भारतीय सूचना प्रौद्योगिकी संस्थान गुवाहाटी
Indian Institute of Information Technology Guwahati

OPERATING SYSTEM LAB (CS232)
ASSIGNMENTS–01

1 Linux System call Basics

These routines allow access to the Linux system at a lower, perhaps more efficient, level than the standard library, which is usually implemented in terms of what follows.

1.1 File Descriptor

Small positive integers are used to identify open files. Three file descriptors are automatically set up when a program is executed:
0 - stdin
1 - stdout
2 - stderr
which are usually connected with the terminal unless redirected by the shell or changed in the program with close and dup.

1.2 Low Level I/O

Opening files:
int fd;
fd = open(name, rwmode);
where name is the character string file name and rwmode is 0, 1, 2 for read, write, read and write respectively. Better to use $'\#include < fcntl.h >'$ and defined constants like O_RDONLY. A negative result is returned in fd if there is an error such as trying to open a nonexistent file. **Creating Files:**

fd = creat(name, pmode);

which creates a new file or truncates an existing file and where the octal constant pmode specifies the chmod-like 9-bit protection mode for a new file. Again a negative returned value represents an error.
Closing files:

close(fd);

Removing files:
unlink(filename);
Reading and writing files:
n_read = read(fd, buf, n);
n_written = write(fd, buf, n);
will try to read or write n bytes from or to an opened or created file and return the number of bytes read or written in n_read, for example
$\#define\ BUFSIZ$ 1024
main () {
$char\ buf[BUFSIZ];$
int n;
$while((n = read(0, buf, BUFSIZ)) > 0) write(1, buf, n);$
}

If read returns a count of 0 bytes read, then EOF has been reached. Changing File Descriptors If you have a file descriptor fd that you want to make stdin refer to, then the following two steps will do it: close(0); dup(fd);
— close stdin and free slot 0 in open file table
— dup makes a copy of fd in the smallest unused slot in the open file table, which is now 0

Assignments

1. Write a C program that prints out all of its command-line arguments except those that begin with a dash.
   For example, after compiling your program into the file a.out, entering the command
   a.out arg1 -arg2 arg3
   should print
   arg1 arg3
   including a newline so your next shell prompt is not right after arg3. If all arguments begin with a dash, (or if there are no arguments at all) print nothing, that is, do not even print a newline. This is so that a blank line does not appear between the a.out command line and your next shell prompt.

2. You are to write another C program which will read characters from its standard input stdin, count the number of NON-alphabetic ones including newlines (see the file /usr/include/ctype.h), that is, count those characters not in the a-z range nor in the A-Z range, and write out all characters read. To read and write characters, this program uses only stdin and stdout, and only the stdio library routines (see stdio.h) for input and output (see getchar and putchar). When it hits EOF in its input, the

program will print out the final non-alphabetic count on stderr using fprintf and then exit(0).

3. You are to write yet another C program that reads characters from stdin, reverses lowercase and uppercase letters, that is, converts to lowercase all uppercase letters and vice versa (again, see ctype.h), and writes the results onto its standard output, stdout. All characters read should be written whether converted or not. This program uses only stdin and stdout, and the stdio library routines. The program will exit(0) when it hits EOF in its input.

4. Write C programs to know the use of following system calls 1. open()
   2. creat()
   3. dup()
   4. dup2()
   5. pipe()
   6. read()
   7. write()

5. You are to combine the C programs from 1, 2 and 3 that: (1) reads from the first file argument, reverses lowercase and uppercase letters, that is, makes lowercase all uppercase letters and vice versa, and writes out all characters, whether or not reversed; and (2) reads the above output, counts the number of NON-alphabetic characters (that is, those not in the a-z range nor in the A-Z range), and writes out all characters, whether counted or not, into its second file argument.
   So that you become familiar with the various LINUX system calls and libraries (fork, execl, pipe, creat, open, close, dup, stdio, exit), you will write this program in a certain contrived fashion. The program will be invoked as "driver file1 file2". Both file arguments are required (print an error message on stderr using fprintf if either argument is missing and then exit(1)). The driver program will open the first file and creat the second file, and dup them down to stdin and stdout. The driver program sets up a pipe and then forks two children.
   The first child forked will dup the read end of the pipe down to stdin and then execl the program you have written, count, which will read characters from stdin, count the non-alphabetic ones (see /usr/include/ctype.h), and write all characters out to stdout. Thecount program uses only stdin and stdout, and only the stdio library routines (see stdio.h) for input and output (see getchar and putchar). The program will print out the final count on stderr using fprintf and then exit(0) when it hits EOF in its input.
   The second child forked will dup the write end of the pipe down to stdout and then execl the program you have written, convert, which will read characters from stdin, reverse uppercase and lowercase (again, see ctype.h), and write them all out to stdout. Like count, this program uses only stdin and stdout. Also, it uses only the

stdio library routines. The program will exit(0) when it hits EOF in its input. The reason for creating the children in this order (first the one that reads from the pipe, usually called the second in the pipeline, and then second the one that writes to the pipe, usually called the first in the pipeline – confusing!!) is that LINUX will not let a process write to a pipe if no process has the pipe open for reading; the process trying to write would get the SIGPIPE signal. So we create first the process that reads from the pipe to avoid that possibility. Meanwhile, the parent process, the driver, will close its pipe file descriptors and then call wait (see /usr/include/sys/wait.h) twice to reap the zombie children processes when they finish. Then the parent will exit(0).

Remember to close all unneeded file descriptors, including unneeded pipe ones, or EOF on a pipe read may not be detected correctly. Check all system calls for the error return (-1). For all error and debug messages, use "fprintf(stderr, ...)" or use the function perror or the external variable errno (see also /usr/include/errno.h). Declare a function to be of type void if it does not return a value, that is, if it is a subroutine rather than a true function. If you absolutely must ignore the output of a system call or function, then cast the call to void. ———