Programming Assignment I

Due on Wednesday, February 3, 2016 @ 10pm

50 points

This is a warm-up simple programming assignment using Unix system calls `fork()` and `wait()`.

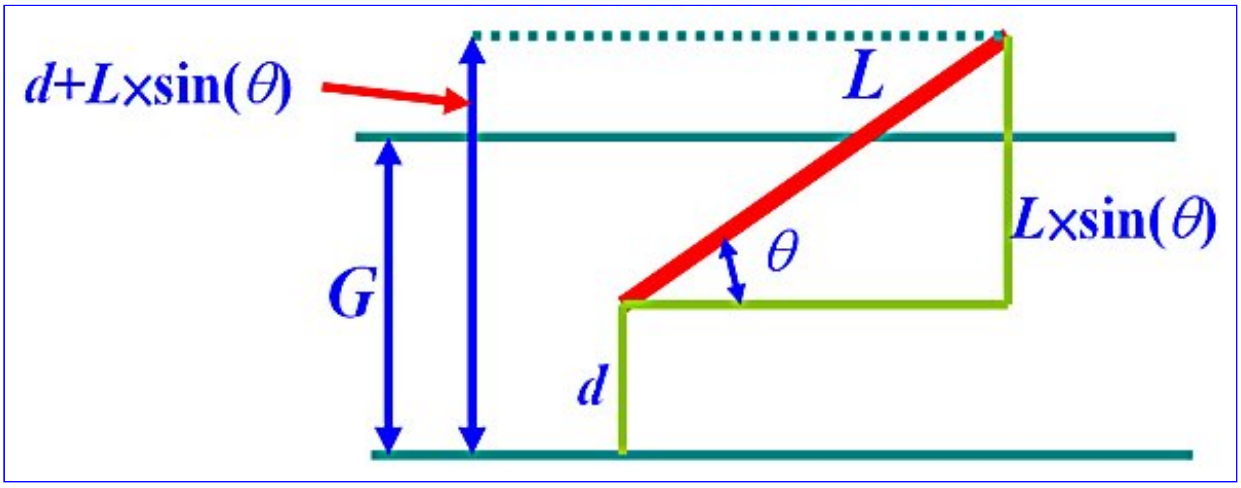
Running Four Independent Processes

Write a program that takes four positive integers *m*, *n*, *r* and *s* from its commend line arguments, creates four child processes, waits for them to complete, and exits. The first process sorts *m* random integers using the heap sort, the second computes the *n*-th Fibonacci number f_n using recursion, the third process finds the solution of Buffon's needle problem by throwing a needle *r* times, and the fourth process computes the integration of function sin(x) between 0 and π. Here are the details:

1. You may copy the heap sort algorithm from your data structures textbook.
2. The *n*-th Fibonacci number is defined recursively as follows:

   $f_1 = f_2 = 1$
   $f_n = f_{n-1} + f_{n-2}$ if $n > 2$

   Use **THIS** recursive formula to compute the *n*-th Fibonacci number.
3. The **Buffon's Needle** Problem. The problem was first stated by the French naturalist and mathematician George-Louis Leclerc, Comte de Buffon in 1733, and reproduced with solution by Buffon in 1777. Suppose the floor is divided into infinite number of parallel lines with a constant gap *G*. If we throw a needle of length *L* to the floor randomly, what is the probability of the needle crossing a dividing line? The answer is (2/π)*(*L*/*G*), where π is 3.1415926....



We may use a program to simulate this needle throwing process. For simplicity, let *L* = *G* = 1.

We need two random numbers:

1. *d*, a random number in [0,1), represents the distance from one (fixed) tip of the needle to the lower dividing line.

