**CS232 Operating Systems Lab**

**Winter 2021**

**Assignment 1**

**Linux System Calls**

**DUE DATE: January 13 2020**

**Assignment on Linux System Calls. File I/O, fork, signals etc.**

**There are five parts to the assignment: A to E. All have to be done.**

*(A tutorial on basic Linux programming is also included with this assignment.)*

**Submission Details:**

1. Create a directory named lab1 in your home directory (use only lower case everywhere)
2. Create executables named a, b, c, and d for the programs described under A, B, C, and D below respectively. ( so, if your program A is in a.c then you do gcc –o a a.c)

**(A) You are to write a C program that prints out all of its command-line arguments** except those that begin with a dash. Hint: find out about argc and argv. For example, after compiling your program into the file a.out, entering the command

a.out arg1 -arg2 arg3

prints

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.

**(B) 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).

**(C) 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.

----------------------------------------------------------------------

**Hints and Notes on Linux Programming**

1

*Subroutines and Functions*

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.

*Command-line arguments*

A C main program is actually called with two arguments: argc and argv.

argc is the number of command-line arguments the program is invoked

with including the program name, so argc is always at least one. argv

is a pointer to an array of character strings that contain the

arguments, one per string.

Here is a program that echoes its arguments to show how to use these:

main (int argc, char \*argv[]) /\* echo arguments \*/

/\* argv is not a string but an array of pointers \*/

/\* argv[0] is command-line program name \*/

{

int i;

for (i=1; i<argc; i++)

printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');

}

Or more cryptically:

main (int argc, char \*argv[]) /\* echo arguments \*/

{

while (--argc>0)

printf((argc>1) ? "%s " : "%s\n", \*++argv);

/\* argv[i] and \*(argv+i) are same \*/

}

Also argv[i][j] or (\*(argv+i))[j] is the jth character of the ith

command-line argument.

*Standard input and output*

Remember IO is not a part of the C language, but is provided in the

``standard IO library'' accessed by putting

#include <stdio.h>

at the beginning of your program or in your include files. This will

provide you with the functions getchar, putchar, printf, scanf and

symbolic constants like EOF and NULL.

*File access*

It is also possible to access files other than stdin and stdout

directly. First, make the declaration

FILE \*fopen(), \*fp;

declaring the open routine and the file pointer. Then, call the open

routine

fp = fopen(name, mode);

where name is a character string and mode is "r" for read, "w" for

write, or "a" for append.

Now the following routines can be used for IO:

c = getc(fp);

put(c, fp);

and also fprintf, fscanf which have a file argument.

The last thing to do is close the file (done automatically anyway at

program termination) (fflush just to flush but keep the file open):

fclose(fp);

The three file pointers stdin, stdout, stderr are predefined and can

be used anywhere fp above was. stderr is conventionally used for error

messages like wrong arguments to a command:

if ((fp = fopen(\*++argv, "r")) == NULL) {

fprintf(stderr, "cat: can't open %s\n", \*argv);

exit(1); /\* close files and abort \*/

} else ...

*Storage allocation*

To get some dynamically allocated storage, use

calloc(n, sizeof(object))

which returns space for n objects of the given size and should be cast

appropriately:

char \*calloc();

int \*ip;

...

ip = (int \*) calloc(n, sizeof(int));

Use free(p) to return the space.

***Linux system call interface***

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.

*File descriptors*

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.

*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 /\* already defined by stdio.h if included earlier \*/

main () /\* copy input to output \*/

{

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); /\* close stdin and free slot 0 in open file table \*/

dup(fd); /\* dup makes a copy of fd in the smallest unused

slot in the open file table, which is now 0 \*/

*Standard library*

Do a ``man stdio'' to find out the standard IO library.

Do a ``man printf'', ``man getchar'', and ``man putchar'' for even

more information.

Do a ``man atoi'' to find out ascii to integer conversion.

Do a ``man errno'' to find out printing error messages and the

external variable errno. Also do a ``man strerror''.

Do a ``man string'' to find out the string manipulation routines

available.

strcat /\* concatenation \*/

strncat

strcmp /\* compare \*/

strncmp

strcpy /\* copy \*/

strncpy

strlen /\* length \*/

index /\* find c in s \*/

rindex

Do a ``man 2 intro'' to find out about system calls like open, read,

dup, pipe, etc.

