EECS 3540 – Operating Systems & System Programming
Spring 2023 – Programming Project #1 – Due February 23rd at 11:59 PM

In this project, you will simulate parts of the system program **`time`** (see **`man time`**), which tells how long (real elapsed time), and how much CPU time (in both user and kernel modes) a command takes to run. Try running the command:

```
time df
```

The **`df`** command is "**d**isplay **f**ilesystems" – this shows how much space is used / available on each storage device in the system. On my VM, this command produced the following output (yours should look similar):

```
lgt@os-lgt:~$ time df
Filesystem     1K-blocks     Used Available Use% Mounted on
tmpfs             398320     2052    396268   1% /run
/dev/sda3       30267332 12488344  16216160  44% /
tmpfs            1991592        0   1991592   0% /dev/shm
tmpfs               5120        4      5116   1% /run/lock
/dev/sda2         524252     5364    518888   2% /boot/efi
tmpfs             398316     2404    395912   1% /run/user/1000
/dev/sr1         3737140  3737140         0 100% /media/lgt/Ubuntu 22.04.1 LTS amd64
/dev/sr0          129834   129834         0 100% /media/lgt/CDROM

real    0m0.004s
user    0m0.003s
sys     0m0.001s
lgt@os-lgt:~$
```

Output of the **`df`** command

Real (elapsed), user-mode, and kernel-mode times spent running the program represented by the command

Your program will accept *__multiple__* command-line arguments, each of which is a command to run. Your main program will be responsible for parsing the commands, and keeping (and displaying) the overall time totals. The main program will call **`fork`** for each command, creating a *child* process, which will execute the command by forking a *grandchild* process to actually execute the command. The child process will use the system call **`times`** (see the **`man`** pages) to get the user- and system-mode times for the command's execution (how much time the grandchild process spent in user mode, and time it spent in kernel mode), as well as using the **`gettimeofday`** function to get the actual (wall clock) time. The main (parent) function will also use these system calls to get the user, system, and elapsed times for the *entire* set of commands, and display the totals at the end of execution.

You will want to call **`times`** to get the individual child times (one-by-one as you execute the command(s)). When the commands have all been executed, call times again to get the grand totals for the parent and all of the subprocesses it created along the way. In your summary statistics, make sure you compute the *total* elapsed time (**`gettimeofday`**) and the aggregate user and system times (**`times`**) from each child process's execution.

The **`gettimeofday`** function (which requires you to **`#include <sys/time.h>`**) has the format:

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Make sure you read the **`man`** page on this function (you will likely spend a LOT of time in **`man`** on this project!). You don't need to use the **`tz`** parameter; you may leave that one **`NULL`** (Ubuntu doesn't seem to do anything with it; I got zeroes for the two members of that struct when I tried it). Remember that when the parameter list in the documentation includes **`*`**, the function expects to *receive* a __pointer__ to something, which means you must *pass* the *__address__* of the something (by using **`&`**). So, you may have something like:

```
struct timeval tvStruct;
int returnValue = gettimeofday(&tvStruct, NULL);
```

Many of the functions you will be using have return codes to tell you whether they were successful (or not). Don't ignore these return values; if your code *isn't* working correctly, they can be a wealth of troubleshooting information.

The **gcc** complier is (in some ways) smarter than the one built into Visual Studio that you might have used previously. In the case of something like **gettimeofday**, which takes two pointers as its arguments, the compiler will issue a warning if it knows you're calling it with **NULL** for one of the arguments (after all, how many times have we all run into parameters we were expecting to be able to *use* that turned out to be **NULL**, and then generated an exception?). In order to suppress these warning messages, include the compiler option **-Wno-nonnull** on the **gcc** command line:

```
gcc -o lab1 -Wno-nonnull lab1.c
```

The next topic that needs to be covered (which will relate to your testing your programs, and how I will auto-grade them) is about *redirection*, which may be a new idea to you, so here's a tutorial on it:

You will want to test your program with commands that spend *long* enough in kernel (system) mode that the times show up (notice that the **sys** time on my **df** command above was **0.001** seconds; LOTS of commands take less than 1 ms of kernel time). What commands are "system-heavy" enough to do so? One possibility is to run a directory listing of everything on the entire system. Remember, there are no drive letters in Linux; all of your "drives" appear on the system as subdirectories somewhere below the root. The command:

```
ls -AlR /
```

will start at the root (**/**) of the directory tree, and recursively (**-R**) list everything it finds. Unfortunately, this is a LONG list, and you may run out of scrolling room in your terminal window if you try it (on my VM, it produced over 600,000 lines of output!). So, if we want to run the command, but don't want to wade through all of the output, what can we do? Pipes and redirection to the rescue!