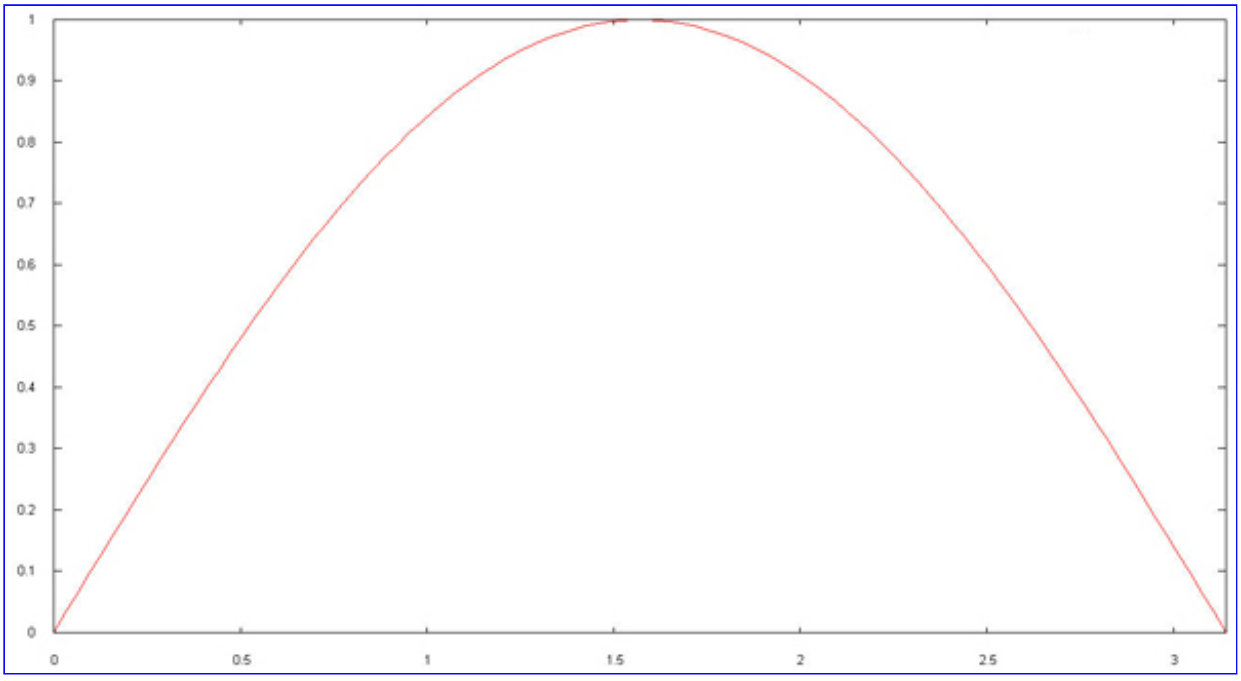
2. *a*, a random number in [0,2*π), represents the angle between the needle and a dividing line. Thus, if $d + L \times \sin(a)$ is less than 0 or larger than *G*, the needle crosses a dividing line.

In this way, your program loops *r* times. In each iteration, your program uses the above formula to throw a needle, and checks if the needle crosses a dividing line. If the needle crosses a dividing line *t* times, *t*/*r* is an approximation of the exact probability. In fact, if *r* is very large, the simulated result would be very close to the exact result. In our case, since $L = G = 1$, the result for large *r* should be close to $2/\pi = 0.63661.....$

4. You learned in your calculus class the following integration:

$$\begin{aligned} \int_0^{\pi} \sin(x) &= -\cos(x)\Big|_0^{\pi} \\ &= -(\cos(\pi) - \cos(0)) \\ &= -(-1 - 1) \\ &= 2 \end{aligned}$$

This means the area between the sin(x) function and the *x*-axis is 2. This area is completely enclosed in a rectangular area with length π and height 1 as shown in the figure below. The area of this rectangle is $\pi \times 1 = \pi$. You may use **`acos(-1.0)`** to obtain the value of π.



If we randomly pick *s* points in the rectangle and find out *t* of them is in the area of the sin(x) function, then the ratio *t*/*s* suggests that the area under sin(*x*) is *t*/*s* of the rectangle. Therefore, $(t/s) \times \pi$ should be close to 2. Here is what your program is supposed to do.

