

Operating Systems, Section 002, Assignment 1

This assignment includes a few programming problems. When grading, we will compile and test your programs on one of the EOS Linux machines. In general, we'll compile with options like the following. Some programs require additional options. These are described with the programming problem when they're needed.

```
gcc -Wall -g -std=c99 -o program program.c
```

If you develop a program on a different system, you'll want to make sure it compiles and runs correctly on University EOS Linux machines using the given options before you turn it in. Also, remember that your programs need to be commented and consistently indented, and that you **must** document your sources and write at least half your code yourself from scratch.

1. (16 pts) Log in to one of the EOS Linux machines, use the online manual pages to answer each of the following questions about system calls on Linux. As with the first assignment, be sure to look at the section of the online manual for system calls. Sometimes there are library functions or shell commands with the same name.

Write your answers into a plain text file called **problem1.txt** and submit it for assignment_1 Moodle.

- (a) The `stat()` and `fstat()` system calls let you query information about a file. What's the difference between `stat()` and `fstat()`?
 - (b) You may remember, in C and Software Tools class, we talked about the dangers of opening a file in a directory where someone else could potentially get there first. If an attacker creates a symbolic link with the same name, an attempt to create a new file could be tricked into overwriting a file elsewhere on the system.
The `open()` system call provides a way to prevent this attack. How can you call `open()` so it can't accidentally overwrite a file that's pointed to by a symbolic link?
 - (c) An operating system will normally keep track of resource usage for each of its processes. On a Unix system, a process can use the `getrusage()` system call to check on its own resource usage. This call fills in the fields of a struct, reporting the process' usage of various types of resources. In this struct, what are `ru_utime` and `ru_stime` fields and how are these two fields different?
 - (d) Lots of system calls will return immediately if a signal is delivered while the process is waiting in the system call. For example, `mq_receive()` is like this. If a signal is delivered during a call to `mq_receive()`, how can a program tell that `mq_receive()` returned because of a signal rather than the arrival of a message?
2. (40 pts) For this problem, you're going to write a program named **zerosum.c**. It will be able to use multiple processes and multiple CPU cores to solve a computationally expensive problem more quickly. The task is simple. You're given a sequence of positive and negative integer values, like the one shown below.

1	3	-4	-1	1	1
---	---	----	----	---	---

Your job is to find all contiguous ranges of values that add up to zero (like substrings of the sequence with a sum of zero). For example, the values at index 0 up to index 2 are 1, 3 and -4. These add up to zero. The values from index 0 to index 4 also add up to zero. So do the values from index 1 to 5 and the values from index 3 to 4.

1	3	-4	-1	1	1
---	---	----	----	---	---

0 ... 2

1	3	-4	-1	1	1
---	---	----	----	---	---

0 ... 4

1	3	-4	-1	1	1
---	---	----	----	---	---

1 ... 5

1	3	-4	-1	1	1
---	---	----	----	---	---

3 ... 4

Program Input

Your program will read input from standard input. As you can see in the examples below, we will use input redirection to get it to read input from a file rather than from the terminal. Input will be a sequence of up to 1,000,000 integer values, one value per input line. The following shows the contents of the first sample input file, `input-1.txt`. It contains the same sequence shown above.

```
1
3
-4
-1
1
1
```

Program Output

The program will find all contiguous ranges of values in the input sequence with a sum of zero and report the number of different ranges it finds. Your program's last line of output should look like the following, where n is the total number of contiguous ranges of values that add up to zero. For the example above, this would be four.

Total: n

Command-line Arguments

Your program will take up to two command-line arguments. The first argument gives the number of worker processes to use (see below).

