Nikesh Patel

Final CS3377

C/C++ Programming in a UNIX Environment

Date: 12/03/2018

Scoring: 100 points

**Problem 1: 25 points**

**UNIX, bash, compiled vs scripting**

**Problem 2: 25 points**

**Binary File I/O**

**Problem 3: 25 points**

**Fork, exec, and pipes**

**Problem 4: 25 points**

**Signals**

**Bonus 1: 10 points**

**Bonus 2: 10 points**

**Bonus 3: 10 points**

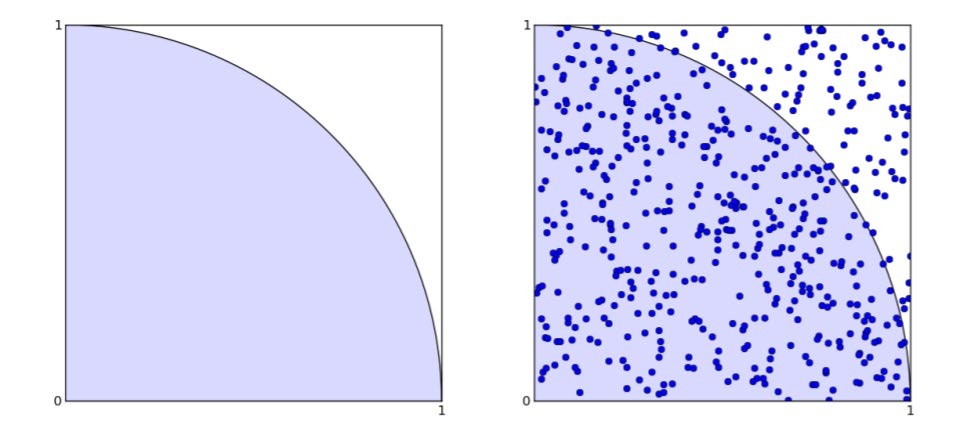
**Total:**

**Deliverables**:

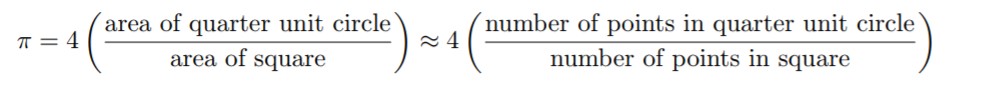
You are to submit a gzipped tarball of all your source code, makefiles and git log to eLearning. Also please include a README file that describes any bonus problems you’ve done and any other relevant notes to make my life easier.

**Problem 1: basic UNIX, bash scripting, compiled vs interpreted languages**

Supposed we want to compute an estimate of π. As you know, the area of a circle of radius (r) is A = π r2. A circle with r = 1 is called the unit circle and has an area π, so the area of one fourth of the unit circle is π / 4. The left side of the figure below shows a unit square, that is a square with sides of length one, with an “inscribed” quarter unit circle. The right side of the figure below is the same unit square with 500 random points plotted inside it.



The ratio of the number of points inside the quarter circle to the total number of points in the square should ideally be the same as the ratio of the area of the quarter circle to the area of the square. That is:

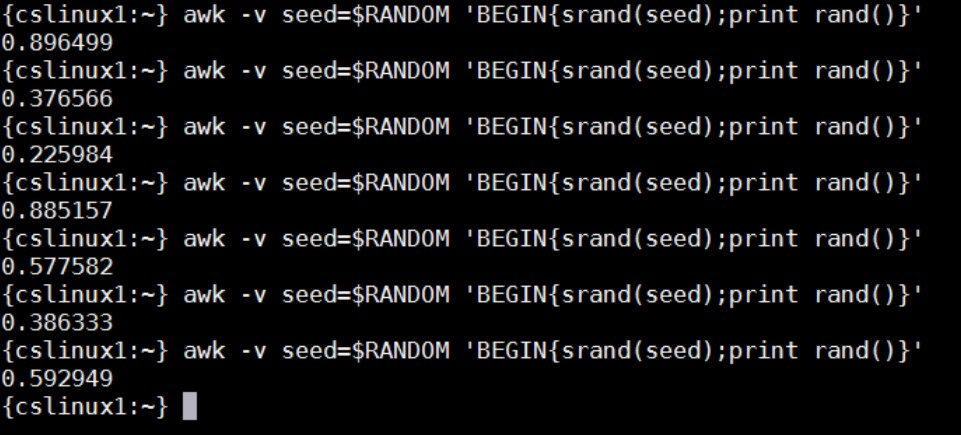


This gives us a way to estimate π. The more random points we generate, the better the

estimate.

Your assignment is to write a bash script and a C/C++ code to calculate π using the above information together with a random number generator to perform a “monte carlo” simulation.