1. Generate two random numbers *a* and *b* where $0 \le a < \pi$ and $0 \le b < 1$. This (*a*,*b*) represents a point in the rectangle.
2. If $b \le \sin(a)$, point (*a*,*b*) is in the area between sin(x) and the *x*-axis.
3. Let the total number of points in the area between sin(x) and the *x*-axis be *t*.
4. Then, $(t/s) \times \pi$ should be close to 2, the desired result. This is especially true for very large *s*.

Program Specification

Write a program `prog1.c` in C to perform the following tasks

1. It reads in four command line arguments and converts them to four integers. This means `prog1.c` is run with the following command line

   ```
   ./prog1 m n r s
   ```

   where `m`, `n`, `r` and `s` are four positive integers.
2. Then, the main program forks four child processes, waits for their completion, and exits.
3. The first child process generates `m` random integers with C functions `srand()` and `rand()` into a ***local*** array (of the child process). **Each number should be scaled to the range of 0 and 99**. For example, `rand()%100` would do the trick. Then, this process does the following in this order:
   1. Prints the generated random integers.
   2. Uses the heap sort to sort this array. **Write your own heapsort function,**
   3. Prints the sorted array
4. The second child process computes the n-th Fibonacci number f_n with recursion. (Yes, you must use recursion to kill time!) This process does its task in the following order:
   1. Prints the value of `n`
   2. Uses recursion to compute f_n
   3. Prints the result.

   Use **`long`** for computation.
5. The third child process simulates throwing a needle r times and estimates an approximation of the exact probability. This process does this simulation in the following order:
   1. Prints the value of `r`
   2. Iterates r times. Note that $L = G = 1$.
   3. Prints the computed result.

   Since `rand()` only generates random integers in the range from 0 to `RAND_MAX`, you should convert an integer random number to a real one in the range of 0 and 1 using (`float rand()`)/`RAND_MAX`. If necessary, you may scale this real random number to other range.
6. The fourth child process simulates the calculation of the integration of the sin(x) function as given earlier. This process does the following work:
   1. Prints the value of `s`.
   2. Iterates s times and count the number of points in the area between sin(x) and the x-axis.
   3. Prints the computed area.

You must use recursion to compute f_n. Moreover, the parent and its four child processes must run concurrently. Violating these rules you will receive zero point for this programming assignment.

Input and Output

The input to your program should be taken from command line arguments. Your source program must be named as `prog1.c`. The command line looks like the following:

```
./prog1  m n r s
```

Here, `m`, `n`, `r` and `s` are four positive integers. There are always four arguments and they are always correct.

Moreover, `r` and `s` are usually very large, say 1,000,000.

Suppose the command line is

```
./prog1  8 10 100000 200000
```

Then, your program should (1) generate 8 random integers and sort them with the heap sort, (2) compute the 15-th Fibonacci number, (3) simulate throwing a needle 1000000 times, and (4) use 200000 points to compute the area between the sin(x) function and the *x*-axis in the range of 0 and π.

The output format from the heap sort process should be

```
Heap Sort Process Started
Random Numbers Generated:
   6   8   3  10  25   5   2  30
Sorted Sequence:
   2   3   5   6   8  10  25  30
Heap Sort Process Exits
```

Note that all output from this heap sort process starts on column 4 (*i.e.*, 3 leading spaces). Use `%4d` to print integers.

The Fibonacci process should have its output as follows:

```
Fibonacci Process Started
Input Number 10
Fibonacci Number f(10) is 55
Fibonacci Process Exits
```

All output from this process starts on column 7 (*i.e.*, 6 leading spaces). Since the computed Fibonacci number may be very large, use `%ld` to print the computed result.