We covered pipes a bit in class. Pipes essentially let us "string together" programs such that the output of one becomes the input to the next. We saw how to use the **more** program to mete out a program's output one screenful at a time. The command:

```
ps -el
```

lists all processes on the machine. There are <u>*lots*</u> of them (over 300 on my VM). If we want to see it one screenful at a time, we can "pipe" the output from **ps** to the screen with the **more** program and the pipe symbol (**|**):

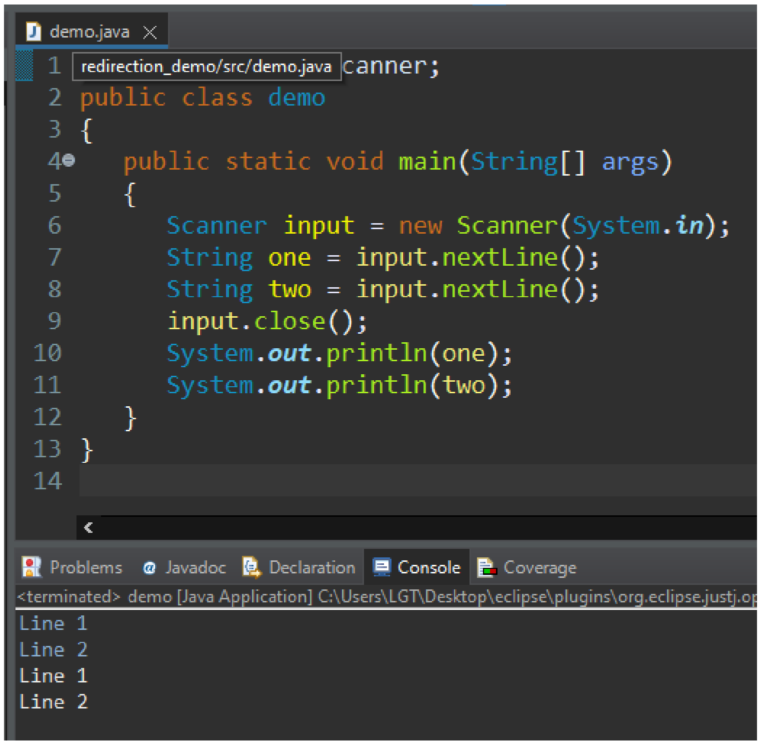
```
ps -el | more
```

What the **more** program actually does is take input from what it thinks is the keyboard, and sends it to the screen. The "what it *thinks* is the keyboard" is a <u>***key***</u> concept in Linux (and in Windows, too). In Java, you used **System.in** as the keyboard, and **System.out** as the console (terminal) window.

As it turns out, **System.in** doesn't *necessarily* mean the keyboard; rather, it means "the standard input source", which *defaults* to the keyboard, but we can override that connection if we so choose. It's the same with **System.out**. Consider the following Java code:

```
import java.util.Scanner;
public class demo
{
   public static void main(String[] args)
   {
      Scanner input = new Scanner(System.in);
      String one = input.nextLine(); // read in two lines of text
      String two = input.nextLine();
      input.close();
      System.out.println(one); // output the same two lines
      System.out.println(two);
   }
}
```

If we run this program from within Eclipse, it will read two lines of input from the keyboard and send the output to the console (screen):



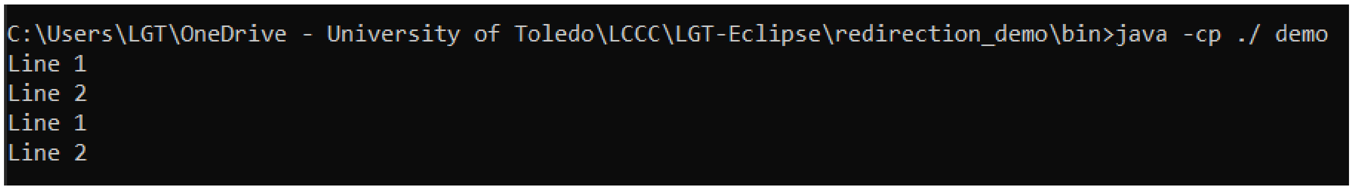
```
canner;
public class demo
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        String one = input.nextLine();
        String two = input.nextLine();
        input.close();
        System.out.println(one);
        System.out.println(two);
    }
}

```

```
<terminated> demo [Java Application] C:\Users\LGT\Desktop\eclipse\plugins\org.eclipse.justj.o
Line 1
Line 2
Line 1
Line 2
```

The two blue lines in the console window are the input as I typed them; the two white lines are the output. So far, so good.

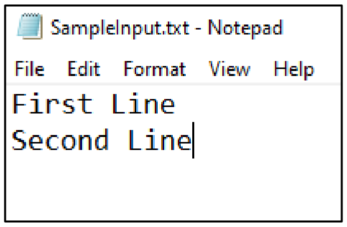
If we open a command window in Windows, and make our way (**`CD`**) to the **`bin`** folder holding our (compiled) **`.class`** file, we can run the program from the command line by starting the JVM (called **`java`**) and specifying the **`.class`** file:



```
C:\Users\LGT\OneDrive - University of Toledo\LCCC\LGT-Eclipse\redirection_demo\bin>java -cp ./ demo
Line 1
Line 2
Line 1
Line 2
```