You can do a ``man 2 open'', ``man 2 dup'', etc. to find out more

about each one (``man 3 execl'', ``man 2 execve'', ``man 2 write'',

``man 2 fork'', ``man 2 close'', ``man 3 exit'', ``man 2 wait'',

``man 2 getpid'', ``man 2 alarm'', ``man 2 kill'', ``man 2 signal'').

/usr/include and /usr/include/sys contain .h files that can be

included in C programs with the ``#include <stdio.h>'' or ``#include

<sys/inode.h>'' statements, for example.

Other useful ones are ctype.h, errno.h, math.h, signal.h, string.h.

*Error messages*

This program shows how to use perror to print error messages when

system calls return an error status.

#include <stdio.h>

#include <sys/types.h> /\* fcntl.h needs this \*/

#include <fcntl.h> /\* for second argument of open \*/

#include <errno.h>

void exit(int); /\* gets rid of warning message \*/

main (int argc, char \*argv[]) {

if (open(argv[1], O\_RDONLY) < 0) {

fprintf(stderr, "errno=%d\n", errno);

perror("open error in main");

}

exit(0);

}

*grep and find commands*

To search all .c and .h files in the current directory for a string,

do this

grep "search string" \*.c \*.h

To search for all files with a certain string in their name in the

current directory and all its subdirectories, do this

find . -name "\*string\*" -print

To search all files in the current directory and all subdirectories

for a certain string, do this

find . -exec grep "string" {} /dev/null \;

The /dev/null is a LINUX guru trick to get grep to print out the file

name in addition to the line containing the matching string.

-----------------------------------------------------------------------

**(D) You are to combine the C programs from (A), (B) and (C) 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. The

count 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.

------------------------------------------------------------------------------------------------------

Hints and Notes

Program skeleton

o parent opens file1:

#include <sys/types.h>

#include <fcntl.h> /\* defines O\_RDONLY etc. \*/

...

fd\_in = open(argv[1], O\_RDONLY); /\* not fopen \*/

/\* Check fd\_in for -1!!! If it is, then the file does not

exist or you do not have read permission. \*/

o parent creat's file2:

fd\_out = creat(argv[2], 0644); /\* mode = permissions, here rw-r--r-- \*/

/\* Check fd\_out for -1!!! If it is, then the file could not

be created \*/

o parent uses close and dup so stdin is now file1 (done like

children below)

o parent uses close and dup so stdout is now file2: if dup returns

-1, then a problem has occurred

o parent calls pipe system call to create pipe file descriptors

o parent forks a child to read from pipe:

-- this first child manipulates file descriptors

use close, dup so stdin is read end of pipe

(see text Figure 1-13, similar to else clause)

-- this first child execs count

execl("count", "count", (char \*) 0)

execl overlays the child with the binary compiled program in

the file given by first argument and the process name becomes

the second argument

o parent forks again a child to write to pipe:

-- this second child manipulates file descriptors

use close, dup so stdout is write end of pipe

(similar to if clause in text Figure 1-13)

-- this second child execs convert

execl("convert", "convert", (char \*) 0)

o parent closes both ends of pipe

o parent waits twice (see text Figure 1-10)

-- use wait(&status); instead of waitpid(-1, &status, 0);

Before compiling and running driver, remember to:

o in count.c: fprintf(stderr, "final count = %d\n", count);

o cc -o convert convert.c, and

o cc -o count count.c

The two forks form nested if statements.

pipe(...

if (fork() != 0) { /\* parent continues here \*/

if (fork() != 0) { /\* parent continues here \*/

close(write end of pipe...

wait(...

wait(...

} else {

... /\* second child, writes to pipe \*/

}

} else {

... /\* first child, reads from pipe \*/

}

It is important to check all system calls (open, creat, dup, etc.) for

a return value <0, particularly -1, because such a return value

means an error has occurred. If you just let your program continue to

execute, it will just dump core at some later time and you will not

know why.

It is IMPERATIVE that the parent and the first child forked (the one

to read the pipe) close their write end file descriptors to the pipe.

If they do not do this, then the first child will never get EOF

reading from the pipe, even after the second child has sent

everything. This is because EOF on a pipe is not indicated until ALL

write file descriptors to the pipe are closed. If the parent and first

child fail to close their file descriptors for the write end of the

pipe, the program will hang forever (until killed). Furthermore, the

parent must close its copy of the file descriptor to the write end of

the pipe BEFORE the waits, not after, or deadlock will result. And the

first child forked (the one to read the pipe) must close its file

descriptor to the write end of the pipe before it starts reading the

pipe, not after, or deadlock will result.

