CPSC 457: Principles of Operating Systems

Assignment 1: basic C++, make, time, strace, system calls

Weight: 15%

Collaboration

Discussing the assignment requirements with others is a reasonable thing to do and an excellent way to learn. However, the work you hand in must ultimately be your work. This is essential for you to benefit from the learning experience and for the instructors and TAs to grade you fairly. Handing in work that is not your original work but is represented as such is plagiarism and academic misconduct. Penalties for academic misconduct are outlined in the university calendar.

Here are some tips to avoid plagiarism in your programming assignments.

1. Cite all sources of code you hand in that are not your original work. You can put the citations into comments in your program. For example, if you find and use code found on a website, include a comment that says, for example:

   # The following code is from https://www.quackit.com/python/tutorial/python_hello_world.cfm.

   Use the complete URL so that the marker can check the source.

2. A tool like chat-GPT can be used to improve small code blocks. For example, three lines of code. If you get help from code assistance like Chat-GPT, you should comment above the block of code you requested assistance on debugging or improving and cite the tool used to get that suggestion. Using a tool like chat-GPT to write the majority of your assignment requirements will be treated as plagiarism if found without citation, and with citation, it will be treated as 0 for the component the student did not complete. Code improvement of short length will get credit if commented/cited properly.
3. Citing sources avoids accusations of plagiarism and penalties for academic misconduct. **However, you may still get a low grade if you submit code not primarily developed by yourself. Cited material should never be used to complete core assignment specifications. Before submitting, you can and should verify any code you are concerned about with your instructor/TA.**
4. Discuss and share ideas with other programmers as much as you like, but make sure that when you write your code, it is your own. A good rule of thumb is to wait 20 minutes after talking with somebody before writing your code. If you exchange code with another student, write code while discussing it with a fellow student, or copy code from another person's screen, this code is not yours.
5. **Collaborative coding is strictly prohibited.** **Your assignment submission must be strictly your code**. Discussing anything beyond assignment requirements and ideas is a strictly forbidden form of collaboration. This includes sharing code, discussing the code itself, or modelling code after another student's algorithm. **You can not use (even with citation) another student's code.**
6. Making your code available, even passively, for others to copy or potentially copy is also plagiarism.
7. We will look for plagiarism in all code submissions, possibly using automated software designed for the task. For example, see Measures of Software Similarity (MOSS - https://theory.stanford.edu/~aiken/moss/).
8. Remember, if you are having trouble with an assignment, it is always better to go to your TA and/or instructor for help rather than plagiarizing. A common penalty is an F on a plagiarized assignment.

Late Policy

All students will have 6 total days during the semester that can be used for none, one, some, or all the 6 assignments. Students can use these days all on one assignment, or some on each of the six.

Each 24-hour period late after an assignment deadline counts as one full day regardless of how many hours the assignment was late within that period. For example, deadlines are generally Fridays 11:59pm local time. That means an assignment submitted any time Saturday before 11:59pm local time will be considered as 1 day late and count against the students 5 total days.

As long as a student still has days left to use, their assignment will be graded without penalty. If a student has no more days left or their submission exceeds the days they have remaining, then they will receive a 0 grade for a late assignment. Students who use up late days on earlier assignments will not have them available for later assignments.

TAs will indicate the student's late day usage in grading feedback.

Due date is posted on D2L.

Description

Well written C++ code will usually outperform an equivalent Python code. However, a badly written C++ can easily run slower than Python code. A common reason why some C++ programs run slowly is due to inefficient use of system calls. You will be given two programs – one written in Python and the other one written in C++. They both find and print the longest **stutter** from text supplied via standard input. The Python version is well written and runs fast. The C++ version is not well written and runs slowly. In this assignment:

- You will analyze and compare the performance of the Python and the C++ program.
- You will improve the performance of the badly written C++ program, by modifying it to use system calls more efficiently.
- Finally, you will compare the performance of your new C++ implementation to the original Python and C++ programs.