Your program will take the word “report” as an optional, second command line argument. If the report option is given, then the program will print a line of output for each each zero-sum range as it discovers it. This will be reported like “ $a \dots b$ ”, where a is the index of the start of the range and b is the index of the end. These indices count from zero for the first element of the sequence. In the sequence 1, 3, -4, -1, 1, 1, the values from index 1 up to index 5 add up to zero, so the program would report it as the following if the report flag was given on the command line (along with similar reports for three other ranges):

```
1 .. 5
```

Since your program is expected to report ranges as it finds them, different executions may report them in different orders. The output order will depend on the timing of the execution and how you chose to implement your program. It’s OK if the order varies, but the Total: output should always be last.

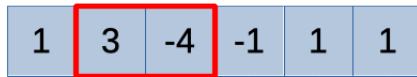
Partial Implementation

You get a starter file to help you with your implementation. The starter includes code to parse the input sequence and the command-line arguments. The input is guaranteed to be in the format described above; you don’t have to add code to detect and respond to bad input. Also, you’re guaranteed that all ranges from the input sequence will add up to a number that fits in a signed, 32-bit integer and that the number of zero-sum ranges will also fit in a 32-bit signed integer. You don’t need to worry about overflow for either of these.

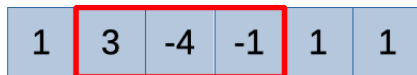
You’ll add code to look for ranges of values that add up to zero. We will do this by checking every range of values, starting from index i and ending at some index j with $j \geq i$. You should be able to do this in $O(n^2)$ time, where n is the number of elements in the sequence. For checking the sum of values in a range, you can organize your code so you check all ranges that start in the same place one after another (or, alternatively, all ranges that end in the same place). The following figure illustrates how you can handle all the ranges that start at index 1.



sum: 3



sum: -1



sum: -2



sum: -1



sum: 0

Computing the sum of the range that starts and ends at index 1 takes constant time (it's just the value 3). Then, we can compute the sum of the sequence that starts at index 1 and ends at index 2 by adding just one value, -4. Then, we can check the sum of the range of values from index 1 up to index 3 by adding just one value, -1. Each time we extend the sequence by one element, we just have to add in the value of that one element. So, checking the sum of a particular range of values should just take constant time.

Parallelization

With an $O(n^2)$ solution, it could take a long time to handle a large sequences of values. As you can see below, my solution took several seconds on a 100,000-element sequence. However, the $O(n^2)$ approach is easy to parallelize, dividing the work among multiple processes.

On the command-line, the user specifies how many child processes the program should create. We will call each child process a **worker**. After reading the input sequence, the parent will `fork()` to create each worker and then let the workers do all the real work of solving the problem.

Each worker will be responsible for checking some ranges of values from the sequence. You can divide up the work however you want, but try to divide it evenly so all the workers have a similar amount of work to do. For example, you could make each worker responsible for checking ranges based on where the range starts (or, alternatively, where the range ends). If you had 3 workers, you could make the first one responsible for checking ranges starting at index 0, index 3, index 6 and so on. The next worker

could be responsible for ranges starting from index 1, index 4, index 7 The last worker could be responsible for ranges starting at index 2, index 5, index 8 This isn't the only way to divide up the work, but it's an easy technique to implement. Also, it lets each worker efficiently compute the sums in each range the way it's described above.

Each worker should count (and optionally report) all the zero-sum ranges that it finds in its part of the problem. When they're done, the workers will send their counts back to their parent process, so it can report a total count.

If the user asks your program to report ranges, your worker processes will just print out each range as soon as it is found. You don't have to report them in any particular order. In fact, with workers printing out reports as soon as they find them, you may even get different orderings of the reported output each time you run your program. That's OK as long as you report all the right ranges and only print each range once.