For bash, you will need some way to generate random numbers. Using awk, we can generate (pseudo-)random numbers for our script.



For your C/C++ program, use GSL’s RNG.

Once both programs are running, fill out and submit the following table twice, once for the bash script and once for the compiled program.

Bash:

# points Time (s) Estimated value Deviation % error

10 .173 3.20 -.058 1.84%

100 1.370 3.24 -.098 3.11%

1000 13.395 3.12 .021 .66%

10000 131.694 3.12 .021 .66%

C Program:

# points Time (s) Estimated value Deviation % error

10 .010 2.80 0.34 10.87%

100 .013 3.32 -0.18 5.68%

1000 .037 3.17 -0.03 0.84%

10000 .197 3.16 -0.02 0.74%

Also include the answers to the following questions in the write up in your tarball:

* 1. **Instead of running a give number of points, run as many points as you can in 100 seconds. How many points were you able to compute for your bash and your compiled implementations?**

Bash Implementation: ~7132 points

Compiled Implementation: 7420021 points

**1.2 Which code was faster and why?**

**The C code was significantly faster. Bash is an interpreted language, while C is a compiled language. Interpreted languages must be translated at run time to machine instructions. A compiled language can be executed directly by the computers CPU.**

**1.3 Did you notice any difference in the accuracy? If there was a difference, what could you attribute this to?**

**For my results, when executing the bash script with 10,000 points, I was receiving random syntax errors and displaying the same estimated value of pi as the previous “1000” trials. However, the trials before that, I was receiving no syntax errors when executing the script. This is due to the high load for the interpreted language. The method, object, and global variable space model is dynamic, thus requiring many extra hash-table lookups on each access to a variable or a method call.**

**Problem 2: Binary File I/O**

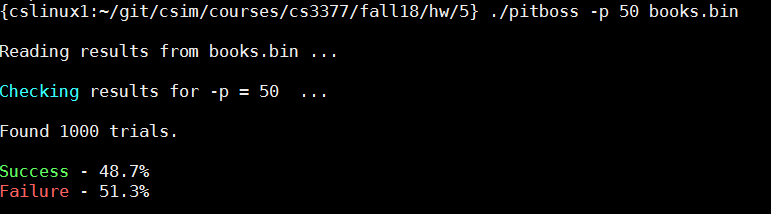
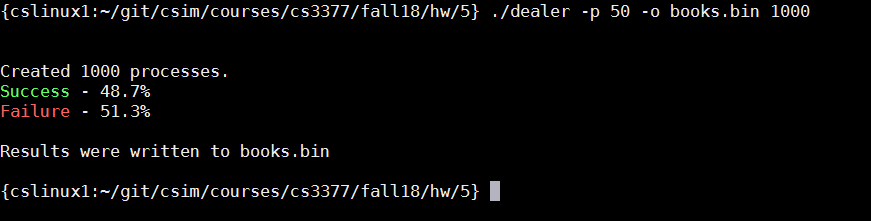
For this problem, you are to start with the same codes that you created for HW 5, ***dealer*** and ***hand***. You are to augment ***dealer*** so that it can take an additional input option, -o *filename*. This *filename* will be a binary file that contains the results of our simulation. If *filename* does not exist, your program ***dealer*** will create the file. If *filename* does exist, then simply append the results to the end of the file.

Once your ***dealer*** program can save statistics to *filename*, run it with 1000 trials per percentage “-p” starting with an input percentage of 10 and incrementing by 10 until you get to 90% for a total of 9 x 1000 trials run.

Finally, create a program named ***pitboss.*** This program will take two arguments, the ***dealer***

output file that you want it to parse as well as the “-p” value from the above runs for which

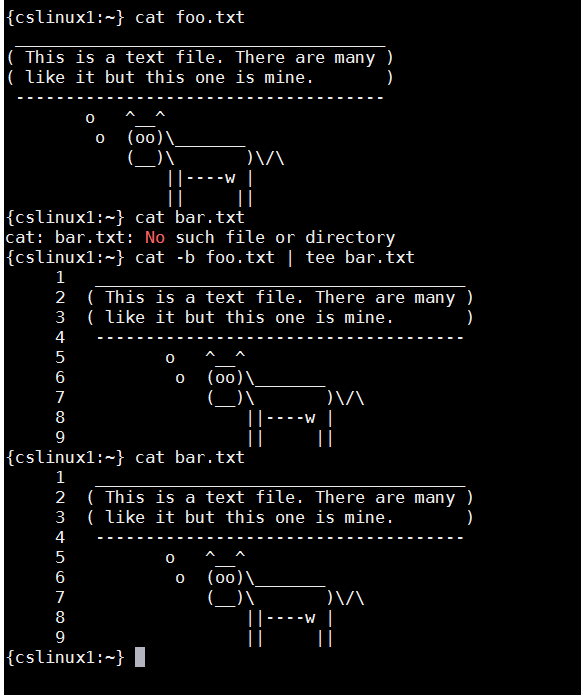
it is to generate statistics. Example:



**Problem 3: Fork, exec, and pipes**

UNIX systems include a program (filter) that copies standard input to standard output and a file descriptor passed on the command line at the same time. This program is ***tee***.