Start by cloning the repository with starter code (type the following on the command line on the linux lab computers). You may need to update the repository if you pulled it before to see future assignment changes before starting an assignment:

```
$ git clone https://csgit.ucalgary.ca/jwhudson/cpsc457w24.git

$ cd cpsc457w24/stutter
```

The repository contains the following files:

stutter.py	Python program that reads in text from standard input and reports the longest **stutter** to standard output.
slow-stut.cpp	Inefficient C++ implementation of stutter.py. Feel free to re-use any part of this code in your solution.
fast-stut.cpp	This is where you will write your efficient implementation, which you then submit for grading.
Makefile	Makes compilation a bit easier.
t*.txt	Few sample test files.
dup.py	Python3 script that can generate big data (see appendix).

To remove ambiguity, we will use the following definitions for this assignment:

standard input	Please read http://www.linfo.org/standard_input.html **.**
white space	Any character that isspace() reports as white-space. Read the man page for isspace() or https://www.cplusplus.com/reference/cctype/isspace/ for more information.
word	Non-zero-length sequence of non-white-space characters delimited by white space, or beginning of file, or end of file.
stutter	Any word in which the first half is the same as the second half after ignoring the case. Examples of stutters: '**DiddiD**', '**0101**', '**xx**'
longest	Stutter with most characters. If multiple stutters have the same stutter maximum length, your program must report the first one.

Q1 – Written question (5 marks)

For this question you will compare the performance of the python program (**stutter.py**) to the C++ program (**slow-stut.cpp**) by using **time** and **strace** utilities. For example, to time **stutter.py** on the **t5.txt** file, execute this command:

```
$ time python stutter.py < t5.txt

Longest stutter: DetartrateDDetartrateD

real    0m0.034s

user    0m0.021s

sys     0m0.011s
```

To get a summary of all system calls made by **stutter.py**, run this:

```
$ strace -c python stutter.py < t5.txt

Longest stutter: DetartrateDDetartrateD
```

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 23.89    0.000984           3       290       108 newfstatat
 13.74    0.000566          21        26           mmap
 12.92    0.000532         532         1           execve
  9.49    0.000391           7        51         7 openat
  7.77    0.000320          17        18           getdents64
  7.53    0.000310           4        65           read
  4.56    0.000188           4        44           close
  4.47    0.000184           2        66           rt_sigaction
  2.79    0.000115           2        55         2 lseek
  2.35    0.000097          19         5           mprotect
  2.16    0.000089           2        34        29 ioctl
  1.99    0.000082           7        11           brk
  0.90    0.000037           9         4           munmap
  0.78    0.000032           6         5         3 readlink
  0.66    0.000027          13         2           pread64
  0.63    0.000026          13         2         1 arch_prctl
  0.49    0.000020          10         2           getrandom
  0.41    0.000017          17         1         1 access
  0.32    0.000013           6         2           getcwd
  0.32    0.000013          13         1           set_robust_list
  0.29    0.000012          12         1           set_tid_address
  0.29    0.000012          12         1           rseq
  0.24    0.000010           2         4           fcntl
  0.24    0.000010          10         1           futex
  0.22    0.000009           9         1           gettid
  0.22    0.000009           9         1           prlimit64
  0.15    0.000006           6         1           write
  0.05    0.000002           2         1           getuid
  0.05    0.000002           2         1           getgid
  0.05    0.000002           2         1           geteuid
  0.05    0.000002           2         1           getegid
  0.00    0.000000           0         1           sysinfo
------ ----------- ----------- --------- --------- ----------------
100.00    0.004119           5       700       151 total
```

The results above indicate that the **read()** system call was executed 65 times.

Before you can time the C++ code, you will need to compile it. The easiest way to compile it is using the included Makefile:

```
$ make
```

You can also compile it by hand, if you prefer. Don't forget the "-O2" option:

```
$ g++ -O2 -Wall slow-stut.cpp –o slow-stut
```

Now you can use '**time**' and '**strace -c**' on the resulting executable **slow-stut**:

```
$ time slow-stut < t5.txt
Longest stutter: DetartrateDDetartrateD
real    0m0.011s
user    0m0.004s
sys     0m0.006s
$ strace -c slow-stut < t5.txt
Longest stutter: DetartrateDDetartrateD
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 48.39    0.001828           8       215           read
 17.65    0.000667         667         1           execve
 16.94    0.000640          27        23           mmap
...
```

Answer the following questions:

a) Use the **time** utility to time **stutter.py** and **slow-stut.cpp** on files **t3.txt** and **t4.txt**. Copy/paste the output of **time** from the terminal window into your report.
b) How much time did the C++ and python programs spend executing on **the** CPU, and how much time did each of them spend waiting for **I/O** to finish?
c) Run '**strace -c**' on **stutter.py** and **slow-stut.cpp** on **t3.txt** and **t4.txt**. Copy/paste the output from the terminal window into your report.
d) When compared to the C++ code, why is the python program faster on some inputs, but slower on others? Try to justify your answers using the results you obtained above.

Q2 – Programming question (15 marks)

Your job is to improve **slow-stut.cpp** by writing a new implementation called **fast-stut.cpp**. Your new implementation should be faster than **slow-stut.cpp** and at least as fast as **stutter.py** for all possible inputs! Your new implementation must match the output of the slow implementation and the Python implementation. You may re-use any code from the **slow-stut.cpp** file.

Hints

The **slow-stut.cpp** makes too many calls to the **read()** system call, as it calls **read()** for every single character. You need to find a way to reduce the number of calls to read. I suggest you refactor the slow code so that **read()** is called with a buffer size of 1MB, i.e. you should read about 1 million bytes per system call. This should dramatically speed up your program.

The repository sub-directory **longest-int** contains a similar problem and the corresponding solution. Feel free to reuse any parts of this code in your own solution, but please include citations for the parts you reuse.

If you study and understand the above code, this assignment will be very easy!

Valid input

Your program must be able to handle any data on standard input. You may assume that no word will be longer than 1024 bytes. The files may or may not include a new line at the end.

A small number of test files are available in the GitLab repository, but it is expected that you create your own test files to help you validate your solutions. Use **stutter.py** to obtain correct outputs for your own test files.

Please note: we will grade your solution on inputs that are not published to you.

Requirements

- Your code must produce the same output as **slow-stut.cpp**
- Your code must run efficiently – e.g. on 2GB input, it should finish under 30s.
- You are **only allowed** to use the **read()** system call to read data from standard input.
- You **may not** use any other I/O APIs, such as **mmap(), fopen(), fread(), fgetc()**, C++'s **streams**, etc.
- Do not store the entire input in memory. You need to write your code so that it can handle any input size, even if it is bigger than the available memory.
- Your program must run on **linux lab** computers or **cslinux.ucalgary.ca**. You should test your code on the Linux workstations in the MS labs or use SSH to test it remotely.

Marking

Your code needs to be both correct, and efficient. Programs that output wrong results, or run very slowly, will receive 0 marks. On 2GB input your program should finish under 30s on **linux**

lab machines. Below are some timings I obtained using my own solution, to give you an idea of what you should be aiming for.