The Buffon's Needle process should have its output as follows:

```
Buffon's Needle Process Started
Input Number 100000
Estimated Probability is 0.63607
Buffon's Needle Process Exits
```

Since the output value is always in the range of 0 and 1, use 5 positions after the decimal point for printing. All output from this process starts on column 10 (*i.e.*, 9 leading spaces).

The integration (*i.e.* fourth) process should have its output as follows:

```
Integration Process Started
Input Number 200000
Total Hit 127498
Estimated Area is 2.0027339
Integration Process Exits
```

All output from this process starts on column 13 (*i.e.*, 12 leading spaces).

Note that **the results of Buffon's Needle problem and the integration problem may be slightly different from the output shown here even with the same *r* and *s* because random numbers are used.**

The main program should print the following output:

```
Main Process Started
Number of Random Numbers   = 8
Fibonacci Input            = 10
Buffon's Needle Iterations = 100000
Integration Iterations     = 200000
Heap Sort Process Created
Fibonacci Process Created
Buffon's Needle Process Created
Integration Process Created
Main Process Waits
Main Process Exits
```

All output line from the main process should start on column 1 (*i.e.*, no leading spaces).

It is very important to remember the following. All processes are run concurrently and may print their output lines in an unpredictable order. As a result, the output lines from a process may mix with output lines from other processes. This is the reason that proper indentation is required to know who prints what. **Also make sure that each line will not contain the output from different processes.**

Submission Guidelines

General Rules

1. All programs must be written in C.
2. Use the `submit` command to submit your work. You may submit as many times as you want, but only the last on-time one will be graded.
3. Your program should be named as `prog1.c`. Since Unix filename is case sensitive, `PROG1.c`, `prog1.C`, `pROG1.c`, etc are ***not*** acceptable.
4. We will use the following approach to compile and test your programs:

   ```
   gcc  prog1.c -lm -o prog1 <-- compile your program
   ./prog1 8 10 100000 200000   <-- test your program
   ```

   This procedure may be repeated a number of times with different input to see if your program works correctly.
5. Your implementation should fulfill the program specification as stated. Any deviation from the specification will cause you to receive **zero** point.
6. A `README` file is always required.
7. **No late submission will be graded.**
8. **Programs submitted to wrong class and/or wrong section will not be graded.**

Compiling and Running Your Programs

This is about the way of compiling and running your program. If we cannot compile your program due to any

syntax errors, wrong file names, etc, **we cannot test your program, and, as a result, you receive 0 point**. If your program compiles successfully but fails to run, **we cannot test your program, and, again, you receive 0 point**. **Therefore, before submitting your work, make sure your program can compile and run properly.**

1. **Not-compile programs receive 0 point.** By **not-compile**, I mean any reason that could cause an unsuccessful compilation, including missing files, incorrect filenames, syntax errors in your programs, and so on. Double check your files before you submit, since I will **not** change your program. Note again: Unix filenames are ***case sensitive***.
2. **Compile-but-not-run programs receive 0 point. Compile-but-not-run** usually means you have attempted to solve the problem to some degree but you failed to make it working properly.
3. **A meaningless or vague program receives 0 point even though it compiles successfully.** This usually means your program does not solve the problem but serves as a placeholder or template just making it to compile and run.

Program Style and Documentation

This section is about program style and documentation.

1. For each file, the first piece should be a program header to identify yourself like this:

```
// ------------------------------------------------------------
// NAME : John Smith                              User ID: xxxxxxxx
// DUE DATE : mm/dd/yyyy
// PROGRAM ASSIGNMENT #
// FILE NAME : xxxx.yyyy.zzzz (your unix file name)
// PROGRAM PURPOSE :
//    A couple of lines describing your program briefly
// ------------------------------------------------------------
```

   Here, **User ID** is the one you use to login. It is ***not*** your social security number nor your M number.

