**Cheat Sheet:**

Exam 1 Part 2

**Fork()**

fork() in C:

Fork system call use for creates a new process, which is called ***child process***, which runs concurrently with process (which process called system call fork) and this process is called ***parent process***. After a new child process created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

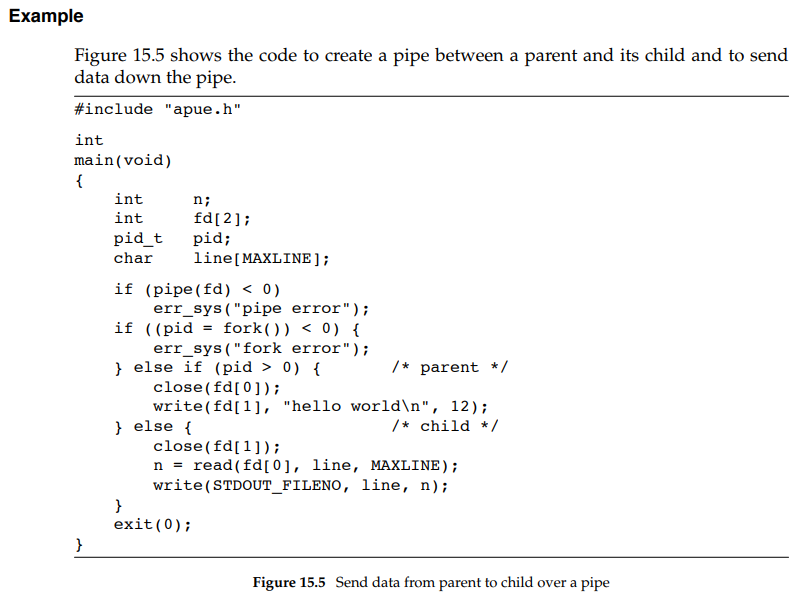
It takes no parameters and returns an integer value. Below are different values returned by fork().

***Negative Value***: creation of a child process was unsuccessful.  
***Zero***: Returned to the newly created child process.  
***Positive value***: Returned to parent or caller. The value contains process ID of newly created child process

**Pipe()**

When one end of a pipe is closed, two rules apply:

1. If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
2. If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns −1 with errno set to EPIPE.



In the previous example, we called read and write directly on the pipe descriptors. What is more interesting is to duplicate the pipe descriptors onto standard input or standard output. Often, the child then runs some other program, and that program can either read from its standard input (the pipe that we created) or write to its standard output (the pipe). In the previous example, we called read and write directly on the pipe descriptors. What is more interesting is to duplicate the pipe descriptors onto standard input or standard output. Often, the child then runs some other program, and that program can either read from its standard input (the pipe that we created) or write to its standard output (the pipe).

Consider a program that displays some output that it has created, one page at a time. Rather than reinvent the pagination done by several UNIX system utilities, we want to invoke the user’s favorite pager. To avoid writing all the data to a temporary file and calling system to display that file, we want to pipe the output directly to the pager. To do this, we create a pipe, fork a child process, set up the child’s standard input to be the read end of the pipe, and exec the user’s pager program. Figure 15.6 shows how to do this

Before calling fork, we create a pipe. After the fork, the parent closes its read end, and the child closes its write end. The child then calls dup2 to have its standard input be the read end of the pipe. When the pager program is executed, its standard input will be the read end of the pipe. When we duplicate one descriptor onto another (fd[0] onto standard input in the child), we have to be careful that the descriptor doesn’t already have the desired value. If the descriptor already had the desired value and we called dup2 and close, the single copy of the descriptor would be closed. (Recall the operation of dup2 when its two arguments are equal, discussed in Section 3.12.) In this program, if standard input had not been opened by the shell, the fopen at the beginning of the program should have used descriptor 0, the lowest unused descriptor, so fd[0] should never equal standard input. Nevertheless, whenever we call dup2 and close to duplicate one descriptor onto another, we’ll always compare the descriptors first, as a defensive programming measure. Note how we try to use the environment variable PAGER to obtain the name of the user ’s pager program. If this doesn’t work, we use a default. This is a common usage of environment variables

