PURPOSE: The purpose of this lab is to allow students to learn a user interface aspect of a UNIX shell. Student will work with process management and some basic system calls.
Important note: please use sp1, sp2, sp3, or atoz servers for this lab.

UNIX Shell

A simple shell is a basic shell program that supports commands with I/O re-direction. Once a user enters a command string (ending with a return key), your program should parse the command line and determine whether it has a regular command, or contains I/O redirections signified by **>** for output to a file or **<** for input from a file.

The built-in functions are **not** executed by forking and executing an executable. Instead, the shell process executes them itself. All other commands must be executed in a child process.

Your shell is basically an interactive loop:

- it repeatedly prints a prompt "**csc60msh >** ",
- parses the input, executes the command specified on that line of input,
- and waits for the command to finish.

You should structure your shell such that it creates a new process for each new command (except for *cd*, *pwd*, and *exit*). Running commands in a new process protects the main shell process from any errors that occur in the new command.

FILES TO COPY:

To get the files you need, first move to your class folder by typing: **cd csc60**
The following command will create a directory named **lab7** and put some sample files into it below your csc60 directory.
Type: **cp -R /gaia/home/faculty/bielr/classfiles_csc60/lab7 .**
Spaces needed: (1) After the **cp** ↑ Don't miss the space & dot.
(2) After the **-R**
(3) After the directory name at the end & before the dot.
You will have created a *lab7* directory and copied *redir.c* and *waitpid.c*
Take your finished lab6.c (from the lab6 directory) and copy it into lab7.c. **$cp lab6.c ../lab7/lab7.c**

Please follow these steps:

- Review the source codes, compile, and execute the programs. Examine the output texts to understand the behavior of each program.
- **Function process_input**. Input processing. Calls the function ***handle_redir*** and then does the ***exec***.
- **Function handle_redir.** Handling **input and output redirection**: Please also review the sample code for *redir.c* which can be found in your *lab7* directory. You should be able to reuse part of the code here. (Separate function.) Below are the high level steps:

Order of Execution	Implementing "<" and ">":	Examples:
1	Open the file	newfd=open(.....)
2	Assign newfd to 0(if "<") or 1(if ">") using **dup2**	dup2(newfd,0);
3	Close your file	close(newfd);
4	Execute the command using execvp system call (which will happen in *process_input*)	The command reads from 0 which is your file now

- **Handling input errors**: Be very careful to check for error conditions at all stages of command line parsing. Since the shell is controlled by a user, it is possible to receive bizarre input. For example, your shell should be able to handle all these errors:

Input	**Error**
csc60mshell > /bin/cat < file1 < file2	ERROR – Can't have two input redirects on one line.
csc60mshell > /bin/cat	ERROR – No redirection file specified.
csc60mshell > > out.txt	ERROR - No command. Make sure file out.txt is not overwritten
csc60mshell > cat test.c >	ERROR - No redirection file specified.

(Note: even though the Unix shell MIGHT not provide you an explicit error message for the above cases, the actual expected result is incorrect. Example: /bin/cat < file1 < file2)

• **Handling test cases**: Your program should be able to handle many examples of commands. Check the list under the section on preparing your script.

Pseudo Code

```
/*----------------------------------------------------------*/
int main (int argc, char **argv)
{
   while (1)
  {
          int childPid;
          char *cmdLine;
          print the prompt();     /* i.e. csc60mshell > , Use printf*/
          cmdLine= readCommandLine(); /* use fgets not gets, provided */
          argc = parseCommand(cmdLine); /* use provided function parseline */
          if ( argc compare equal to zero)
              /* a command was not entered */
              continue to end of while-loop
          if (isBuiltInCommand(cmd);
              /* exit, cd, pwd */
              /* These should be all working from lab6 */

          ………………………. IGNORE line - to be removed…………………..
          else
          {
              pid = fork();
              switch(pid)
              {
                 case -1:    /* error */
                 case 0:     /* child process called */
                    process_input(argc, argv)
                    break;
```

```
                default:    /* parent Process*/
                   /* use code provided */
              }              /* end of switch */
          }              /* end of if */
   }              /* end while */
}                 /* end main */
/*-------------------------------------------------------*/
int parseline(char *cmdline, char **argv)
{
   /* CODE PROVIDED
      Using white-space as a delimiter, the command line is broken up in to pieces,
      which are placed into an array of pointers named argv.
      Ex:  command = "ls > ls.out"
      argv[0] = ls;  argv[1] = >;  argv[2] = ls.out;  and argc = 3.  */
}
/*-------------------------------------------------------*/
void process_input(int argc, char **argv)
{
    call handle_redir(argc, argv);
    call execvp  and do an error check

}
/*-------------------------------------------------------*/
void handle_redir(int count, char*argv[ ])
{
         (1) handle error situations for > and <
           EX: trying to open more than one file per redirect.  "ls > x.out > y.out"
                Issuing no command before the redirect symbol. "> x.out" or "< x.in"
                no file specified.  "ls > " or "cat < "
           You may print your error message using fprintf(stderr,"…")
       (2) open the needed file (fd = open(…))
           If opening for output,
                 use flags:  (1) to write; (2) to create file if needed; (3) to truncate existing file to zero
                             length
                 use permission bits for:  (1) user-read; (2) user-write
           if opening for input,
                 use flags; (1) for read only
           check for error on open
       (3) after the open
           Issue a dup2 to stdin or stdout as needed.
           Close the file
           To set up for the exec call coming,
                 reset the element of argv that contained the redirect symbol to NULL
}
/*-------------------------------------------------------*/
```

