Operating Systems Lab
Dept of CSE, LNMIIT, Jaipur
18 Jan 2021
Lab Session 01

Note:

1. The exercises require some work outside the scheduled lab session. The lab sessions are primarily for providing closer support to the students from a tutor. The lab sessions require compulsory attendance and will be used to ensure that the students are learning in a continuous and progressive manner. Your work in each session will be assessed and grades recorded.
2. Each student is expected to have a personal computer with a recent specification. However, we will request each student to install an Oracle VM Virtual box (https://www.virtualbox.org/) and install an Ubuntu 20 operating system on it. This will ensure that the class has a common work platform. If you have difficulty in this regard, please discuss it with your instructor.

Section 1 – Installing Ubuntu 20

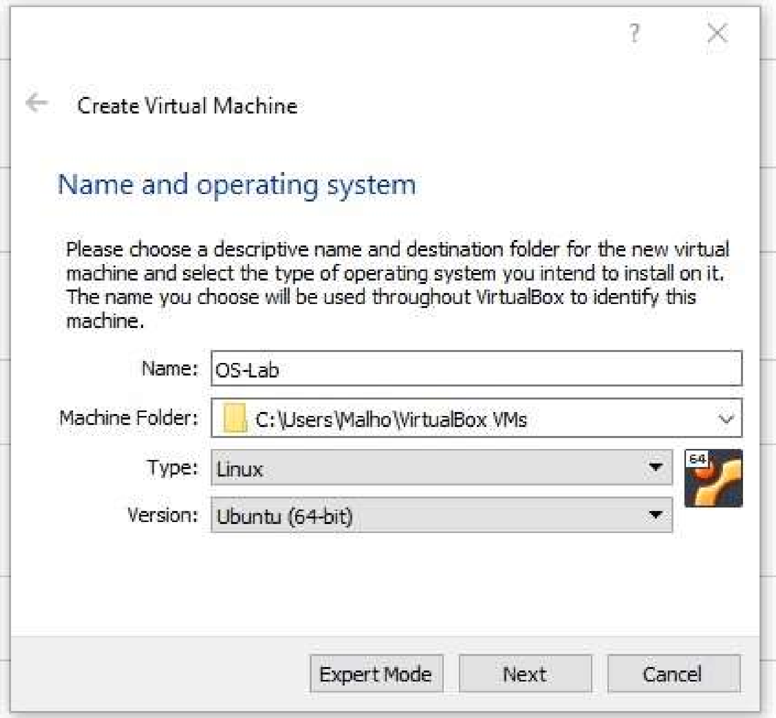
In this section we will create a virtual computer and install an Ubuntu OS on it.

Phase 1: Creating a virtual computer.

On your host computer download Virtual box software from https://www.virtualbox.org/ and install the software. Select "New" tab (on top menu bar) and a window as in Figure 1 should pop-up.

1. Fill details as suggested in Figure 1. If your host processor is a 32-bit computer, you need to select appropriately. Press "Next".
2. When you press "Next" button, you will be asked to select "Memory Size". Choose a value 2GB or more. Make sure that there is a large amount of memory left for your native operating system.
3. On next screen, select option "Create a virtual hard disk now".
4. On accepting Create, accept suggested Hard disk file type.
5. Storage on physical hard disk: Leave it as Dynamically allocated.
6. File location and size: I suggest 20GB. But 10 GB may be enough.
7. Finally, press Create button and a virtual computer is created on the left sidebar. Give time for this action to complete.

The newly created virtual computer is without an operating system. It needs the operating system.



Name and operating system

Please choose a descriptive name and destination folder for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: OS-Lab

Machine Folder: C:\Users\Malho\VirtualBox VMs

Type: Linux

Version: Ubuntu (64-bit)

Figure 1: Use button "New" on Oracle VM VirtualBox Manager to create a virtual machine for OS-Labs.