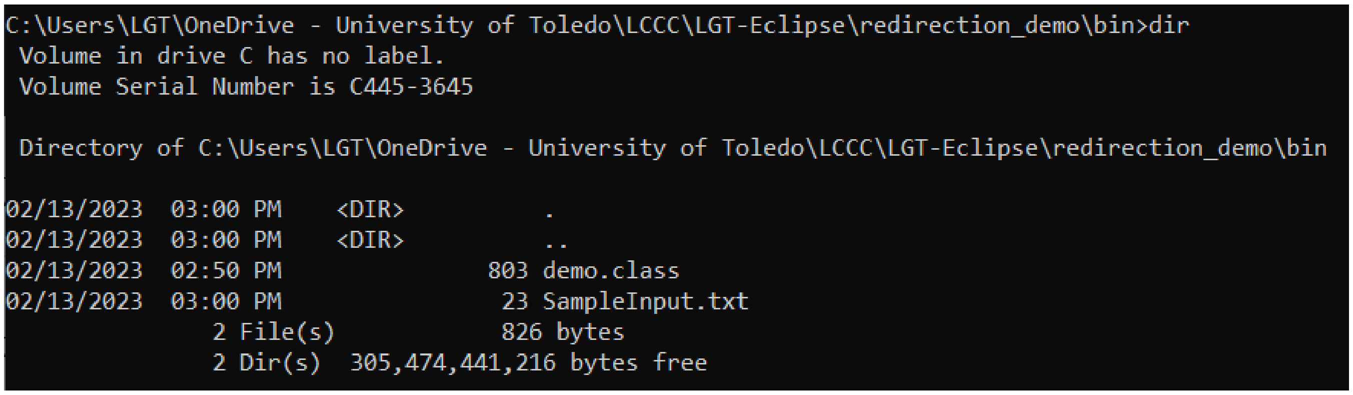
The "**`-cp ./`**" parameter tells the JVM that the file we're looking for is in the current directory. Again, the first two lines that appear in the screenshot above were the input as I typed them; the second two were the output.

Now, let's suppose we have a text file in the same directory called **`SampleInput.txt`**, which contains two lines of text:



```
First Line
Second Line
```

If we save that file in the same directory as the **`demo.class`** file (the **`bin`** folder):

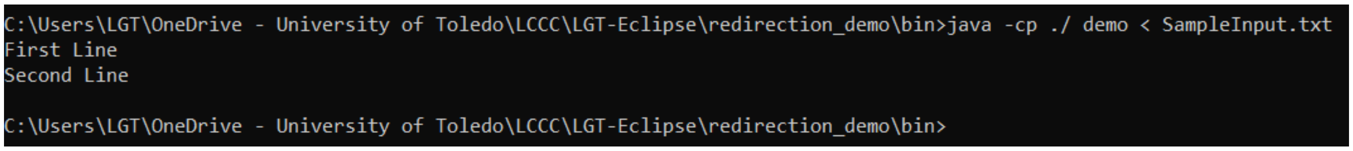


```
C:\Users\LGT\OneDrive - University of Toledo\LCCC\LGT-Eclipse\redirection_demo\bin>dir
 Volume in drive C has no label.
 Volume Serial Number is C445-3645

 Directory of C:\Users\LGT\OneDrive - University of Toledo\LCCC\LGT-Eclipse\redirection_demo\bin

02/13/2023  03:00 PM    <DIR>          .
02/13/2023  03:00 PM    <DIR>          ..
02/13/2023  02:50 PM               803 demo.class
02/13/2023  03:00 PM                23 SampleInput.txt
               2 File(s)            826 bytes
               2 Dir(s)  305,474,441,216 bytes free
```

Now we can run the program and "fake out" the JVM, and have the input come from **SampleInput.txt**, *rather than from the keyboard*, by using the "standard input redirection" operator (**<**) on the command line (think of this as an arrow that points FROM a file TO the program):



```
C:\Users\LGT\OneDrive - University of Toledo\LCCC\LGT-Eclipse\redirection_demo\bin>java -cp ./ demo < SampleInput.txt
First Line
Second Line

C:\Users\LGT\OneDrive - University of Toledo\LCCC\LGT-Eclipse\redirection_demo\bin>
```

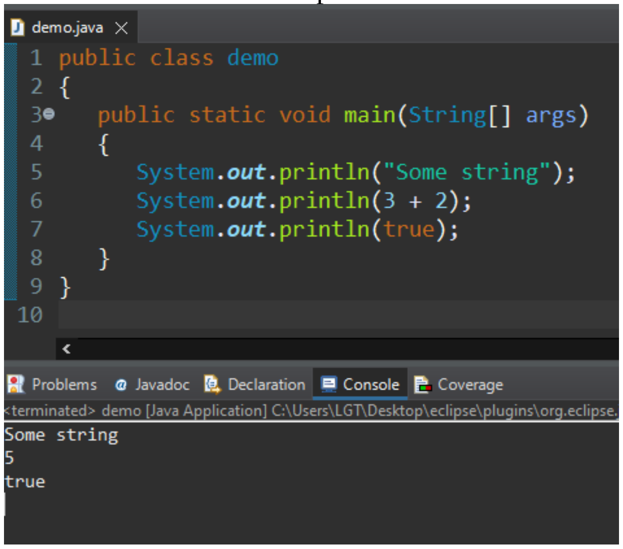
This time, we don't see the first two lines of input, because we didn't type them! We just see the output the program generated.