```
$ python dup.py 2000000000 < t4.txt | time fast-stut

Longest stutter: Tartar

35.60user 0.62system 0:36.76elapsed 98%CPU (0avgtext+0avgdata
3584maxresident)k

0inputs+0outputs (0major+171minor)pagefaults 0swaps


$ seq 101010101 | time -p ./fast-stut

Longest stutter: 10001000

real 11.80

user 11.45

sys 0.29
```

Q3 – Written question (5 marks)

a) Run your **fast-stut.cpp** on **t3.txt** and **t4.txt** files using **'time'** and **'strace -c**'. Copy/paste the output from the terminal window into your report.
b) Is your **fast-stut.cpp** faster than **slow-stut.cpp**? Why do you think that is?
c) Is your program faster than **stutter.py** and why?

Justify your answers for (b) and (c) by comparing the outputs of **'time'** and '**strace -c**'.

Submission

Submit two files for this assignment to D2L:

- Answers to the written questions Q1 and Q3 combined into a single file called **report.pdf**. You can also use **.docx** and **.txt** file format. Do not use any other file formats.
- Your solution to Q2 in a file called **fast-stut.cpp.**

Submit these as two separate files. **Do not** submit an archive, such as ZIP or TAR. If you submit an archive, you will receive a penalty.

General information about all assignments

All assignments are due on the date listed on D2L. Late submissions without remaining late days banked will not be marked.

1. Extensions beyond the late day policy can be discussed more than 5 business days in advance and are granted only by the course instructor.

2. After you submit your work to D2L, verify your submission by re-downloading it.
3. You can submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It is better to submit incomplete work for a chance of getting partial marks, than not to submit anything. Please bear in mind that you cannot re-submit a single file if you have already submitted other files. Your new submission would delete the previous files you submitted. So please keep a copy of all files you intend to submit and resubmit all of them every time.
4. Assignments are likely going to be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA, then you can contact your instructor.
5. All programs you submit must run on **linux lab** or **cslinux.ucalgary.ca.** If your TA is unable to run your code on these, you will receive 0 marks.
6. Unless specified otherwise, you must submit code that can finish on any valid input under 10s on **linux lab** or **cslinux.ucalgary.ca**, when compiled with **-O2** optimization. Any code that runs longer than this may receive a deduction, and code that runs for too long (about 30s) will receive 0 marks.
7. **Assignments must reflect individual work.** Here are some examples of what you are not allowed to do for individual assignments: you are not allowed to copy code or written answers (in part, or in whole) from anyone else; you are not allowed to collaborate with anyone; you are not allowed to share your solutions (code or pseudocode) with anyone; you are not allowed to sell or purchase a solution; you are not allowed to make your code available publicly (e.g. via public git repositories). This list is not exclusive. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k-5.html .
8. We will use automated similarity detection software to check for plagiarism. Your submission will be compared to other students (current and previous), as well as to any known online sources. Any cases of detected plagiarism or any other academic misconduct will be investigated and reported.

Appendix – dup.py utility (aka. Testing your code on large inputs)

Many of you probably do not have enough storage quota to store 2Gib test files in your accounts. There is a python script **dup.py** to make it possible to test your program on large inputs, without having to store big files.

dup.py is a simple python program that accepts a single command line argument "N", which indicates the number of bytes that the script will generate on standard output. **dup.py** reads in data from stdin, byte by byte, and outputs the data to stdout. It always outputs N bytes. If the data on stdin is bigger than N bytes, only the first N bytes are copied. If the data on stdin is shorter than N, the script will repeat the input data, until N bytes are generated. Example:

```
$ echo "hello." | python ./dup.py 10
```

```
hello.

hel
```

Here is an example of how to feed 2GB of data to your program, generated by repeating **t3.txt**:

```
$ python ./dup.py 2000000000 < t3.txt | ./fast-stut
```

Here is how you can time your code on the same data:

```
$ python./dup.py 2000000000 < t3.txt | time ./fast-stut
```

Here is how to run strace in combination with dup.py:

```
$ python./dup.py 2000000000 < t3.txt | strace -c ./fast-stut
```

Warning: If you follow the hints, this assignment is quite simple, as it requires minimum amount of coding. Please do not assume that future assignments will be this simple.

More hints

The code for reading a singular character using read() in slow_stut.cpp looks like this:

unsigned char buff;

read(STDIN_FILENO, & buff, 1);

The reason you include & is because read() requires a pointer to memory where to store the data it reads. If I change the definition of the buffer to be an array of chars

unsigned char buff[1024];

then the variable buff is already a pointer (it points to an array of chars). Now the call to read() must be made without the &. i.e.

read(STDIN_FILENO, buff, 1024);

It would be a mistake to write:

read(STDIN_FILENO, &buff, 1024);

Questions

My code crashes on large inputs, but works fine on smaller inputs. The error message looks like this: "Command terminated by signal 9." What is wrong?

You are likely trying to read all the input into memory at once. Try and debug to see the size of your inputs or input requests.

Why do I see errors like this when I use dup.py? “broken pipe”

Your code does not read all input from dup.py before your program exits. This means you have a bug in your code.