Phase 2: Installing a real OS in the Virtual Computer

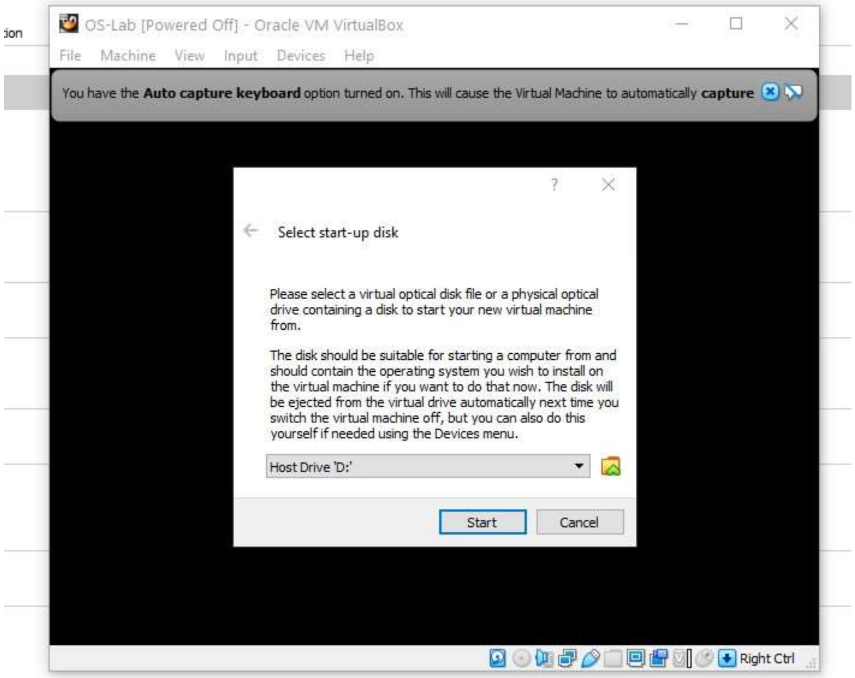
On your host computer download an Ubuntu Desktop installation. Relevant, website is:

https://ubuntu.com/download/desktop

Extract the contents if relevant.

IF YOUR COMPUTER IS A 32-BIT COMPUTER, YOU MAY WISH TO INSTALL UBUNTU 16.04 instead of Ubuntu 20. The procedure is like what we have described above. A good link is: https://releases.ubuntu.com/16.04.7/ubuntu-16.04.6-desktop-i386.iso

1. Click on the newly created computer icon on the left bar in VirtualBox window to turn ON your new (virtual) computer. The pop-up box in Figure 2 appears.
2. Click on the browse box in the window (Yellow box in bottom right section of the window). And then browse to your newly downloaded iso file (in your host computer's download folder). Press Start button.
3. Select install Ubuntu in the next window and follow the instructions. Accept all default suggestions till you come to the world map. Select your country correctly.
4. See Figure 3. I filled my name as os-lab. Use identical password (os-lab). I am not likely to have anything important here – you may do the same as password will be important even though I have chosen to log in automatically at the bottom of the form. Press Continue.
5. Restart. And press enter (with cursor on the simulated computer screen when it tries to restart).
6. I declined automatic updates. But, I allowed new updates when asked in the initial start-up screen. If you are on a 32-bit processor computer then AVOID UPDATES.


OS-Lab [Powered Off] - Oracle VM VirtualBox
File Machine View Input Devices Help
Start
Cancel
Right Ctrl

You have the **Auto capture keyboard** option turned on. This will cause the Virtual Machine to automatically **capture**

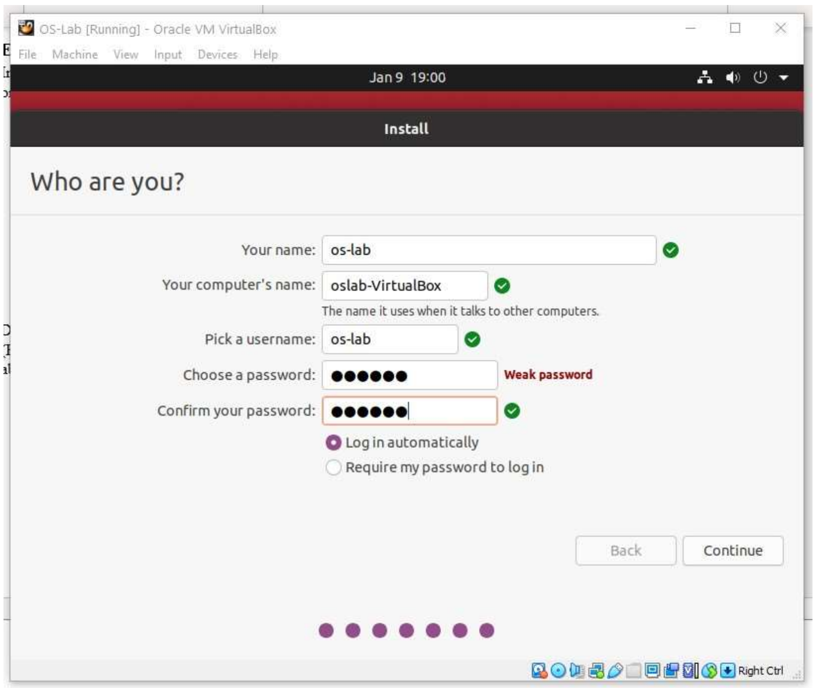
Select start-up disk

Please select a virtual optical disk file or a physical optical drive containing a disk to start your new virtual machine from.

