Topic to be covered

- Dup/ Dup2 system call
  - Read end redirection
  - Write end redirection
  - Error end redirection

- Exec () system call

Objective

- Students will be able to duplicate the descriptor(s) in a program.
- Students will be able to replace the caller process with the image of a new program.
- Students will learn how various processes communicate by reading or writing to the pipe and also how to have two-way communication between processes.
- Students will learn how shell runs a UNIX command having multiple processes connected via pipe with input/output redirections.

Pre-Requsit:

- Student must know about process creation and termination
- Student must know interprocess communication using unnamed pipes.
- Student must know basic C Programing
- Student must have GCC compiler
- Student must have visual studio code

Implementation of Redirection

When a process forks, the child inherits a copy of its parent's file descriptors. When process execs, the standard input, output, and error channels remain unaffected. The UNIX shell uses these two pieces of information to implement redirection.

To perform redirection, the shell performs the following series of actions:

- The parent shell forks and then waits for the child shell to terminate.
- The child shell opens the file "output", creating it or truncating as necessary.
- The child shell then duplicates the file descriptor of "output" to the standard output file descriptor, number 1, and then closes the original descriptor of "output". All standard output is therefore redirected to "output".
- The child shell then exec's the ls utility. Since the file descriptors are inherited during an exec ( ), all of standard output of ls goes to "output".
- When the child shell terminates, the parent resumes. The parent's file descriptors are unaffected by the child's actions, as each process maintains its own private descriptor table.

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/file.h>
int main (int argc, char** argv)
{
    int   fd ; /* file descriptor or pointer */
    fd = open (argv[1], O_CREAT | O_TRUNC | O_RDWR, 0777) ;
                        /* open file named in argv[1] */
   dup2 (fd, 1) ;  /*  and assign it to fd file pointer */
   close (fd) ; /* duplicate fd with 1 which is standard output (the monitor) */
   execvp (argv[2], &argv[2]) ;
        /* the output is not printed on screen but is redirected to "output" file
*/

printf ("End\n") ; /* should never execute */
}
```

To execuate the above program you have to pass input and output file/command name from command line.
./a.out ls out.txt

Exec system call (Also Covered in Fork Lab).

int execl(const char *_path_, const char *_arg_, ...);
int execlp(const char *_file_, const char *_arg_, ...);
int execle(const char *_path_, const char *_arg_,..., char * const _envp_[]);
int execv(const char *_path_, char *const _argv_[]);
int execvp(const char *_file_, char *const _argv_[]);
int execvpe(const char *_file_, char *const _argv_[],char *const _envp_[]);

The **exec**() family of functions replaces the current process image with a new process image. The initial argument for these functions is the name of a file that is to be executed.

The *const char *arg* and subsequent ellipses in the **execl**(), **execlp**(), and **execle**() functions can be thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments *must* be terminated by a NULL pointer, and, since these are variadic functions, this pointer must be cast *(char *) NULL*.

The **execv**(), **execvp**(), and **execvpe**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers *must* be terminated by a NULL pointer.

The **execle**() and **execvpe**() functions allow the caller to specify the environment of the executed program via the argument *envp*. The *envp* argument is an array of pointers to null-terminated strings and *must* be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable *environ* in the calling process.

The **execlp**(), **execvp**(), and **execvpe**() functions duplicate the actions of the shell in searching for an executable file if the specified filename does not contain a slash (/) character. The file is sought in the colon-separated list of directory

pathnames specified in the **PATH** environment variable. If this variable isn't defined, the path list defaults to the current directory followed by the list of directories returned by *confstr(_CS_PATH)*. (This **confstr**(3) call typically returns the value "/bin:/usr/bin".)

If the specified filename includes a slash character, then **PATH** is ignored, and the file at the specified pathname is executed.

dup / dup2 System Call

int dup (**int** *oldfd*)
int dup2 (**int** *oldfd*, **int** *newfd*)

dup ( ) finds the smallest free file descriptor entry and points it to the same file as *oldfd*. dup2 ( ) closes *newfd* if it's currently active and then points it to the same file as *oldfd*. In both cases, the original and copied file descriptors share the same file pointer and access mode.

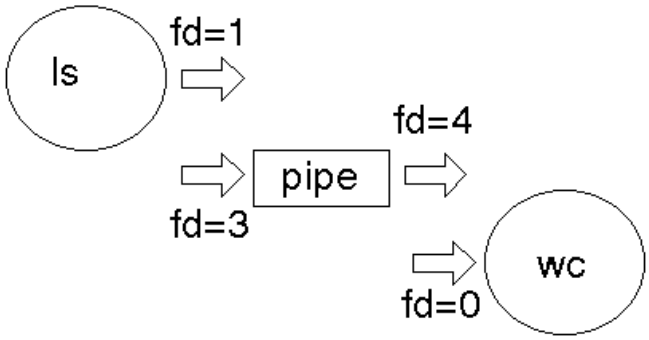
They both return the index of the new file descriptor if successful,
and –1 otherwise.
dup/dup2 duplicates an existing file descriptor, giving a new file descriptor that is open to the same file or pipe. The two share the same file pointer, just as an inherited file descriptor shares the file pointer with the corresponding file descriptor in the parent. The call fails if the argument is bad (not open) or if 20 file descriptors are already open.

A pipeline works because the two processes know the file descriptor of each end of the pipe. Each process has a stdin (0), a stdout (1) and a stderr (2). The file descriptors will depend on which other files have been opened, but could be 3 and 4, say.

Suppose one of the processes replaces itself by an "exec". The new process will have files for descriptors 0, 1, 2, 3 and 4 open. How will it know which are the ones belonging to the pipe? It can't.

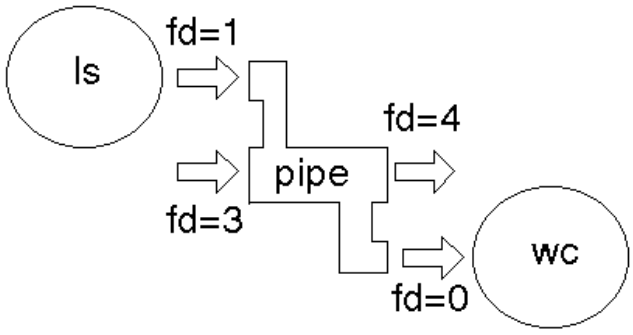
Example

To implement "ls | wc" the shell will have created a pipe and then forked. The parent will exec to be replaced by "ls", and the child will exec to be replaced by "wc" The write end of the pipe may be descriptor 3 and the read end may be descriptor 4. "ls" normally writes to 1 and "wc" normally reads from 0. How do these get matched up?

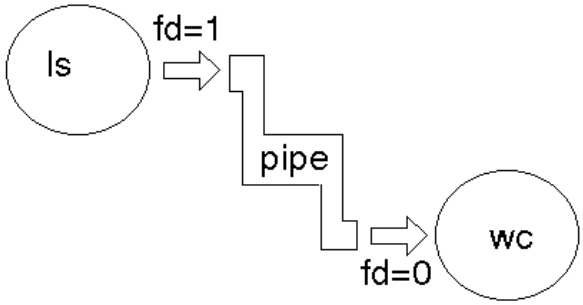


The dup/dup2 function call takes an existing file descriptor, and another one that it "would like to be". Here, fd=3 would also like to be 1, and fd=4 would like to be 0. So we *dup* fd=3 as 1, and *dup* fd=4 as 0. Then the old fd=3 and fd=4 can be closed as they are no longer needed.

After dup



After close



The UNIX shells use unnamed pipes to build pipelines. They use a trick similar to the redirection mechanism to connect the standard output of one process to standard input of another. To illustrate this approach, here's the program that executes two named programs, connecting the standard output of the first to the standard input of the second.

