is no file pointer associated with a pipe, as there is with a normal file in UNIX.

Pipes are generally created in the kernel of the OS, and generally each process will have it's own pipe that it writes to and reads from. In other words, if *Process A* is reading from a pipe that *Process B* is writing to, the will also be a pipe to which *Process A* can write and *Process B* can read. Let's take a look at the system call to write to a file:

```
size_t write(int file_des, const void *buffer, size_t bytes)
```

This function will write *bytes* amount of *buffer* to the file descriptor given by *file_des*. It returns -1 if the call fails, or the number of bytes written otherwise. Similarly we have the *read* system call:

```
size_t read(int file_des, const void *buffer, size_t  num_bytes)
```

The *read* function will number of bytes read if successful, or -1 otherwise. We should note that we actually have two types of pipes, unnamed and named. From a programatic standpoint, we see no difference between the file descriptors for the two pipes besides the call used to make them. Unnamed pipes are used between related processes, such as parent/child, whereas named pipes are created with a directory entry and read/write permissions. Unnamed and named pipes are made with the *pipe* and *mknod* system calls, respectively. Let's take a look at the *pipe* system call.

```
int pipe (int file_des[2])
```

This system call takes in an array of ints that, upon success, will contain the file descriptors for reading and writing. It will also return 0 upon success, and -1 upon failure.

You can image that pipes could be used for I/O between processes. We have two system calls that we can use to accomplish this feat:

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