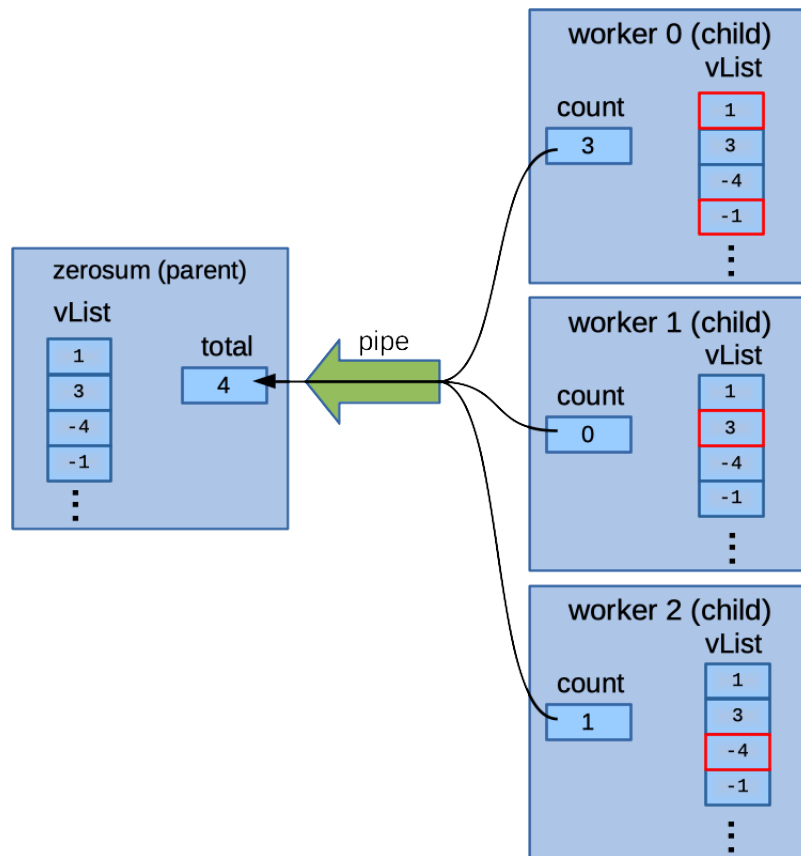
At the end, your parent process will print out the total number of zero-sum ranges found across all the worker processes. This is going to require communication from the child processes to the parent (see below). Once the parent has added up the number of ranges found by all the workers, the parent should print out the total.

Interprocess Communication

When the workers are created, they will automatically get a copy of the list of values read in at program start-up, since children get a copy of everything in the parent's memory. So, after creating the workers, the parent will just wait for the workers to find all the zero-sum ranges.

When the workers are done, they need a way of sending their resulting local counts back to their parent. We'll use an anonymous pipe for this. Before creating its children, the parent will make a pipe. Then, when a child is done, it will write its local count into the writing end of the pipe and the parent will be able to read it out of the other end.

We'll send this count in binary; that will be easier. Since communication is just going between processes on the same host, we don't have to worry about byte order or other architectural differences. Parent and child will always be running on the same hardware. To send the binary representation of an integer through a pipe, we just need to use the starting address of that integer as the buffer to send. The size of an integer is the number of bytes we want to send. On the receiving side, we can use a similar technique to read the contents of the integer into a variable of type `int`.



After reading the local count from each child, the parent will add them up to report an overall total at the end of execution. Before exiting, the parent should use `wait()` to wait for each of its children to terminate.

File Locking

There is a small risk that two workers will try to write their locally computed count into the pipe at the same time. For a moderate number of workers, I don't think this would really cause a problem, but, either way, it's easy to prevent. Before writing its result into the pipe, a worker will lock the pipe. This can be done with a call like the following.

```
lockf( fd, F_LOCK, 0 );
```

Then, after writing its value, a worker can let others use the pipe by making a call like:

```
lockf( fd, F_ULOCK, 0 );
```

This is an example of advisory file locking. If two workers try to lock the pipe at the same time, one of them will be granted the lock and the other will automatically wait until the first one unlocks it. As long as all our workers lock the pipe before using it, two of them won't be able to write into it at the same time.

