

# CSC 360 Summer 2023

## Assignment 1

### Programming Environment

For this assignment, your code must work in the JupyterHub environment (<https://jhub.csc.uvic.ca>). Even if you already have access to your own Unix system, we recommend you work as much as possible with the JupyterHub environment. Bugs in systems programming tend to be platform-specific and something that works perfectly at home may end up crashing on a different computer-language library configuration.

### Individual Work

This assignment is to be completed by each individual student (i.e. no group work). You are encouraged to discuss aspects of the problem with your classmates, however **sharing of code is strictly forbidden**. If you are unsure about what is permitted or have other questions regarding academic integrity, please ask the instructor. Code-similarity tools will be run on submitted programs. Any fragments of code found or generated on the web and is used in your solution must be properly cited (citation in the form of a comment in your code).

### Assignment Objectives

1. Write a program `getstats.c` which prints out certain statistics about the environment in which it is run or statistics about a single process running in that environment.
2. Write a program `gopipe.c` that executes up to four shell instructions, piping the output of one to the input of the next.

### Submission

Submit your `getstats.c` and `gopipe.c` containing your programming solutions to Brightspace.

## Part 1: `getstats.c`

Every Unix file system includes a `/proc` directory. This directory contains information in the form of text files that relate to the properties of the operating system. For example its memory usage, its kernel version number, scheduling and file system information, and so on. Details about processes running on the system are also available in subdirectories of the `/proc` directory.

### Your Task

A basic starter file (`getstats.c`) has been provided for you. Modify the file so that the resulting program has the following behaviour:

- If run without arguments, the program should print to the console the following information, in order (one item per line):
  1. The model name of the CPU
  2. The number of cores
  3. The version of Linux running in the environment
  4. The total memory available to the system in kilobytes
  5. The uptime (the amount of time the system has been running) expressed in terms of days, hours, minutes, and seconds.
- If run with a numerical argument, the program should print the console information about the process with number corresponding to the given argument:
  1. The process number
  2. The name of the process
  3. The console command that started the process, if any (this may look like the filename or directory to filename)
  4. The total number of threads running in the process
  5. The total number of context switches that have occurred during the running of the process

If there is no running process that corresponds to the numerical argument provided, the program should quit with an error message.

The program should terminate once it has finished printing the appropriate information.

### Examples

If I execute the command `./getstats` on JupyterHub, my output is:

```
model name      : Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
cpu cores       : 8
Linux version 4.15.0-169-generic (buildd@lcy02-amd64-015) (gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04
)) #177-Ubuntu SMP Thu Feb 3 10:50:38 UTC 2022
MemTotal:       131942328 kB
Uptime: 433 days, 5 hours, 26 minutes, 43 seconds
```

Your output may vary from this (for example, uptime will vary from one second to the next), but the expected output above should serve as a guide for your work. The exact formatting of long lines (such as Linux version above) will also vary depending on console width and font settings.

If I execute the command `./getstats 1` on JupyterHub, my output is:

```
Process number: 1
Name:   tini
Filename (if any): tini
Threads:   1
Total context switches: 2122
```

Again, your output may vary from this, depending on the actual contents of `/proc/1` at the time you run the program.

Finally, if I execute the command `./getstats -1`, my output is:

```
Process number -1 not found.
```

Note that your solution must work for process numbers other than 1 and -1.

## Hints

- All the information you need is in the form of text files in the `/proc` directory or one of its subfolders. Remember to use `fopen()` and `fclose()` properly.
- You may need to read strings from these files and extract information from them in order to produce the statistics you need. The libraries that have been included into the starter `getstats.c` file may help you with this.

## Restrictions

- Your solution must print information that reflects the actual contents of `/proc` at the time your program is run. Solutions that simply hard-code their output will receive a 0.
- You should not need to use dynamic memory (`malloc`, `free`, etc) for this part of the assignment. If you do choose to use dynamic memory, you may lose marks if your program leaks memory or performs undefined memory operations (double frees, dereferences of NULL pointers, etc).
- Your solution must close any file pointers that result from calls to `fopen()`.
- Your source code must be compiled with the following command: `gcc -Wall -Werror -std=c18 -o getstats getstats.c` If your code cannot be compiled due to syntax errors or warnings, it may receive a 0.