C Library functions:

#include <string.h>
String compare:
```
int strcmp(const char *s1, const char *s2);
strcmp(….,"cd")
strcmp(….,"exit")
strcmp(….,"pwd")
strcmp(….,">")
strcmp(….,"<")
```

print a system error message:
```
perror("Shell Program error");
```

input of characters and strings:
```
fgets(cmdline, MAXLINE, stdin);
```

Useful Unix System Calls:

getenv/setenv: get/setenv the value of an environment variable
path = **getenv**("PATH");
cwd = **getenv**("PWD");
setenv("PWD", tempbuf,1);

getcwd: get current working directory.

chdir: change the current working directory (use this to implement cd)

fork-join: create a new child process and wait for it to exit:
```
if (fork() == 0)    { // the child process
        ……….
} else { // the parent process
        pid = wait(&status);
}
```

execv: overlay a new process image on calling process
execvp(full_path_name, command_argv_list, 0);
(The **exec**() functions only return if an error has occurred. The return value is -1, and *errno* is set to indicate the error.)

open, close, dup2: for I/O Redirection:
int fid = **open**(filename, O_WRONLY|O_CREAT);
close(1);
dup2(fid,1);
close(fid);

Exit shell:
_exit(…); - exit without flushing stdio. Use by child process.
exit(); - exit with flushing stdio.Use by parent process.

Compilation & Building your program

The use of *gcc* is just fine. If you want to have the output go elsewhere from a.out, type:

```
gcc –o  name-of-executable  name-of-source-code
```

Partnership

Students may form a group of 2 students (maximum) to work on this lab. Include both names on all files. Both students should submit both files under their own name on SacCT. As usual, please always contact your instructor for questions or clarification.

Hints

Writing your shell in a simple manner is a matter of finding the relevant library routines and calling them properly. Please see the resources section above.

Remember to get the **basic functionality** of your program working before worrying about all of the error conditions and corner cases. For example, for your shell, first get a single command running (probably first a command with no arguments, such as "ls"). Then try adding more arguments. Next, try working on multiple commands. Make sure that you are correctly handling all of the cases where there is miscellaneous white space around commands or missing commands. Finally, support for built-in commands, redirection, and pipes (for next assignment).

I strongly recommend that you check the return codes of all system calls from the very beginning of your work. This will often catch errors in how you are invoking these new system calls. And, it's good programming practice.

Keep versions of your code. This is in case you need to go back to your older version due to an unforeseen bug/issue.

I hope you enjoy this project….or at least learn from it ….and don't lose too much of your sanity.

Marks Distribution

Your documented c program: lab7 with a working **handle_redir** function	58 points

Deliverables

Your source file(s):

1. **lab7.c**
2. **YourNameLab7.txt**
    a. Your program's output test (with various test cases). Please use the UNIX **script** command to capture your program's output.

- Submit your ***two*** files to **SacCT**

All files should include the names of students involved in this assignment
Teams of two students should submit both files under their own name on SacCT

➔ more on next page.

Team Members:

Create a small file and type into it both of your names. After you open your script, type: **cat small_file_name**
and your script will display both names at the top.
What you name the script itself is up to you, but not just plain *lab7.txt*.

Preparing your script file:

Run the program, and enter in sequence:

```
csc60mshell > the Enter Key,
csc60mshell > a Space, and then the Enter Key,
csc60mshell > pwd  , This comma was a typo. Ignore it.
csc60mshell > cd ..
csc60mshell > cd

csc60mshell > ls
csc60mshell > ls > lsout                #redirect ls's stdout to file ls.out
csc60mshell > ls > ls1out > ls2out      #should be an error
csc60mshell > wc < lsout
csc60mshell> wc < ls.out < redir.c      #should be an error
csc60mshell > cat redir.c               #should reside in lab7 directory
csc60mshell > cd /usr/bin
csc60mshell > cd
csc60mshell > /usr/bin/ps
csc60mshell > wc foo1.txt               #use a text file that you have
csc60mshell > gcc –o  redir  redir.c  -g
csc60mshell > exit
```

If all worked, submit your script.