Lab 2: The Inebriated Beetle

Spring 2016
Originally by: Dr. Phil Kearns (Used with permission)
Modified by Dr. Robert Marmorstein
Due: Friday, Feb. 26th by 11:59:59pm

Step 0. Setting Up

This assignment requires you to design and implement a small C program. Its primary purpose is simply to review memory allocation in C, but it also will help you practice using system calls such as fork and exec. Make a directory named "Lab3" and make it the current working directory. Then create a file named "beetle.c" which solves the problem described below:

Step 1. The Drunk Beetle and the Boiling Oil

A single intoxicated beetle is placed in the exact center of a cardboard square that is suspended over a large vat of boiling oil. The beetle, being quite drunk, behaves erratically. It moves exactly 1 inch (in 1 second) in a straight line in a randomly chosen direction. It then passes out for exactly 1 second. When it wakes, it again picks a random direction and moves 1 inch. We are interested in selling insurance to drunk beetles on squares suspended above vats of boiling oil, so we want to compute the average time that a drunk beetle will live after it starts to move for the first time. Clearly the way to get a handle on this average is to measure the time it takes for a beetle to get boiled for lots of beetles and compute the average over the entire sample. To make the problem a bit easier, assume that the last movement by the beetle lasts a full second. That is, don't worry about taking into account the fact that the last movement of the beetle will neither be a full inch nor a full second.

You should check out the man pages for:

- **random()**, the ``better'' random number generator on our system,
- **sin()**, **cos(),** etc., the trigonometric functions,
- **printf()**, the formatted output facility (paying special attention to the conversions for floating point numbers), and
- **atoi()**, and related functions that convert strings into numeric types.

Note that the trig functions are part of the mathematics library, so you will have to tell the linker to link in that library explicitly (using the -lm flag). If your source program is called `beetle.c`, and you want an executable called `Beetle`, do the following:

```
clang -Wall -o Beetle beetle.c -lm
```

(You may also use `gcc` if you prefer it to clang). Your program should accept two command line arguments that control the simulation:

> The first argument should be a positive nonzero integer which specifies the length of one side of the square (in inches).
>
> The second argument should be a positive nonzero integer which specifies how many beetles we will boil to estimate the average lifetime.

For example, if your binary is called `beetle`, the command line

```
        ./beetle 20 10000
```

will simulate the actions of 10000 beetles on a 20" by 20" square in order to compute the mean beetle lifetime. Make sure your program exits gracefully with an error message for "bad" command line arguments. Argument checking is a significant component of this assignment.

One last note: a direction isn't necessarily "north", "south", "east", or "west". Your program should support movement of any angle from 0 to 360 degrees. Keep in mind that the standard UNIX trig functions expect arguments in **radians** not degrees.

Step 2. History

Create a coords struct with the following code:

```
struct Coords {
        double x;
        double y;
        Coords* next;
};
```

Use this struct to make a linked list named ``history" which keeps track of every pair of coordinates which the beetle reaches. At the end of the inner loop, print out the history to a file named `history.log`.

You may find useful the man pages for:

malloc(),	which allocates memory on the heap
free(),	which returns memory to the operating system
fopen(),	which opens a file
fprintf(),	which writes to a file
fclose(),	which closes the file

Step 3. Parallel Programming

The new computers in the lab have quad core processors. The current Beetle lab only takes advantage of one core because it is running in a single process with only one thread. This means that it takes a long time to run if the number of Beetles is very large. Follow the steps below to modify your program to use the "fork" and "exec" system calls so that it that spawns multiple beetle simulations and then collates the results into a single average.

Architecture of the Program

Your existing (sequential) program is in the file "beetle.c". Your new program should consist of two separate executables (and therefore two separate source files). One executable will perform the beetle simulation. The other will be a "manager" program that spawns the beetle simulations and waits for them to finish. When each simulation has finished, the manager will print out a report and terminate.

Copy your simulation process from the "beetle.c" file to a new file named "sim.c" and create a new file named "manager.c" for the manager process.

To make this work, you need to change a few things in the beetle simulation:

1. Instead of calculating an average, the beetle simulation should simply find a sum (the total number of seconds all of the beetles survived).

2. The beetle should take a new command line argument, the "id", which will be a unique string created by the manager to identify the different children.

3. Instead of printing to the screen, the beetle should send its output to a file. The name of the file should be ``beetle-output-{id}.txt" where {id} is the "id" argument passed to the program.

4. You can (and should) remove the history logging code from your "sim.c" file.

Step 4. Creating the Manager

The code for your manager should be in a file named "manager.c"'. Your manager program should expect three command line arguments:

> The first argument should be a positive nonzero integer which specifies the length of one side of the square (in inches).
>
> The second argument should be a positive nonzero integer which specifies how many beetles we will boil to estimate the average lifetime.
>
> The third argument should be a positive nonzero integer which specifies how many simultaneous simulations to run.

For example, if your binary is called `manager`, the command line

```
./manager 20 10000 4
```

should simulate 10000 beetles on a 20" X 20" square using four processes. Your program should parse the command line inputs and then fork off enough child processes to satisfy the command line arguments. Each child process (but not the parent) should immediately exec the ``sim" program, passing in the appropriate command line arguments. Try to split the number of beetles roughly equally between your processes. For example, if I am simulating 10000 beetles on 4 processes, each process should simulate roughly 2500 beetles. It's okay if the last child has slightly more or slightly less than the others. For example, if I was only simulating 30 beetles, it would be fine for the first three processes to have 7 beetles a piece and the last one to have only 2.

After forking off its children, the manager should block (wait) until every child has terminated. Then it should open up the beetle-output-id.txt files for each beetle, read in the number of seconds, and compute the overall average. It should then print a message to the screen in this form:

```
Simulation Complete.
The average lifetime of the beetles was: X seconds.
```

Replace X with the actual average, of course.

You may find useful the man pages for:

fork(), which splits a running process into two processes.
execve(), which loads a new program from disk into an existing process.
wait(), which cause a process to block until one of its children terminates.
fopen(), which opens a file
fscanf(), which reads values from an open file
fclose(), which closes an open file

Step 5. Glitter

As with other projects, extend this one in a unique and creative way. Name the result "glitter.c". There are all sorts of ways to make this more interesting --- one idea is to change the simulation so that instead of a rectangle, the cardboard is cut into the shape of a C or an F. Another is to use animation (either ASCII art or using SDL) to illustrate the simulation graphically. Even better -- come up with your own unique idea.

Submission

Create a tarball named "Lab3.tar.gz" and submit through the course web site as usual.