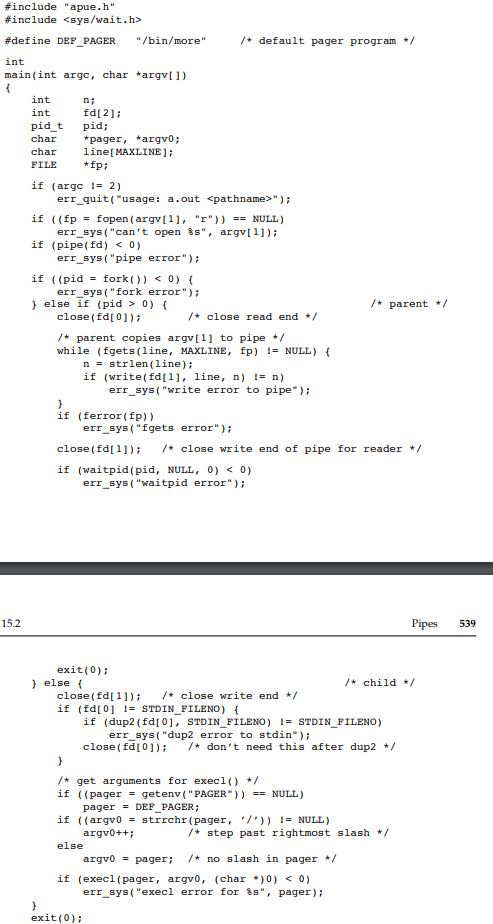
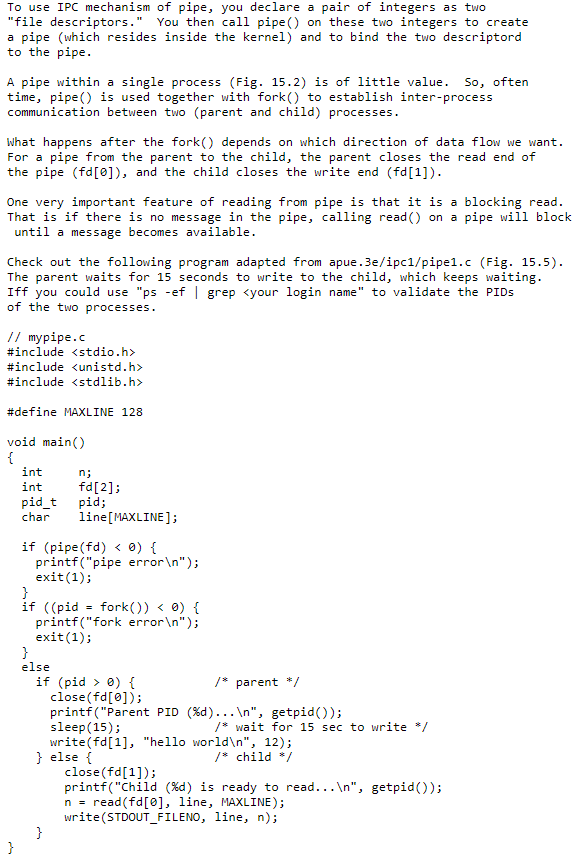
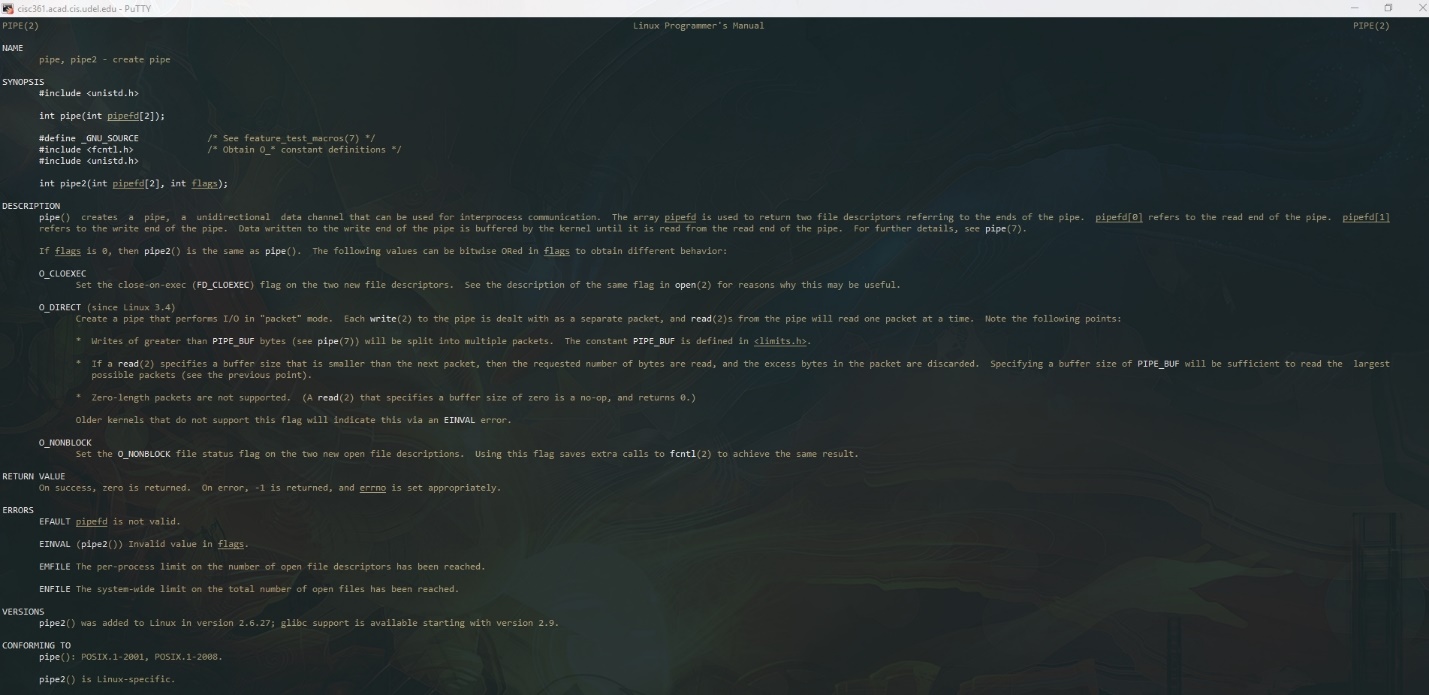