Compiling

Using the `lockf()` call requires an extra preprocessor flag on the EOS Linux systems. You'll want to compile your program like the following. The `-lm` option links with the math library, in case you need to use the `sqrt()` function.

```
gcc -Wall -std=c99 -D_XOPEN_SOURCE=500 -g zerosum.c -o zerosum
```

Sample Execution

Once your program is working, you should be able to run it as follows. I've included some shell comments to help explain these examples (the lines starting with a pound sign). Of course, you don't need to type these in, but the shell should just ignore them if you do.

```
# Use one worker on the smallest test case.
$ ./zerosum 1 report < input-1.txt
0 .. 2
3 .. 4
0 .. 4
1 .. 5
Total: 4

# Try the next, larger test case using two workers.
# Your output may be in a different order, but the total report
# should still be last.
$ ./zerosum 2 report < input-2.txt
1 .. 2
0 .. 12
12 .. 14
4 .. 18
3 .. 7
1 .. 7
5 .. 9
9 .. 13
6 .. 15
2 .. 17
7 .. 19
Total: 11

# Use four workers on input-3.txt (a 100-element sequence)
$ ./zerosum 4 report < input-3.txt
11 .. 24
1 .. 28
17 .. 32
29 .. 36
1 .. 36
... lots of lines omitted in the sample output ...
80 .. 99
78 .. 99
70 .. 99
44 .. 99
```

```
22 .. 99
Total: 225
```

```
# See how long it takes on input 5 with just one worker, a 100,000 -
# element sequence. Here, we're not using the report option so it won't
# slow down the execution. You may get different timing results than
# I do, depending on your implementation and the system you run
# your solution on.
```

```
$ time ./zerosum 1 < input-5.txt
Total: 2500985
```

```
real 0m11.171s
user 0m11.079s
sys 0m0.055s
```

```
# Try again with 4 workers. This should take about the same amount of
# total CPU time, but, if you have multiple cores it should finish
# faster. With more than one CPU core, workers should be able to run in
# parallel, reducing the amount of time from start to finish. Read
# the online manual page for the time command to make sure you understand
# what the time output means.
```

```
$ time ./zerosum 4 < input-5.txt
Total: 2500985
```

```
real 0m3.027s
user 0m11.817s
sys 0m0.071s
```

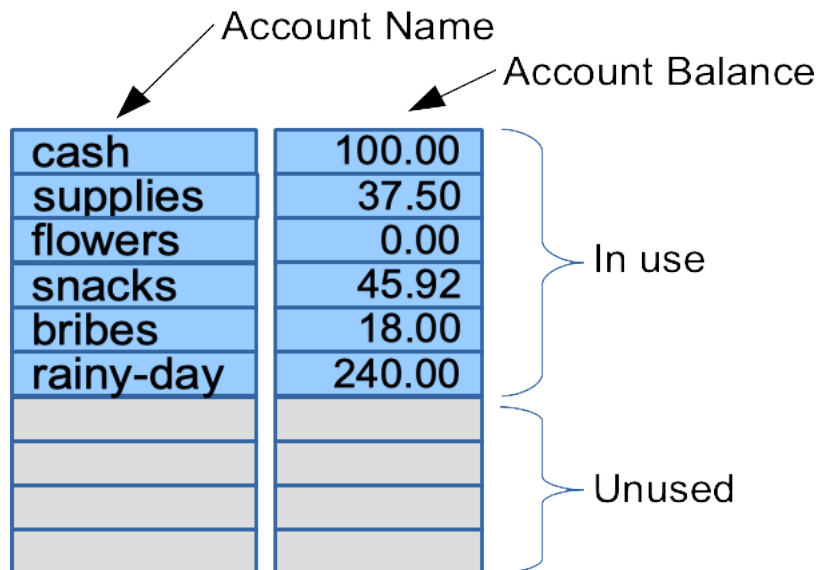
Be sure to try out your program with the `time` command on the largest input file, using different numbers of workers. As long as you have at least as many cores as workers, you should see speedup that's close to linear in the number of workers. So, if you use two workers, it should take about half as much time as 1 worker; if you use three workers, it should take about a third of the time. There will be some overhead and maybe some uneven distribution of work, so don't expect perfect linear speedup. However, if you're not getting much speedup at all, that's a sign that you've done something wrong, or you could divide up the work more evenly among the workers. We will be looking at your execution time when we test your solution. To test performance, you may need to use your own Linux machine or one of the EOS Linux lab machines; the systems you access via `remote.eos.ncsu.edu` are not good for this test (i.e., don't use them for this). It looks like they only have two cores each, and since they are shared systems, you will be competing with other users for CPU time.