Similarly, we can direct the *output* of a program to a file, rather than having it go to the screen ("standard output"), by using the redirection character (**>**). Let's change the code in **demo.java** to this:

```
public class demo
{
   public static void main(String[] args)
   {
      System.out.println("Some string");
      System.out.println(3 + 2);
      System.out.println(true);
   }
}
```

If we run this code from within Eclipse or the command line, we get what we would expect:

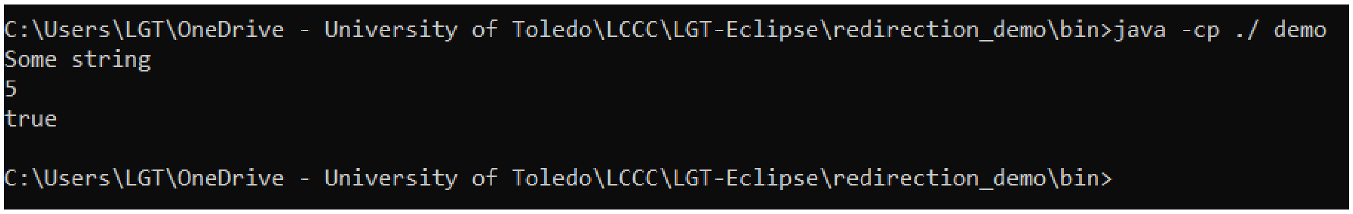
Eclipse



```
public class demo
{
   public static void main(String[] args)
   {
      System.out.println("Some string");
      System.out.println(3 + 2);
      System.out.println(true);
   }
}
```

```
<terminated> demo [Java Application] C:\Users\LGT\Desktop\eclipse\plugins\org.eclipse.
Some string
5
true
```

Command Line



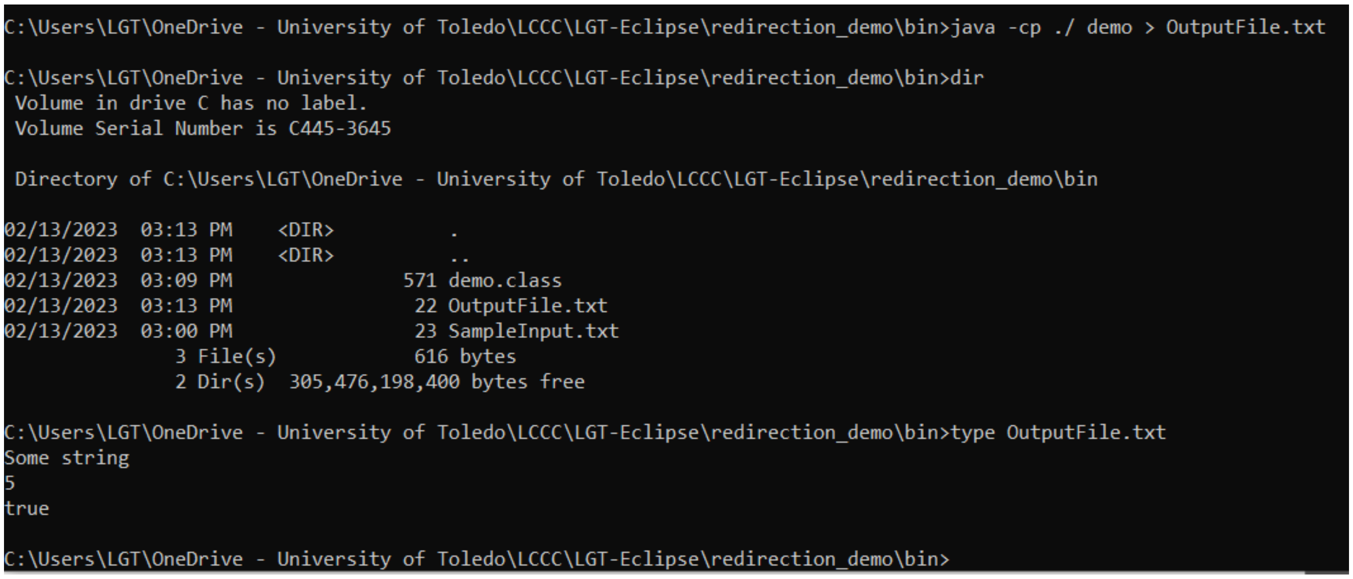
```
C:\Users\LGT\OneDrive - University of Toledo\LCCC\LGT-Eclipse\redirection_demo\bin>java -cp ./ demo
Some string
5
true

C:\Users\LGT\OneDrive - University of Toledo\LCCC\LGT-Eclipse\redirection_demo\bin>
```

If we want the output to go to a *file*, instead of the console (re-directing the standard output to a file), we can run this from the command line with:

`java -cp ./ demo > OutputFile.txt`

Now when the program runs, we see no output at all – rather than going to the screen, it goes to the text file!



```
C:\Users\LGT\OneDrive - University of Toledo\LCCC\LGT-Eclipse\redirection_demo\bin>java -cp ./ demo > OutputFile.txt

C:\Users\LGT\OneDrive - University of Toledo\LCCC\LGT-Eclipse\redirection_demo\bin>dir
 Volume in drive C has no label.
 Volume Serial Number is C445-3645

 Directory of C:\Users\LGT\OneDrive - University of Toledo\LCCC\LGT-Eclipse\redirection_demo\bin

02/13/2023  03:13 PM    <DIR>          .
02/13/2023  03:13 PM    <DIR>          ..
02/13/2023  03:09 PM               571 demo.class
02/13/2023  03:13 PM                22 OutputFile.txt
02/13/2023  03:00 PM                23 SampleInput.txt
               3 File(s)            616 bytes
               2 Dir(s)  305,476,198,400 bytes free

C:\Users\LGT\OneDrive - University of Toledo\LCCC\LGT-Eclipse\redirection_demo\bin>type OutputFile.txt
Some string
5
true

C:\Users\LGT\OneDrive - University of Toledo\LCCC\LGT-Eclipse\redirection_demo\bin>
```