The disk should be suitable for starting a computer from and should contain the operating system you wish to install on the virtual machine if you want to do that now. The disk will be ejected from the virtual drive automatically next time you switch the virtual machine off, but you can also do this yourself if needed using the Devices menu.

Host Drive 'D:'

Figure 2: Ubuntu installation disk selection.


OS-Lab [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Jan 9 19:00
Install
Back
Continue
Right Ctrl

Who are you?

Your name: os-lab

Your computer's name: oslab-VirtualBox

The name it uses when it talks to other computers.

Pick a username: os-lab

Choose a password: ●●●●●● Weak password

Confirm your password: ●●●●●●

[✓ Log in automatically]

Require my password to log in

Figure 3: Please select and REMEMBER your password carefully as you will need it often. Chose log in automatically for easy access.

7. You can right click to start your terminal windows under Ubuntu. Open a terminal window after update process has completed and finished.
8. In this window run command: sudo apt-get install indicator-appmenu
9. Provide password when asked.
10. After this, close the terminal window and right click to adjust your virtual computer's screen size to suit your real computer screen.

This finishes our Section A of the first lab.

PLANS FOR TWO LATER LABS

I do plan to come back to a similar work later again. However, we will work under a different arrangement for the exercises planned for later part of the term.

Oracle VirtualBox is an intermediatory software. It relies on the host Processor and OS (many of you will have Windows as your host OS) to maintain the virtual computer (os-lab).

In the later exercises (perhaps after mid-term), we will work on a full-function hardware simulator called Bochs – this simulates (almost) all activities of a X86 computer. And, we have an unfinished tiny OS source code, called PintOS. We will complete and implement some (minor) features of this OS and test it on Bochs. The idea is to provide some feel for the internals of an OS to the students.

============== SECTION B ================
This part is adapted from an IIIT, Guwahati assignment.

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

There are four tasks in this section: A to D.

Only the work completed by the end of student's scheduled Lab 1 session will be included in the assessment.

(A tutorial lessons on basic Linux programming are included as needed in this assignment.)

Task A: You are to write a C program that prints out all 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.

Task B (count): You are to write another C program which will read characters from its standard input stdin, count the number of NON-alphabetic characters including the 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).

Task C (convert): You are to write yet another C program that reads characters from stdin, and rotates alphabetic characters by 13 positions (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.

Explanation: Lower-case and upper-case letters are rotated in their separate groups. Thus, Letter A is printed as letter N; letter b as letter o; and (interestingly), letter N will be printed as letter A.

You may wish to run this program as (assuming you have compiled it as executable covert):

```
$./convert | ./convert
```

I miss LNMIIT campus life.

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 invoked 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.

```
int fd;
fd = open(const char *pathname, int flags, mode_t pmode);
```

Another Linux open system call requires 3 arguments. This call is preferred system call when creating a new file. The third argument sets desired file access permissions. When this information is not available (as in the two argument open()), it may be assumed to be 0000 -- that is no access permission to any user. So, the owners processes are not permitted to write into the file.

Suggested call to create a file is:

```
open (“File-name”, O_CREATE|O_RDWR, 0600);
// 0600 means read and Write permissions to the creator.
```

We recommend that you use system call creat() to create a new file. And use open() to open previously created files.

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.
-----------------------------------------------------------------------
```

Task D: You are to combine the three programs completed in tasks (A), (B) and (C) through a single driver program that:

1. reads from the first named file argument, rotates the alphabetic letters, and writes out all characters, whether rotated or not; 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 create 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, rotate the alphabets (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
```

Addendum:

0. Here is a tutorial on pipe() https://tldp.org/LDP/lpg/node11.html

1. The program we prepare for Part D needs care. If the driver process waits for two children processes to finish (See item 3 below), then it must close at least the write end of the pipe. Ideally, it should close all file descriptors other than 0, 1 and 2 (that is, everything but stdin, stdout and stderr) before beginning to wait. Take care that the close() are being done ONLY in the original process. Forked child processes need similar but different set of file descriptor closings. if your original process is not waiting for the children processes to finish, then all its file descriptors will be closed by the kernel and you will be fine.

2. Wait system call requires:

```
    #include <sys/type.h>
    #include <sys/wait.h>
    int status.
    wait(&status);
```