Submission

When you're done, submit your source file, `zerosum.c`, using the `assignment_1` assignment in Moodle.

3. (40 pts) For this problem, you're going to use Inter-Process Communication (IPC) to create a pair of programs, `client.c` and `server.c` that work together. I've already written part of the server for you. The client and server work together to implement a list of accounts, each with a current account balance. The server will maintain the list of all accounts and balances, and the client can send commands to the server to query the balance of individual accounts and to credit or debit a given amount to an account.

The server can keep up with up to 10 accounts, each with a name of up to 12 characters in length. Each account can have a current balance between 0.00 and 10000000.00 (ten million dollars and zero cents).



The diagram shows a table with two columns: 'Account Name' and 'Account Balance'. The first six rows are labeled 'In use' and contain the following data: cash (100.00), supplies (37.50), flowers (0.00), snacks (45.92), bribes (18.00), and rainy-day (240.00). The next four rows are labeled 'Unused' and are empty. Arrows point from the column headers to the first row of the table.

Account Name	Account Balance
cash	100.00
supplies	37.50
flowers	0.00
snacks	45.92
bribes	18.00
rainy-day	240.00

Running the Server

When you start the server, you will list all the account names and their starting balances as command-line arguments. `argv[1]` will be the name of the first account, and `argv[2]` will be its balance. Likewise, `argv[3]` and `argv[4]` will give the name and starting balance for the second account. The account name must be a unique string (not matching the name of any other account) of between 1 and 12 characters, and it may not contain whitespace. The account balance can be given as a floating point number, but it will be stored by the server rounded to the nearest cent.

The following example shows how to start the server with information about six accounts like the ones shown in the figure above.

```
$ ./server cash 100.00 supplies 37.50 flowers 0.00 snacks 45.92 bribes 18 rainy-day 240.00
```

The command line arguments must describe information for between 1 and 10 accounts. If it doesn't, or if the given accounts don't satisfy the requirements described above (unique names, balance between 0.00 and 10,000,000.00, etc) the program should terminate unsuccessfully and print the following usage message to standard error:

```
usage: server (<account-name> <balance>)+
```

You can develop your own representation for storing the account names and balances. For future assignments, you will want to store all the names and balances in a single struct (but you're not required to for this assignment). Dollar amounts are printed and parsed as floating point numbers, but the sever should store the balance as an integer number of cents.

After the server starts successfully, it will continue running indefinitely, responding to client requests until the user terminates it with `ctrl-C`.

Running the Client

Each time you run the client, you indicate with command-line arguments what command it should perform. It sends the command to the server and then prints out the server's response. The client can be run the following ways:

- `./client credit acct v`

Running the client like this requests that the server add the given non-negative value, *v*, to the account with the given name, *acct*. The value is given as a floating point number; it should be rounded to the nearest integer cent. It is an error if the given value isn't a non-negative floating point number, if the given name isn't the name of an account known to the server or if the resulting balance would be larger than ten million. The server should send the client a response indicating success or failure of the request. Before terminating, the client will print out "success" if the command succeeded, or "error" if it failed.