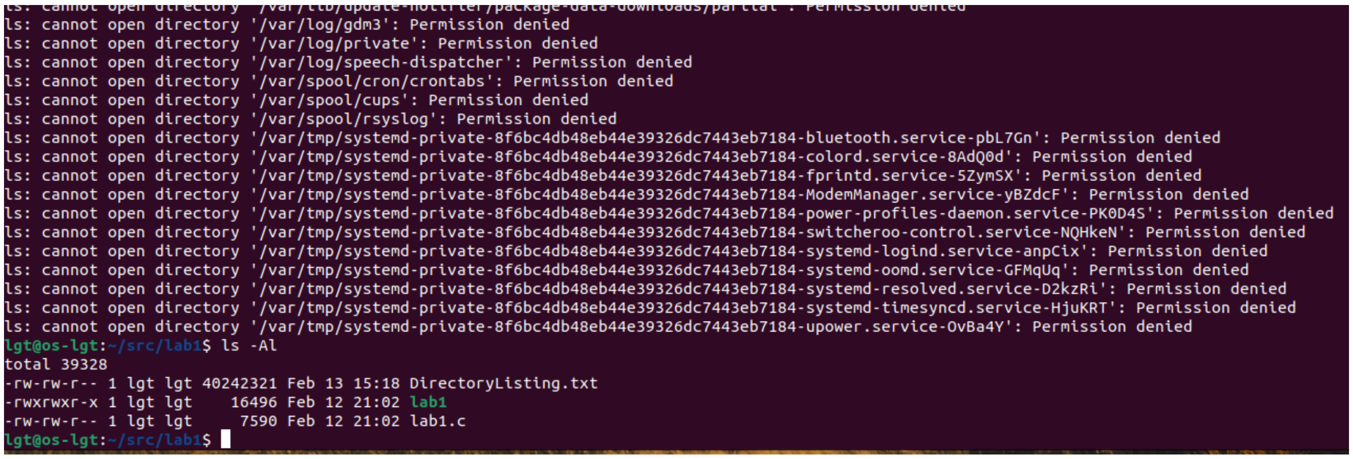
So, what does this long aside into redirection in Java and Windows have to do with Linux? Well, Windows borrowed the ability to redirect input and output from UNIX! So, we can use a command that will take a while to run (and generate a lot of output) and redirect that output to a file, rather than having it clutter up the screen:

`ls -AlR / > DirectoryListing.txt`

This is all well and good, except for two things:

1) It generates a 40+ megabyte text file we may not want to have to keep (or remember to delete each time)!
2) There are a LOT of directories we don't have permissions to, and we get error messages on those.

Here's a screen shot of the last part of the output from my **`ls`** command for the whole tree, and another to show **`DirectoryListing.txt`** (notice the **`40,242,321`**-byte listing file):



```
ls: cannot open directory '/var/lib/update-notifier/package-data-downloads/partial': Permission denied
ls: cannot open directory '/var/log/gdm3': Permission denied
ls: cannot open directory '/var/log/private': Permission denied
ls: cannot open directory '/var/log/speech-dispatcher': Permission denied
ls: cannot open directory '/var/spool/cron/crontabs': Permission denied
ls: cannot open directory '/var/spool/cups': Permission denied
ls: cannot open directory '/var/spool/rsyslog': Permission denied
ls: cannot open directory '/var/tmp/systemd-private-8f6bc4db48eb44e39326dc7443eb7184-bluetooth.service-pbL7Gn': Permission denied
ls: cannot open directory '/var/tmp/systemd-private-8f6bc4db48eb44e39326dc7443eb7184-colord.service-8AdQ0d': Permission denied
ls: cannot open directory '/var/tmp/systemd-private-8f6bc4db48eb44e39326dc7443eb7184-fprintd.service-5ZymSX': Permission denied
ls: cannot open directory '/var/tmp/systemd-private-8f6bc4db48eb44e39326dc7443eb7184-ModemManager.service-yBZdcF': Permission denied
ls: cannot open directory '/var/tmp/systemd-private-8f6bc4db48eb44e39326dc7443eb7184-power-profiles-daemon.service-PK0D4S': Permission denied
ls: cannot open directory '/var/tmp/systemd-private-8f6bc4db48eb44e39326dc7443eb7184-switcheroo-control.service-NQHkeN': Permission denied
ls: cannot open directory '/var/tmp/systemd-private-8f6bc4db48eb44e39326dc7443eb7184-systemd-logind.service-anpCix': Permission denied
ls: cannot open directory '/var/tmp/systemd-private-8f6bc4db48eb44e39326dc7443eb7184-systemd-oomd.service-GFMqUq': Permission denied
ls: cannot open directory '/var/tmp/systemd-private-8f6bc4db48eb44e39326dc7443eb7184-systemd-resolved.service-D2kzRi': Permission denied
ls: cannot open directory '/var/tmp/systemd-private-8f6bc4db48eb44e39326dc7443eb7184-systemd-timesyncd.service-HjuKRT': Permission denied
ls: cannot open directory '/var/tmp/systemd-private-8f6bc4db48eb44e39326dc7443eb7184-upower.service-OvBa4Y': Permission denied
lgt@os-lgt:~/src/lab1$ ls -Al
total 39328
-rw-rw-r-- 1 lgt lgt 40242321 Feb 13 15:18 DirectoryListing.txt
-rwxrwxr-x 1 lgt lgt    16496 Feb 12 21:02 lab1
-rw-rw-r-- 1 lgt lgt     7590 Feb 12 21:02 lab1.c
lgt@os-lgt:~/src/lab1$
```

There are two *more* things we need to talk about in order to solve these problems.

