

Assignment 1: Introductory C

Due Sep 26, 2018 by 4pm **Points** 5 **Available** after Sep 14, 2018 at 6:30pm

Assignment A1: Introductory C

Introduction

For this assignment, you will be writing two small command-line programs: one that verifies a solution to a regular 9x9 sudoku puzzle and one that determines which user is running the most processes on a unix system. The first will give you lots of practice with arrays in C. The second will require that you use command-line arguments and that you process standard input (using `scanf` or `sscanf`).

Part 0: Getting Started (0%)

Follow the instructions carefully, so that we receive your work correctly.

Your first step should be to log into [MarkUs](https://markus.teach.cs.toronto.edu/csc209-2018-09/) [\(https://markus.teach.cs.toronto.edu/csc209-2018-09/\)](https://markus.teach.cs.toronto.edu/csc209-2018-09/) and navigate to the **A1: Intro C** assignment. Like for the labs, this triggers the starter code for this assignment to be committed to your repository. You will be working alone on this assignment, and the URL for your Git repository can be found on the right hand side of the page.

Pull your git repository. There should be a new folder named A1. All of your code for this assignment should be located in this folder. Starter code, including a Makefile that will compile your code and provide a sanity check, has already been placed in this folder. To use the Makefile, type `"make test_part1"` or `"make test_part2"`. That will compile the corresponding program and run our sanity checks. To remove .o files and the executable, type `"make clean"`.

Once you have pushed files to your repository, you can use MarkUs to verify that what you intended to submit was actually submitted. The Submissions tab for the assignment will show you what is in the repository, and will let you know if you named the files correctly.

Part 1: Checking a Sudoku Puzzle (50%)

You will complete a C program that could be used to check whether a completed sudoku puzzle is valid. You can learn about sudoku puzzles from [this website](https://en.wikipedia.org/wiki/Sudoku) [_\(https://en.wikipedia.org/wiki/Sudoku\)_](https://en.wikipedia.org/wiki/Sudoku). For our terminology, we will consider a "regular" sudoku to be a 9x9 board made up of 9 inner 3x3 boxes.

Details


Your task is to write two functions. The first, `check_regular_sudoku`, can be called by a main function in a program to determine if a 9x9 sudoku board is valid. `check_regular_sudoku` calls `check_group` 27 times in


order to check each row, each column and each of the nine inner boxes. You must write the functions according to the specifications we give. You must not change the signatures and you must use only `check_group` to do the 27 checks.

check_group

The `check_group` function has the following signature and function comment:

```
/* Each of the n elements of array elements, is the address of an
 * array of n integers.
 * Return 0 if every integer is between 1 and n^2 and all
 * n^2 integers are unique, otherwise return 1.
 */
int check_group(int **elements, int n)
```

Read the description carefully. Notice first that the first parameter is an array of `int *` and that each of these pointers, points to another array of integers. Those are the elements. [fig1.pdf](#)  is a picture of one possible value for elements where `n` is set to 3.

Now notice that the second parameter `n` is the size of the arrays and that `n` isn't necessarily 3. This means that the function could be used to test more than just regular 9x9 sudoku puzzles. [Here](#)  is a picture of a 16 x 16 puzzle. Think about how you would call `check_group` 27 times on this puzzle to decide if it is a valid Sudoku.

check_regular_sudoku

The second function `check_regular_sudoku` only works on conventional 9x9 puzzles. Again the input is an array of pointers, each to an array of integers.

```
/* puzzle is a 9x9 sudoku, represented as a 1D array of 9 pointers
 * each of which points to a 1D array of 9 integers.
 * Return 0 if puzzle is a valid sudoku and 1 otherwise. You must
 * only use check_group to determine this by calling it on
 * each row, each column and each of the 9 inner 3x3 squares
 */
int check_regular_sudoku(int **puzzle)
```

You **must** use `check_group` nine times to check the nine rows, nine times to check the nine columns and nine times to check the nine inner boxes. You may not just check the rows or columns directly inside `check_regular_sudoku` without calling `check_group`. We realize that this isn't efficient, but the point of this assignment is to practice using arrays and this should provide you with lots of opportunity to do this.

Starter code for Part 1

We have provided two files `sudoku.c` and `sudoku_helpers.c`. The main function is in `sudoku.c` and you are welcome to change it to test your functions. Of course your functions should still work with the original version. You are only required to complete the two helper functions in `sudoku_helpers.c` and we will be testing them with our own different main function.

You compile this code with:

```
gcc -Wall -std=gnu99 -o sudoku sudoku.c sudoku_helpers.c
```

Notice that it compiles without error or warning and even runs. It just doesn't work properly. It is your job to complete the helper functions so that it works correctly.

Part 2: most_processes.c (50%)

Your second task is to write a C program called `most_processes`. The program will inspect the output from the `ps -ef` command and print the user who owns the most processes and how many processes this is.

Details

The `ps` (process status) command generates a report on the current running processes. It has a dazzling set of options and works differently on different versions of unix. Your program must work with the format of the report (i.e. the columns and their order) produced on the teaching lab machine (teach.cs). Notice, that unlike most unix commands, the output from `ps` includes a header that identifies these columns. This is helpful for you to understand the output, but adds an extra small complication when you are generating test input.