How dup works

dup(fd) looks for the smallest unused slot in the process descriptor

table and makes that slot point to the same file, pipe, whatever, that

fd points to. For example, each process starts as

process descriptor table

------------------------

slot # points to

------ ---------

0 (stdin) keyboard

1 (stdout) screen

2 (stderr) screen

If the process does a

fd\_in = open("file1", ...

fd\_out = creat("file2", ...

then the table now looks like

process descriptor table

------------------------

slot # points to

------ ---------

0 keyboard

1 screen

2 screen

3 file1

4 file2

and fd\_in is 3 and fd\_out is 4. Now suppose the program does a

close(0);

dup(fd\_in);

close(fd\_in);

After the close(0), the table will look like

process descriptor table

------------------------

slot # points to

------ ---------

0 -- unused --

1 screen

2 screen

3 file1

4 file2

and after the dup(fd\_in), the table will look like

process descriptor table

------------------------

slot # points to

------ ---------

0 file1

1 screen

2 screen

3 file1

4 file2

and after the close(fd\_in), the table will look like

process descriptor table

------------------------

slot # points to

------ ---------

0 file1

1 screen

2 screen

3 -- unused --

4 file2

**(E) You are to modify part (D) of your C program.** This is to get you familiar with more of the various LINUX system calls and libraries (setjmp, longjmp, getpid,

alarm, read, write, kill, signal).

The program will be invoked as ``driver [-n] [file1] [file2]''. All

arguments are optional. If there is just one file, then the parent

opens it for reading, and dups it down to stdin. If there are two

files, then we have the same situation as part (D) above. If there are

no files, then the parent leaves its stdin and stdout alone. (The

other case could be handled at the command level by invoking the

driver as ``driver [-n] >file2''). If the -n argument is present,

where n is a positive decimal number, then the first child forked, the

one that is going to overlay itself with the count program, passes the

entire -n argument to the program count, which overlays it. The first

child does this in its execl statement. (It is as if the command

``count -n'' had been entered).

The parent sets up a pipe and forks two children, as in part (D).

Before going into its wait loop, waiting for the children to complete,

the parent (worried about children getting hung) sets up a signal

handler for the SIGALRM signal (see /usr/include/signal.h), using the

signal system call. The parent next saves its stack environment, using

the setjmp library routine (see /usr/include/setjmp.h). The parent

then sets an alarm to go off in 15 seconds, using the alarm system

call. Finally, the parent enters its wait loop, as in part (D). The

parent looks something like Figure 1.

If both children exit normally (exit(0)) before the alarm goes off,

then the parent prints a ``normal children exit'' message on stderr,

and then does an exit(0) itself. On the other hand, if the alarm goes

off, then the parent's signal handler for the arriving SIGALRM signal

will do a longjmp, causing the parent to kill both children, using the

kill system call, with the SIGTERM signal, and to print on stderr a

message like ``read timeout in second child, killing both children.''

The parent then waits again for both children and does an exit(1). The

final possibility for the parent to handle is if one of the children

terminates abnormally before the alarm goes off. (See below for the

case when the program count can exit(2).) If this happens, then the

parent should print an informative message on stderr, try to kill the

other child if it is still alive, wait again, and then exit(1) itself.

Each child, after the fork but before the execl, should print its pid

(process id) on stderr, using the getpid system call. Both programs

convert and count need to have signal handlers for the SIGTERM signal.

The handlers should print on stderr a message like ``I've been

killed'' and then exit(1).

The second child forked operates as in part (D). However the program

convert, which overlays the second child, needs to be modified.

Instead of using stdio, it will use the read and write system calls

directly. Pass to read and write an array of size BUFSIZ (defined in

stdio.h) rather than reading one char a time. It will exit(0) upon

hitting EOF on stdin. Also, you will need to add a signal handler to

convert for the SIGPIPE signal that it may get (see below) if count

terminates prematurely. The signal handler should print a message like

``I've been killed because my pipe reader died!'' on stderr and then

exit(2).

The first child forked operates as in part (D). However the program

count, which overlays the first child, needs to be modified. If the -n

argument has been passed to it, the program count reads the number n

(see atoi, strcmp, and /usr/include/string.h) and uses it as the

maximum number of characters to write to its stdout. It will exit

after writing this number of characters to stdout. If there is no

argument -n passed, then count writes all characters to its stdout.