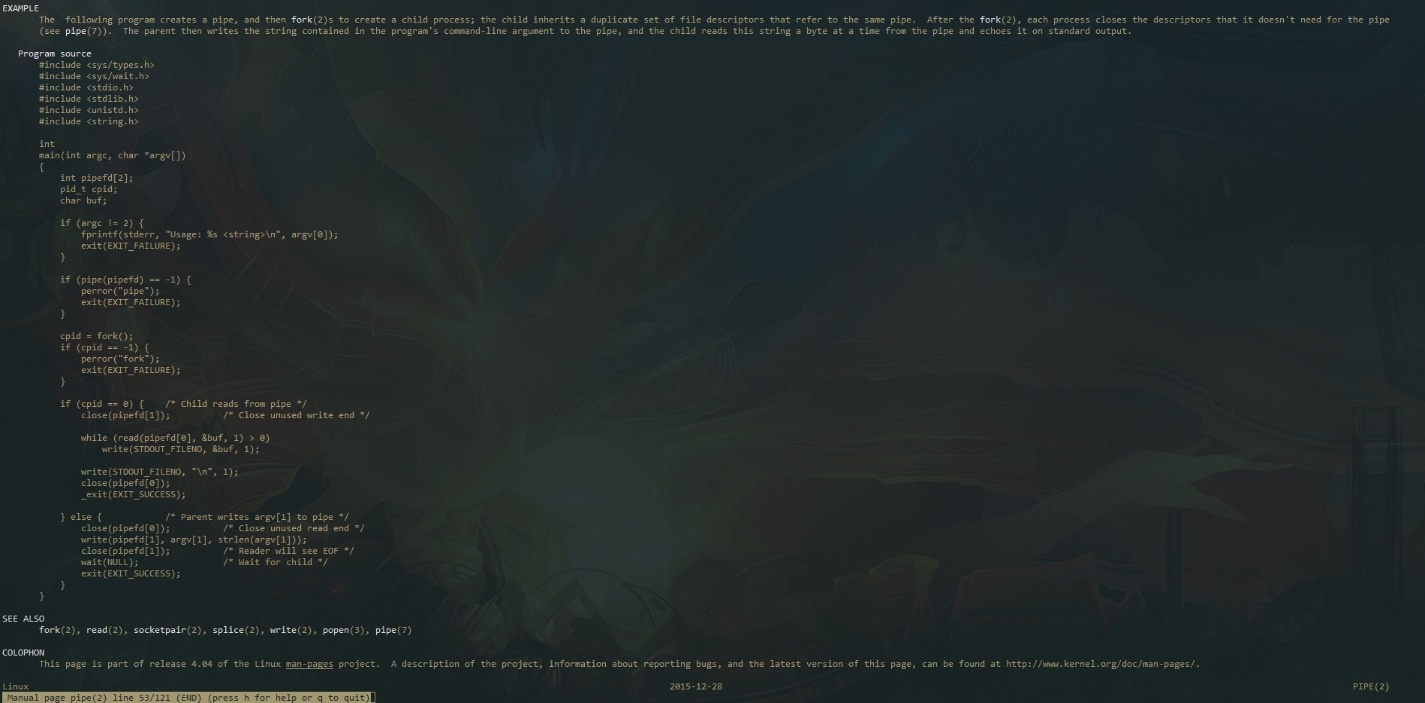


Fig 15.6

****

**Man Pages For Pipe()**

****