- `./client debit acct v`

This command is like the credit command, but it reduces the balance in the given account by the given (non-negative) amount. As with the credit command, *v* should be rounded to the nearest integer cent. It is an error if the given value isn't a non-negative floating point number, if the given name isn't the name of an account known to the server or if the resulting balance would be negative. The server will send back a response indicating the success of the command, and the client will print out either "success" or "error".

- `./client query acct`

When the user enters this command, the client will request the current balance for the account with the given name. After receiving a response from the server, the client should print out the balance as a floating point number, rounded to the nearest cent. It should print "error" if the given account name isn't known to the server.

Error Conditions

If the user gives the client command-line arguments that aren't one of the commands described above, the client should print out "error". Whenever the client exits with the "error" message, it should exit unsuccessfully.

You can detect some usage errors in the client, without ever sending a request to the server. It's OK if you want to report the "error" message for these without even contacting the server. For others, you will need to contact the server and receive some kind of response indicating error.

We're not requiring a particular error message if, say, you can't open a message queue or if there's a problem when you try to receive a message. You will want to report if these errors happen (that may help with debugging) but any reasonable error message will be fine.

In the past, I've seen lots of students have trouble with the `mq_receive()` call. Be sure you look at the documentation for this, and maybe check the value of `errno` if it's not working for you.

Server Termination

The server will continue running until you kill it, accepting requests from any client that sends them. The server will use the `sigaction()` call we looked at in class to register a signal handler for `SIGINT`. When the user kills the server with `ctrl-C`, the server will catch the signal, print out a newline character, then print out a report of the final balance in each account. This report should list the accounts, one per line in the same order they were given on the command line at server start-up. The account name should be given, right justified in a 12-character field, then a space, then the final balance, right justified in a 11-character field and rounded to the nearest cent. If the server was started as shown above then

immediately killed with ctrl-C, this is what you would expect to see in the terminal. Printing the newline character right before the table makes the formatting look better. The first line of the table isn't on the same line as the “^C” that gets printed when you press ctrl-C.

```
$ ./server cash 100.00 supplies 37.50 flowers 0.00 snacks 45.92 bribes 18 rainy-day 240.00
^C
      cash      100.00
    supplies      37.50
      flowers       0.00
       snacks      45.92
       bribes      18.00
    rainy-day     240.00
```

Client/Server Communication

The client and server will communicate using POSIX message queues. Each time the client is run, it will send a message to the server and then wait for a response to report to the user.

We have a pair of example programs from class that demonstrates how to create and use message queues. A message queue is unidirectional communication channel that lets us send messages (arbitrary sequences of bytes) between processes on the same host. Each message queue needs a unique name. Two processes can communicate by using `mq_open()` to create a new or open an existing message queue with an agreed-upon name. Then, one process can use `mq_send()` to put a messages into the queue, and another program can use `mq_receive()` to obtain copies of the messages from the queue in the same order they were sent. When a program is done using a message queue, it can close it using `mq_close()`. The message queue will continue to exist (maybe even with some queued messages) until it is deleted with `mq_unlink()`.

Message queues let us send an arbitrary sequence of bytes. You can decide how you want to encode client requests and server responses as messages. For example, you can just encode them as null-terminated strings with spaces separating different values in the message. Using a text format for messages might help with debugging, but it's fine to use a binary format if you'd like.

We're each going to need two message queues, one to send requests to the server and another to send responses back to the client. On a single host, every different message queue needs a unique name. To avoid name collisions between students, we'll just include our unity IDs in every message queue name. Use the name “/unityID-server-queue” for the message queue used to send messages to your server. Use “/unityID-client-queue” as the name of the queue to send responses back to the client.

Partial Implementation

When started, the server creates both of these message queues, opening one for reading and the other for writing. Then, the server repeatedly reads messages from the server-queue and sends back the response via the client-queue. The course Moodle site has a partial implementation of the server. It already creates the needed message queues and unlinks them when it's done. You will need to add code to store and initialize the list of accounts. You'll also need code to read, process and respond to messages and handle the SIGINT sent when ctrl-C is pressed, reporting a final balances before the program exits.