## Part 2: gopipe.c

Unix commands that are executed in a shell can take advantage of piping; that is, the output of one command can be used as the input of another. For example, the command `ls -1` (**the number 1**) will list the contents of the current directory, one per line. The command `wc -l` (**the letter l**) counts the number of lines in the input, and outputs that number as an integer. We can specify that the output of `ls -1` should be used as the input of `wc -l` by joining the two commands with a pipe:

```
ls -1 | wc -l
```


The resulting compound command prints the number of entries in the current directory.




## Your Task

Write a program (`gopipe.c`) that accepts up to four individual commands, one per line (pressing enter or return at the end of each). The user may indicate that they have no more commands to enter by pressing return on a line without entering a command first. If this happens, or if the user enters four commands, your program must execute each of the instructions, transferring the output of one to the input of the next. The output of the final command in the sequence must be sent to the console.

For example, if the program is run as follows:

```
./gopipe
```

and then the following is entered (where the  symbol indicates pressing enter):

```
/bin/ls -l   
/bin/wc -l   

```

The program should print out the number of directory entries in the current directory (as an integer) and then quit.

The program should only execute as many instructions as were entered (a maximum of four) - if the enter key is pressed without first entering a command, the program should simply quit. If the user enters four commands, the program should not prompt for a fifth, but should immediately begin executing the commands.

Test your program with inputs of various lengths - chains of zero, one, two, three, or four commands. The console output in your program should be identical to what you get at the shell prompt if you enter the same commands separated by pipes.

## Example Output





The shell command

```
gcc --help | grep dump | tr '[:lower:]' '[:upper:]' | sort
```

prints all three lines from the gcc help that contain the word “dump”, converted to uppercase, and sorted alphabetically:

-DUMPMACHINE	DISPLAY THE COMPILER'S TARGET PROCESSOR.
-DUMPSPECS	DISPLAY ALL OF THE BUILT IN SPEC STRINGS.
-DUMPVERSION	DISPLAY THE VERSION OF THE COMPILER.

Running `./gopipe` and entering

```
/usr/bin/gcc --help   
/usr/bin/grep dump   
/usr/bin/tr '[:lower:]' '[:upper:]'   
/usr/bin/sort 
```

should have exactly the same output.

Note that your solution must work for commands other than the examples shown above.

## Hints

- You may assume that no command will have more than seven arguments. That is, each line of input will have a maximum of eight tokens separated by whitespace.
- You may assume that the total length of each input will not be greater than 80 characters.
- You do not need to account for incorrectly entered commands. We will only test your code with correctly entered instructions.
- You will need to use the `fork()` and `pipe()` functions, and `strtok()` will be helpful when it comes to parsing input.
- Remember that you'll need to provide the full paths for each of the commands. That is, only entering `ls` won't work; you will need to enter `/bin/ls`. You can use the `which` command to locate where utilities are located in the file system. For example, `which ls` tells you where the `ls` utility is.
- You are not allowed to use the `<stdio.h>` functions in this part of the assignment. So, to read input from the user, use the `read()` system call with the file-descriptor 0.

## Restrictions

- **You may not call any `stdio.h` library functions in your solution to `gopipe.c`.** The libraries that have been included in the starter file are all the libraries you may use. You may use `stdio.h` functions while debugging, **but any calls to those functions must be commented out of your final submission.**
- You should not need to use dynamic memory (`malloc`, `free`, etc) for this part of the assignment. If you do choose to use dynamic memory, you may lose marks if your program leaks memory or performs undefined memory operations (double frees, dereferences of NULL pointers, etc).
- Your source code file must be compiled with the following command: `gcc -Wall -Werror -std=c18 -o gopipe gopipe.c` If it cannot be compiled due to syntax errors or warnings, it may receive a 0.