   For each function in your program, include a simple description like this:

```
// ------------------------------------------------------------
// FUNCTION  xxyyzz : (function name)
//     the purpose of this function
// PARAMETER USAGE :
//    a list of all parameters and their meaning
// FUNCTION CALLED :
//    a list of functions that are called by this one
// ------------------------------------------------------------
```

   Note that you may also use `/* ... */` for comments if the compiler will complain. The use of `//` is part of the new C standard.
2. Your programs must contain enough concise and to-the-point comments. Do not write a novel!
3. Your program should have good indentation.
4. Do not use global variables!

Program Specification

Your program must follow exactly the requirements of this programming assignment.

Otherwise, you receive 0 point even though your program runs and produces correct output. The following is a list of potential problems.

1. Your program does not use the indicated algorithms/methods to solve this problem.
2. Your program does not follow the structure given in the specification. For example, your program is not divided into functions and files, etc when the specification says you should.
3. Any other significant violation of the given program specification.
4. **Incorrect output format.** This will cost you some points depending on how serious the violations are. The grader will make a decision. Hence, carefully check your program output against the required one.

Program Correctness

If your program compiles and runs, we will check its correctness. We normally run your program with two sets of input data, one posted on this programming assignment page (the public one) and the other prepared by the grader (the private one). You program must deliver correct results for both data sets. Depending on the seriousness of the problem(s), significant deduction may be applied. For example, if your program delivers all wrong results for the public data set, you receive 0 point for that component.

The `README` File

A file named `README` is required to answer the following questions:

1. **Question:** Draw a diagram showing the parent-child relationship if the following program is run with command line argument 4. How many processes are created? Explain step-by-step how these processes are created, especially who is created by whom.

```
void main(int argc, char **argv)
{
     int i, n = atoi(argv[1]);

     for (i = 1; i < n; i++)
          if (fork())
               break;
     printf("Process %ld with parent %ld\n", getpid(), getppid());
     sleep(1);
}
```

2. **Question:** Draw a diagram showing the parent-child relationship if the following program is run with command line argument 4. How many processes are created? Explain step-by-step how these processes are created, especially who is created by whom.

```
void main(int argc, char **argv)
{
     int i, n = atoi(argv[1]);

     for (i = 0; i < n; i++)
          if (fork() <= 0)
               break;
     printf("Process %ld with parent %ld\n", getpid(), getppid());
     sleep(1);
}
```

3. **Question:** Draw a diagram showing the parent-child relationship if the following program is run with command line argument 3. How many processes are created? Explain step-by-step how these processes are created, especially who is created by whom.

```
void main(int argc, char **argv)
{
     int i, n = atoi(argv[1]);

     for (i = 0; i < n; i++)
          if (fork() == -1)
               break;
     printf("Process %ld with parent %ld\n", getpid(), getppid());
     sleep(1);
}
```

You should elaborate your answer and provide details. **When answering the above questions, make sure each answer starts with a new line and has the question number (*e.g.*, Question X:) clearly shown. Separate two answers with a blank line. Also make sure that the diagrams in your `README` file will PRINT correctly. Do not judge the results on screen.**

Note that the filename has to be `README` rather than `readme` or `Readme`. Note also that there is ***no*** filename extension, which means a filename such as `README.TXT` is ***NOT*** acceptable.

`README` must be a plain text file. We do not accept files produced by any word processor. Moreover, watch for very long lines. More precisely, limit the length of each line to no more than 80 characters with the Return/Enter key for line separation. Missing this file, submitting non-text file, file with long lines, or providing incorrect and/or vague answers will cost you many points. Suggestion: Use a Unix text editor to prepare your `README` rather than a word processor.

Final Notes

1. Your submission should include two files, namely: `prog1.c` and `README`. Please note the way of spelling filenames.
2. **Always start early, because I will not grant any extension if your home machine, network connection, your phone line or the department machines crash in the last minute, etc.**
3. **Since the rules are all clearly stated, no leniency will be given and none of the above conditions is negotiable. So, if you have anything in doubt, please ask for clarification.**
4. **Click here to see how your program will be graded.**