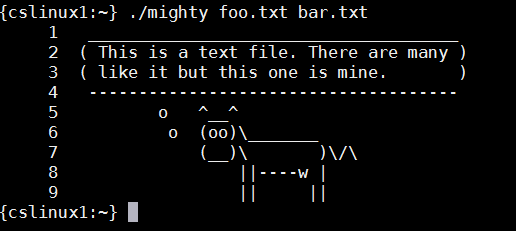
For example:



For this problem, we are going to generate our own version of tee called ***mighty***, which will

be equivalent to the “cat –b …” command from above. At the command line you will type

the following to generate the same results as the above shell command:



Details:

Error check the CLI input

o argv[1] will be the input file that’s going to get “tee”’d

o argv[2] will be the file you write to

create a pipe

fork the process

child will:

o call dup2 so that any writes to standard output will go instead of the write end of the pipe

o then execute the “cat –b” command using “execl”

parent will:

o open argv[2] for writing

o read from the pipe until there are no characters left

o write to STDOUT

o write to the fire descriptor associated with argv[2]

**Problem 4: Signals**

Write a program that will fork a child process. The parent will send a randomly generated signal to this child process. The child will overwrite the default action of all the signals (1 –

31; inclusive) so that a message is displayed with the child receives the signal. The parent will overwrite the default action of the same signals as well. In the parent’s signal handler, it will send a SIGKILL to the child.

Details: Parent will:

Overwrite the default action of signals 1-31

Seed the GSL random number generator

Sleep for 1 second

Loop infinitely doing the following:

o Randomly generating a signal between 1-31

o Printing a message of what signal will be sent to the child

o Sending the randomly generated signal to the child

o Nanosleeping for half a second

Child will:

Overwrite the default action of signals 1-31

Loop infinitely doing the following:

o Printing a message “child waiting”

o Nanosleeping for half a second

The child’s handler will:

print the number of the signal received with a message that the child received it

The parent’s handler will:

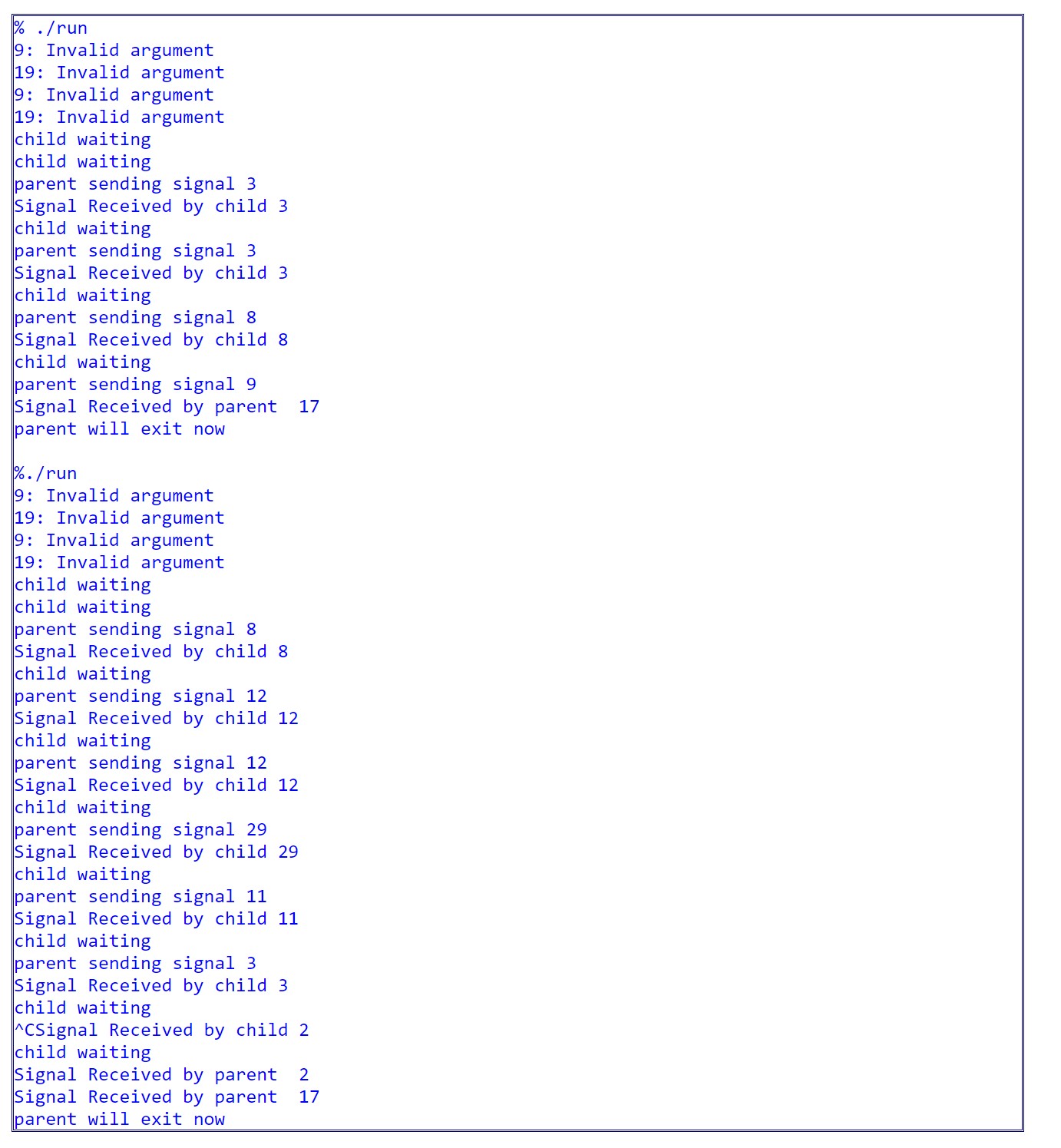
Print the numeric value of the signal received with a message that the parent received it