First, we can use a "special" device on Linux to send out output to, called the **`NULL`** device, which will just throw the output away (sometimes known as "the bit bucket"). Whenever we write to the **`NULL`** device, the output just disappears. Just as our main disk drive is **`/dev/sda`**, the **`NULL`** device is **`/dev/null`**, so we can direct the output of the **`ls`** command *to* it with:

```
ls -AlR / > /dev/null
```

Second, there's a ***third*** data stream associated with programs on Linux (and Widows). There's "standard input", "standard output", and there's *also* "standard error" (the three are known as **`stdin`**, **`stdout`**, and **`stderr`**). Error messages (like "permission denied" and messages we get when a program crashes) always go to **`stderr`**. By default, **`stderr`** goes to the console, just like **`stdout`**. So, when we used **`>`** to redirect the **`ls`** output above to either a file or the **`NULL`** device, it *did* redirect **`stdout`**, but it did not pick up **`stderr`** output.

These three "files" have *file descriptors* of 0, 1, and 2, respectively. Think of them as "abbreviations" or "references" / "pointers" to those three files. Technically, these two are equivalent:

```
ls -AlR / > /dev/null
ls -AlR / 1> /dev/null
```

Both redirect **`stdout`** (file descriptor #1) to the **`NULL`** device ("thin air").

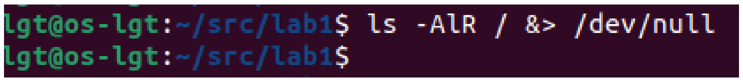
Suppose we want **`stdout`** to be redirected to **`DirectoryList.txt`** and **`stderr`** to go to **`ErrorList.txt`**. We can do that by specifying **`1>`** for **`stdout`** and **`2>`** for **`stderr`**:

```
ls -AlR / 1> DirectoryList.txt 2> ErrorList.txt
```

Now the directory listing (**`ls`**) runs, the "real" output goes to one text file, and any error messages go to the other. Just as the default is to have both **`stdout`** and **`stderr`** go to the console, we can redirect *both* to the same location with the shortcut **`&>`**

At long last, we can put all of this together, and have the ls command run a directory listing of everything on the system, discarding both the output *and* any error messages with the command:

```
ls -AlR / &> /dev/null
```



```
lgt@os-lgt:~/src/lab1$ ls -AlR / &> /dev/null
lgt@os-lgt:~/src/lab1$
```

It *ran* (for several seconds), but didn't generate any *visible* output (it generated plenty, but it was thrown away).

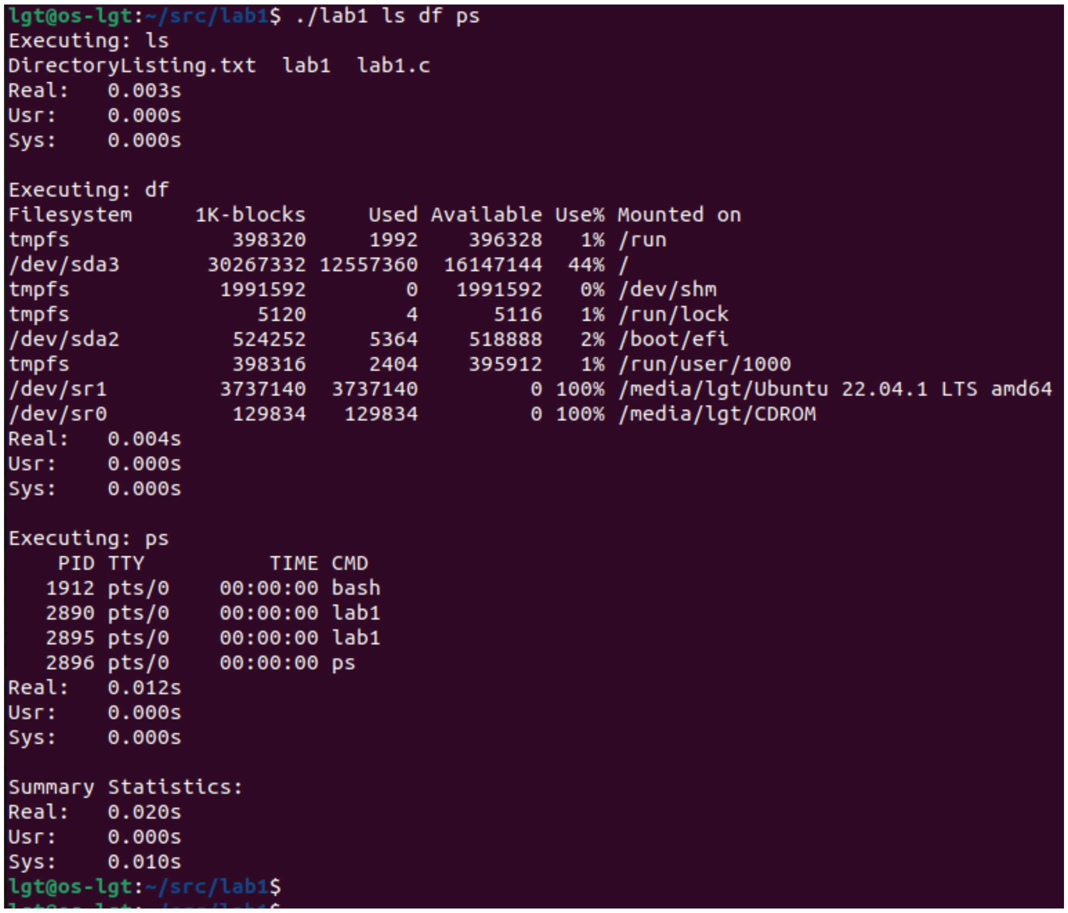
In the interest of tying all of this together, it should be obvious now that what the pipe (**|**) operation does is to start two programs, with **`stdout`** of the first one redirected to **`stdin`** of the next!