Remember that count prints on stderr with fprintf the number of

non-alphabetic characters read and written. Since atoi returns 0 if it

is passed a string that is not an integer, only positive numbers are

acceptable for n. So, if atoi returns 0 or a negative number, then

count should print an informative message on stderr and do an exit(2).

This may cause LINUX to kill the second child forked, convert, with a

SIGPIPE signal. This would happen if convert is reading a large file

and still actually writing to the pipe (the SIGPIPE is generated when

convert tries to write to a pipe with no reader). Remember to add a

signal handler to convert for this possibility. If the input file or

data is short, it may happen that convert can run to completion and

exit before count can detect an illegal -n argument. So a SIGPIPE may

or may not occur when the -n argument is illegal. For testing your

code when the -n argument is illegal, you can put a sleep(5) in

convert so it hangs around long enough for the parent to see that

count has done an exit(2). Or have the program read from a very big

file.

------------------------------------------------------------------------

Note:

On part (E) of the warm-up programming assignment, be careful when checking

out the part of your program that deals with an invalid "-n" argument passed

to count. If count does an exit(2) when it detects an invalid "-n" passed

to it, then what actually happens is that the second child, convert, may get

a SIGPIPE signal, because it might try some more writes to a pipe that now has

no reader (count has died). So the parent will probably get an error message

when it tries to kill the second child, convert. Not to worry: just print

out an informative message if kill returns -1, "second child already dead",

and then have the parent do its waits, and then exit. Also remember that

fork returns to the parent the pid of the child and that wait returns the

pid of the child reaped to the parent (or -1 if the parent has no children).

----------------------------------------------------------------------

The exit status of a reaped child is returned to the parent in the

status variable that the parent passed to wait. The include file

/usr/include/sys/wait.h defines a macro WEXITSTATUS(status) that can

be used to extract the number the child passed to exit when the child

terminated. Question: ``I don't understand the purpose of

WEXITSTATUS(). I thought when wait(&status) returned, status would

contain the value that the child exited with. If that's the case, then

why do I need WEXITSTATUS() to get the exit value?'' Answer:

WEXITSTATUS() is a macro defined in <wait.h> that extracts the exit

value from status. The exit value is only 8 bits and it has been

masked into status along with other information that you can read

about in the Solaris man page.

Check all system calls for the error return (less than zero or -1).

For all error and debug messages, use ``fprintf(stderr, ...),'' or use

the function perror (do a ``man perror'' on the Suns), or use the

external variable errno.

------------------------------------------------------------------------

**Figure 1: Parent skeleton code.**

#include <signal.h>

#include <setjmp.h>

...

jmp\_buf env;

...

main(int argc, char \*argv[]) {

...

signal(SIGALRM, handler);

...

if (setjmp(env) != 0) {

/\* longjmp below causes setjmp to return here \*/

/\* kill children ... \*/

/\* wait loop (as below) for children ... \*/

exit(1);

}

/\* setjmp returns here first time called \*/

...

/\* open, creat, pipe, close, dup, fork 2 children ... \*/

...

alarm(15); /\* set alarm for 15 seconds \*/

while ((pid = wait(&status)) > 0) { /\* wait loop \*/

fprintf(stderr, "child pid=%d reaped with exit status=%d\n",

pid, WEXITSTATUS(status));

/\* if pid is count's and exit status == 2, kill convert ... \*/

}

alarm(0); /\* cancel alarm \*/

...

exit(0);

}

void handler() {

fprintf(...

longjmp(env, 1);

exit(1);

}

========================================================================

Check Code For

. convert uses read/write

. count and convert have SIGTERM handlers

. count and convert print pid

. convert has SIGPIPE handler

. count processes -n argument

. driver

- checks exit status of count and kills convert

- kills both if gets SIGALRM

Check Execution For

driver # type on keyboard, watch screen

driver nonexistent\_file # cannot open error message

driver infile

driver infile outfile

driver -20

driver -20 infile

driver -20 infile outfile

driver -abc infile # both children handle SIGTERM

# count prints illegal -n

driver # wait 30 seconds

cat /tmp/big\_file | driver -abc # test SIGPIPE handler in convert

cat /tmp/big\_file | driver -10 # test SIGPIPE handler in convert

convert </tmp/big\_file | more # ditto