## Supplement to Chapter 3

## Creating Pipes in C

* Creating ``pipelines'' with the C programming language can be a bit more involved than our simple shell example. To create a simple pipe with C, we make use of the pipe() system call. It takes a single argument, which is an array of two integers, and if successful, the array will contain two new file descriptors to be used for the pipeline. After creating a pipe, the process typically spawns a new process (remember the child inherits open file descriptors).
* SYSTEM CALL: pipe();
* PROTOTYPE: int pipe( int fd[2] );
* RETURNS: 0 on success
* -1 on error: errno = EMFILE (no free descriptors)
* EMFILE (system file table is full)
* EFAULT (fd array is not valid)
* NOTES: fd[0] is set up for reading, fd[1] is set up for writing
* The first integer in the array (element 0) is set up and opened for reading, while the second integer (element 1) is set up and opened for writing. Visually speaking, the output of fd1 becomes the input for fd0. Once again, all data traveling through the pipe moves through the kernel.
* #include <stdio.h>
* #include <unistd.h>
* #include <sys/types.h>
* main()
* {
* int fd[2];
* pipe(fd);
* .
* .
* }
* Remember that an array name in C decays into a pointer to its first member. Above, fd is equivalent to &fd[0]. Once we have established the pipeline, we then fork our new child process:
* #include <stdio.h>
* #include <unistd.h>
* #include <sys/types.h>
* main()
* {
* int fd[2];
* pid\_t childpid;
* pipe(fd);
* if((childpid = fork()) == -1)
* {
* perror("fork");
* exit(1);
* }
* .
* .
* }
* If the parent wants to receive data from the child, it should close fd1, and the child should close fd0. If the parent wants to send data to the child, it should close fd0, and the child should close fd1. Since descriptors are shared between the parent and child, we should always be sure to close the end of pipe we aren't concerned with. On a technical note, the EOF will never be returned if the unnecessary ends of the pipe are not explicitly closed.
* #include <stdio.h>
* #include <unistd.h>
* #include <sys/types.h>
* main()
* {
* int fd[2];
* pid\_t childpid;
* pipe(fd);
* if((childpid = fork()) == -1)
* {
* perror("fork");
* exit(1);
* }
* if(childpid == 0)
* {
* /\* Child process closes up input side of pipe \*/
* close(fd[0]);
* }
* else
* {
* /\* Parent process closes up output side of pipe \*/
* close(fd[1]);
* }
* .
* .
* }
* As mentioned previously, once the pipeline has been established, the file descriptors may be treated like descriptors to normal files.
* /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
* Excerpt from "Linux Programmer's Guide - Chapter 6"
* (C)opyright 1994-1995, Scott Burkett
* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
* MODULE: pipe.c
* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
* #include <stdio.h>
* #include <unistd.h>
* #include <sys/types.h>
* int main(void)
* {
* int fd[2], nbytes;
* pid\_t childpid;
* char string[] = "Hello, world!\n";
* char readbuffer[80];
* pipe(fd);
* if((childpid = fork()) == -1)
* {
* perror("fork");
* exit(1);
* }
* if(childpid == 0)
* {
* /\* Child process closes up input side of pipe \*/
* close(fd[0]);
* /\* Send "string" through the output side of pipe \*/
* write(fd[1], string, (strlen(string)+1));
* exit(0);
* }
* else
* {
* /\* Parent process closes up output side of pipe \*/
* close(fd[1]);
* /\* Read in a string from the pipe \*/
* nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
* printf("Received string: %s", readbuffer);
* }
* return(0);
* }