Now, back to the specifics of your programming project…

Your program will accept any number of parameters, each of which is a command to execute. If it's a command with no switches like **`df`**, **`ls`**, or **`ps`**, then it doesn't need to be in quotes. If any of the commands *do* include switches, then the whole command must be in double quotes:

```
./lab1 ls df ps
./lab1 "ls -Al" "df --all"
./lab1 "ls -AlR / &> /dev/null"
```

Your program will run these commands consecutively (left-to-right), and display the command before it is executed, and the **`times`** values from each (upon completion of each, of course), along with totals after the last one has completed. Here is a sample run, in which **`ls`**, **`df`**, and **`ps`** were all executed:



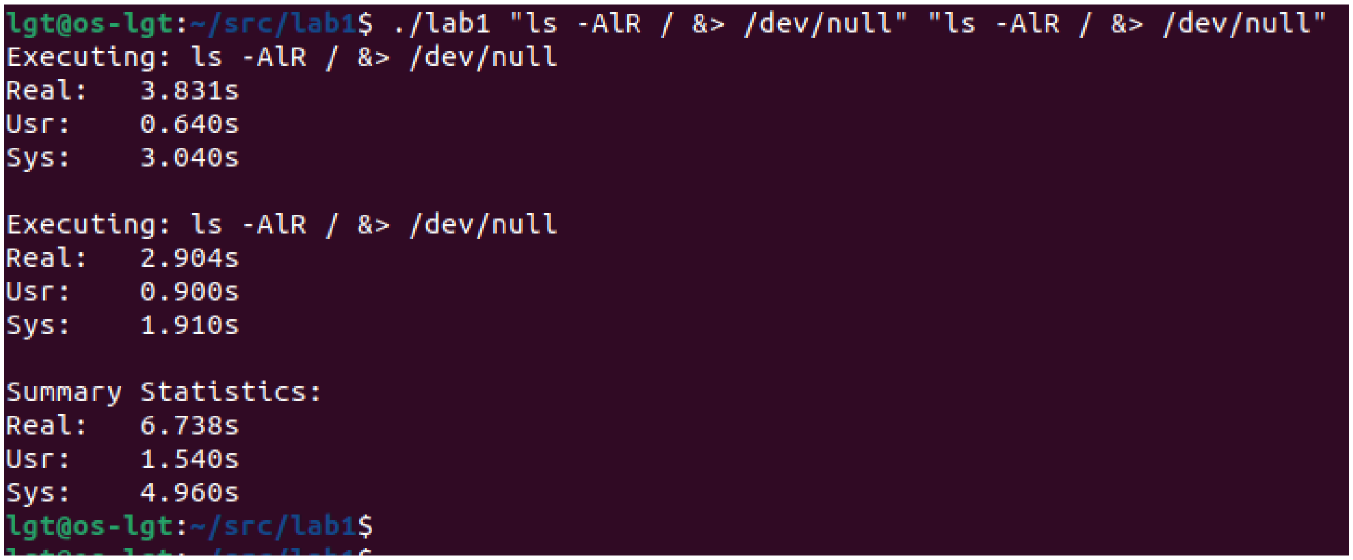
```
lgt@os-lgt:~/src/lab1$ ./lab1 ls df ps
Executing: ls
DirectoryListing.txt  lab1  lab1.c
Real:   0.003s
Usr:    0.000s
Sys:    0.000s

Executing: df
Filesystem     1K-blocks     Used Available Use% Mounted on
tmpfs             398320     1992    396328   1% /run
/dev/sda3       30267332 12557360  16147144  44% /
tmpfs            1991592        0   1991592   0% /dev/shm
tmpfs               5120        4      5116   1% /run/lock
/dev/sda2         524252     5364    518888   2% /boot/efi
tmpfs             398316     2404    395912   1% /run/user/1000
/dev/sr1         3737140  3737140         0 100% /media/lgt/Ubuntu 22.04.1 LTS amd64
/dev/sr0          129834   129834         0 100% /media/lgt/CDROM
Real:   0.004s
Usr:    0.000s
Sys:    0.000s

Executing: ps
    PID TTY          TIME CMD
   1912 pts/0    00:00:00 bash
   2890 pts/0    00:00:00 lab1
   2895 pts/0    00:00:00 lab1
   2896 pts/0    00:00:00 ps
Real:   0.012s
Usr:    0.000s
Sys:    0.000s

Summary Statistics:
Real:   0.020s
Usr:    0.000s
Sys:    0.010s
lgt@os-lgt:~/src/lab1$
```