In addition to the partial server implementation, you also get a `common.h` header file, defining some constants needed by both the client and server. You'll need to update this file to use your unity ID in the message queue names. I'm not providing a partial implementation for the client. You get to write that part yourself. The client is easier; mine is less than half the size of my server. Remember, the

server is responsible for creating both message queues. The client just needs to open and use them; be sure you don't accidentally delete or overwrite them from the client.

Compilation

To use POSIX message queues, you'll need to link your programs with the `rt` library. Also, to use `sigaction()` you will need to define the preprocessor symbol, `_POSIX_SOURCE`. You should be able to compile your client and server using commands like the following:

```
gcc -D_POSIX_SOURCE -Wall -g -std=c99 -o server server.c -lrt -lm
```

```
gcc -Wall -g -std=c99 -o client client.c -lrt -lm
```

Execution

Once your code is working, you should be able to leave the server running in one terminal window and connect to it repeatedly by running the client in a different window.

Logging in via `remote.eos.ncsu.edu` or `remote-linux.eos.ncsu.edu` takes you to a pool of several different hosts, not necessarily to the same machine every time you connect. **Be careful.** If you log in to this address twice, using `putty` or another `ssh` client, you may not actually get two terminal windows connected to the same host. Message queues only let you communicate between processes on the same host, so they won't work if you accidentally login on two different machines.

You can type `hostname` in each terminal to see what host it's running on. To be sure you get two logins on the same host, you can login once, use `hostname` to figure out what host you're on, then login directly to that same host (rather than `remote.eos.ncsu.edu`) a second time using your `ssh` client.

Sample Execution

Once your client and server are working, you should be able to run them as follows. Here, the left-hand column shows the server running in one terminal, and the right-hand column shows the client being run repeatedly in a different terminal window. I've put in some vertical space to show the order these commands are executed, and I've included some comments to explain some of the steps. We start the server on the left, run the client several times and then kill the server. When the server is terminated, you can see a printout for the `ctrl-C` (printed as `^C`), then the server's report of the final account balances.

```
# Run the server with a few accounts
$ ./server cash 100.00 supplies 37.50 flowers 0.00 snacks 45.92 bribes 18 rainy-day 240.00

# In another terminal, run the client
# First, check the balance in an account.
$ ./client query cash
100.00
$ ./client query supplies
37.50

# Try the credit command a couple of times
$ ./client credit snacks 14.08
success
$ ./client query snacks
```

```

60.00
$ ./client credit cash 0.12
success
$ ./client query cash
100.12

# Try the debit command a couple of times
$ ./client debit bribes 18
success
$ ./client debit cash 100
success
$ ./client query bribes
0.00
$ ./client query cash
0.12

# Try some invalid commands
$ ./client query abcdefg
error
$ ./client credit flowers abc.123
error
$ ./client debit cash 50.00
error

# Kill the server with ctrl-C and see the final balances
^C
    cash      0.12
supplies    37.50
  flowers     0.00
    snacks    60.00
    bribes     0.00
rainy-day   240.00

# Try some invalid arguments for the server.

# No balance for c
$ ./server a 5.0 b 10.0 c
usage: server (<account-name> <balance>)+

# Invalid balance for z
$ ./server x 10.50 y 25 z abc.xyz
usage: server (<account-name> <balance>)+

# Too many accounts
$ ./server a 1 b 2 c 3 d 4 e 5 f 6 g 7 h 8 i 9 j 10 k 11
usage: server (<account-name> <balance>)+

```

Submission

When you're done, submit the source code for your **client.c**, your modified **server.c** and your modified **common.h** using the assignment_1 assignment submission link in Moodle. From previous years

teaching this class, I've seen that students sometimes forget to submit their header files, so be sure to submit your **common.h** file. Otherwise, we won't be able to compile your code.