```
#include <stdio.h>
#include <string.h>

#define  READ    0
#define  WRITE   1
 int  main (int argc, char **argv)
{
 int     pid, fd [2] ;

 if (pipe(fd) == -1)
 {
   perror("pipe failed");
   exit(1);
 }
 if ((pid = fork( )) < 0)
 {
   perror("fork failed");
   exit(-1);
 }
 if ( pid  >  0 )                          /*  parent, writer */
 {
    close ( fd [READ] ) ;                 /*  close unused end */
    dup2 ( fd [WRITE], 1) ;              /* duplicate used end to standard out */
    close ( fd [WRITE] ) ;                /*  close used end  */
    execlp ( argv[1], argv[1], NULL) ;       /*  execute writer program  */
 }
 else  if(pid==0)                                  /*  child,  reader  */
 {
    close ( fd [WRITE] ) ;                  /* close unused end */
    dup2 ( fd [READ], 0) ;                 /* duplicate used end to standard input */
    close ( fd [READ] ) ;                  /* close used end */
    execlp ( argv[2], argv[2], NULL) ;         /* execute reader program */
 }
}
```

Run the above program as

$./a.out who wc

Variations

Some common variations on this method of IPC are:

A pipeline may consist of three or more process (such as a C version of **ps | sed 1d | wc -l**).

- In this case there are lots of choices The parent can fork twice to give two children.
- The parent can fork once and the child can fork once, giving a parent, child and grandchild.
- The parent can create two pipes before any forking. After a fork there will then be a total of 8 ends open (2 processes * two ends * 2 pipes). Most of these will have to be closed to ensure that their ends up only one read and only one write end.
- As many ends as possible of a pipe may be closed before a fork. This minimizes the number of closes that have to be done after forking.
- A process may want to both write to and read from a child. In this case it creates two pipes. One of these is used by the parent for writing and by the child for reading. The child for writing and the parent for reading use the other.

Write C programs to implement the given UNIX commands:
All codes must be compiled using –Wall –Werror flags.

CP Task 1

```
ls / -r  > input.txt
```

CP Task 2

```
grep “m” < input.txt | sort  > output.txt
```

Task 3

```
ps | sed 1d | wc -l
```