If you use redirection to run long commands, you may see longer times, but no output:



```
lgt@os-lgt:~/src/lab1$ ./lab1 "ls -AlR / &> /dev/null" "ls -AlR / &> /dev/null"
Executing: ls -AlR / &> /dev/null
Real:   3.831s
Usr:    0.640s
Sys:    3.040s

Executing: ls -AlR / &> /dev/null
Real:   2.904s
Usr:    0.900s
Sys:    1.910s

Summary Statistics:
Real:   6.738s
Usr:    1.540s
Sys:    4.960s
lgt@os-lgt:~/src/lab1$
```

Don't be surprised if your total times don't add up *exactly* – the main (parent) process will take some time that the child processes don't. In this example, the two commands have a combined real time of 6.735 (3.831 + 2.904) seconds, but the displayed summary time is slightly longer at 6.738. The other apparent 3 ms went into operations in the parent process and to roundoff.

You are to display each command just before executing it, and then the times for that command and a blank line, before going on to the next one (or the summary). Use the same numerical format as I have – three digits after decimal point, with a unit of "s" (seconds), and *two* digits before the decimal. The system command **`time`** displays the time in minutes and seconds, but don't use that format – stick with what I have above.

Finally, in order to save everyone a lot of headaches (and a lot of e-mails for me to answer), the command you will need to execute via **`execve`** is, surprisingly enough, NOT the command the user entered!

Let's take a simple example – if the command to execute "**`ls`**", it seems like you should be able to just call **`execve("ls", NULL, NULL)`**. Look closely at the documentation for **`execve`**, however, and you will find that it "executes the program referred to by [the first parameter]". The program "**`ls`**" is actually located in **`/usr/bin`** (much like many of Windows' supporting programs are located in **`C:\Windows\System32`**). You *could* get it work by calling **`execve`** with **`/usr/bin/ls`** as the first parameter, but not *every* command is located there, so hard-coding the path won't work.

Rather, the way around this is to have **`execve`** start a new (**`bash`**) shell (and **`bash`** *is* located in **`/bin`**), and pass the command to **`bash`** as a parameter. The other thing that you need to do is to start **`bash`** with the "**`-c`**" option (see **`man bash`**), so if your command is **`ls`**, you'll actually be having **`execve`** run **`/bin/bash -c ls`**. The **`bash`** shell can take care of things like finding the path for you, so we'll have our process start **`bash`**, so that **`bash`** can actually execute the command.

The third parameter to **`execve`** is an array of environment strings. These strings are of the form **`<variable>=<value>`**. On a Windows machine, you can see the environment strings by using the **`set`** command. On Linux, the **`env`** command does the same thing.

When you first learned about C, you were probably taught that the function header for **`main`** is:

```
int main()
```

Then, when you learned about command-line arguments, you learned that the full function header is:

```
int main(int argc, char* argv[])
```

Well, it turns out that there's one *more* overloaded version of **main**:

```
int main(int argc, char* argv[], char* env[])
```

That third version allows your program to see all of the environment variables – all of those lines of output you got from **set** or **env** appear as the string array **env[]**. So, use this version of the method header for **main**, and simple pass **env** to the call to **execve**.

In terms of resources you might need for this project…

You will need to call **fork**, some form of **wait** (see the documentation), **times**, and **gettimeofday**. For times that are measured in "clock ticks", you will need to figure out how many clock ticks there are per second (there's a system call for that, but I don't want to give too much away). For times measured in seconds and microseconds, you should probably convert them to **double**s.

This is NOT a lot of code to write. I took out all of my comments and blank lines, and it was less than 70 lines, but don't let that lull you into a false sense of security. This is some intricate code, in that one program is (simultaneously) the parent, the child, and creates a grandchild to tend to. If you're not familiar with writing system-level code, you will likely spend a lot of time reading the documentation. I've given you a head-start by pointing out some of the calls you will need. If you think you need *other* calls, contact me before you go very far – this is not a project to decide to "just do your own way".

General requirements:

As always, your code must be yours and yours alone. You are not allowed to use code from any source other than the textbook or what I provide (including online sources). You are certainly welcome (and encouraged) to use the **man** pages for system-level documentation.

You are to submit your source code (which must be called **lab1.c**) and your executable (**lab1**). Copy both files to your host machine's shared folder, and create a 7-zip from there to submit. Your 7-zip must be named **<LastName><comma><space><FirstName>.7z** as in **Smith, John.7z**. If your code won't compile, it will get zero points.

Your code must successfully build (no error messages and no warnings) with the command:

```
gcc -o lab1 -Wno-nonnull lab1.c
```

Test your code with a variety of command lines – one command, multiple commands, commands with redirected output, etc. *I* will be testing your code with a variety of command lines; don't just try one and call it "done" when that one case works.

If you get stuck, and you've re-read the assignment, the **man** pages, the text, and still aren't making any progress, don't just *stay* stuck – send me an e-mail. As I said on the first day of class, I'm here to help you through these landmines, but I can't help with problems I don't know about, and you can't afford to lose much time just being stuck. There's another project coming right after this one.