The task for your program is to print to **standard output**, the UID field for the user who owns the most processes followed by one space followed by the number of those processes. Your program may be called with an optional command-line argument which represents a process id `ppid`. If this argument is provided, then the program only considers processes whose parent is this provided `ppid`.

From **standard input**, your program reads input in **almost** the same format as produced by the `ps` program with the `-ef` option when run on the teaching lab machines. The difference is that the input will not have a header and it will always be in sorted order by UID.

Consider the following input (which is provided as the file `handout.example.input` in the starter code:

```
ange      7264      1  0 09:23 ?        00:00:03 /lib/systemd/systemd --user
ange      7274    7264  0 09:23 ?        00:00:00 (sd-pam)
c4ntest   45909      1  0 11:07 ?        00:00:01 /lib/systemd/systemd --user
c4ntest   45919  45909  0 11:07 ?        00:00:00 (sd-pam)
ccc       34911      1  0 Sep09 ?        00:00:53 /lib/systemd/systemd --user
ccc       34921  34911  0 Sep09 ?        00:00:00 (sd-pam)
ccc       34933  34906  0 Sep09 ?        00:00:00 sshd: ccc@notty
ccc       34944  34940  0 Sep09 pts/4    00:00:00 -csh
daemon    1339      1  0 Aug28 ?        00:00:00 /usr/sbin/atd -f
daemon    1567      1  0 Aug28 ?        00:28:31 lpd Waiting
```

```
mrcraig 4044 4026 0 12:10 ? 00:00:00 sshd: mrcraig@pts/6
mrcraig 4045 4044 0 12:10 pts/6 00:00:00 -bash
mrcraig 4912 4045 0 12:12 pts/6 00:00:00 ps -ef
mrcraig 4913 4045 0 12:12 pts/6 00:00:00 sed 1d
mrcraig 4914 4045 0 12:12 pts/6 00:00:00 sort
```

Your program called with no command-line argument on this standard input will produce the output:

```
mrcraig 5
```

and when called with a command-line argument of 1 on the same standard input will produce:

```
daemon 2
```

When called with a command-line argument of 2 on the same standard input, your program will produce no output since there are no processes at all with a PPID of 2. The program should also produce no output if it is called with an empty file for standard input. In the event of a tie, the program should report any one of the tied winners.

Since your program reads from standard input, it would be possible to type all the input to your program from the keyboard. But typing in a report that looks like the output from `ps -ef` even once would be awful. Instead you could run `ps -ef` and **redirect** its output to another file. Then you could delete the header line and sort the file. But better than that, you could use a pipeline to run `ps`, remove the header, sort it and save it all in a single command as follows:

```
ps -ef | sed 1d | sort > testinput
```

(When called with the argument `1d`, `sed` reads from standard input, removes the first line and sends the remaining lines to standard output.) Next you could edit `testinput` to create different versions that thoroughly test your program. Finally you could run your program and **redirect** standard input to read from one of the test input files that you just created.

Assumptions You May Make

For this assignment, you may assume the following:

1. If the user provides a command-line argument it will be a number.
2. The input read from standard input will have the correct format.
3. UID names will be at most 31 characters.
4. Each line in `ps` output is at most 1024 characters long including the newline character and a string terminator.

Command-line arguments and return codes

If the user specifies more than one command-line argument your program should print to standard error the message "USAGE: most_processes [ppid]" (followed by a single newline `\n` character), and return from main with return code `1`. Following the standard C conventions, main should return 0 when the program runs successfully.

String Functions

In order to complete this program, you will need to be able to compare two strings to see if they are equal and to copy characters from one string variable to another. Unfortunately, we won't have covered string functions in the videos before this assignment is due (and they aren't as simple as in Python where you can just compare string variables with `==` and assign them to each other.) We will briefly talk about string functions in lecture on Sept 18th and teach you just enough for what you need for this assignment. If you want to work ahead, you can read in the text book about strings and specifically the functions `strcmp` and `strcpy`.

Strong Suggestions

Don't try to use `strtok` to process the input. Instead use a function from the `scanf` family. Notice that the command (CMD) in the `ps -ef` report may contain spaces. That makes it slightly more difficult to parse with `scanf` although still possible. Another option is to read the line from the file with `fgets` and then parse it with `sscanf`. You can read about both these commands on their man pages. `sscanf` works just like `scanf` except that it takes its input from a string rather than from `stdin`. Hint: When you are developing your code, first write the bit that reads each line from `stdin` and assigns values to variables. Then before going on to use those values, print out their values to confirm that your parsing is working properly.

Do **not** save the entire listing in memory. That isn't necessary and is a very poor design. It will also make the program much harder to write.

Submission

We will be testing the following files from the A1 directory of your repository:

- `sudoku_helpers.c`
- `most_processes.c`

Remember that your final programs must compile without an error messages on `teach.cs` using the commands we have given in this handout and in the provided Makefile. Programs that do not compile, will get 0. You can still get part marks by submitting something that doesn't completely work but does some of the job -- but it **must at least compile** to get any marks at all.

Do not commit .o files or executables to your repository. If you do this, you are likely to find that the commit will work, but then git will refuse to accept the push and you will be required to revert your commit.