Send the SIGKILL signal to the child. Hint: the fork return will be a global variable

Wait for the child

print a message that the parent is done

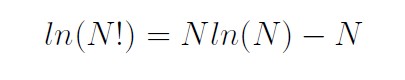
exit



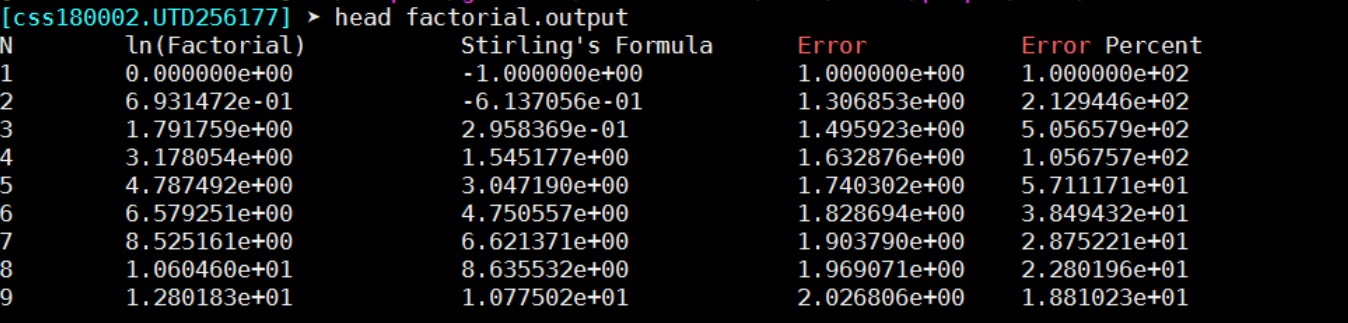
UT Dallas CS 3377 Dr. Simmons Final Page **8** of **9**

**Bonus Questions – each question is worth 10 bonus exam points**

**B1** Write a computer program to compute the natural log of the factorial N for every integer between 1 and 100, and compare the results to that given by a simple form of Stirling’s approximation.



When you run your program, have it write the results to STDOUT using a format similar to:



Calculate the value of N at which Stirling’s approximation will be within 0.1% of the exact result. There are many ways to do this; how you do it is left up to you.

**B2** Write a computer program that calculates and prints out all “ugly numbers” below

10000. Ugly numbers are numbers whose only prime factors are 2, 3 or 5. The sequence 1,

2, 3, 4, 5, 6, 8, 9, 10, 12, 15, … shows the first 11 ugly numbers. By convention, 1 is included.

**B3** Write a computer program that determines if an integer provide as input is an abundant number, a deficient number or a perfect number. If it is abundant, it also prints out its “abundance” and if it is deficient it prints out its “deficiency”.

<https://en.wikipedia.org/wiki/Abundant_number> <https://en.wikipedia.org/wiki/Deficient_number>

<https://en.wikipedia.org/wiki/Perfect_number>

UT Dallas CS 3377 Dr. Simmons Final Page **9** of **9**