Operating Systems, Assignment 2

This assignment includes two programming problems. As always, be sure your code compiles and runs on the EOS Linux machines before you submit. The define for `_XOPEN_SOURCE` is for the shared memory calls used in problem 3.

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

For Java programs, also be sure to try these on an EOS Linux machine before you submit. This will help make sure they compile and run with the version of the compiler we have installed. For Java, we'll compile your source code using a command like:

```
javac Program.java
```

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) For this problem, we're going to look at how different schedulers behave and how this can affect performance. Pretend the following table describes the next CPU burst for a collection of processes. The table also gives the time at which the process arrives in the ready queue and the process' priority values (lower values representing higher priority, like we did in class).

Process	Burst Length	Arrival Time	Priority
P_1	4	0	4
P_2	6	2	6
P_3	2	6	8
P_4	3	8	2
P_5	7	10	5
P_6	2	12	7

   For each of the following scheduling algorithms, draw a Gantt chart like the ones in the CPU scheduling chapter of your textbook. Also, for each schedule, compute the average waiting time and the average turnaround time, where turnaround time is defined as the time from when a process arrives in the ready queue to when if finishes its CPU burst.

   (a) SJF

   (b) SRTF

   (c) Preemptive Priority

   (d) Round Robin with time quantum = 3

   For SRTF, it's possible for there to be a tie between the remaining burst time of the currently running process and a newly arrived process (we saw this in our in-class example). If this happens, just let the currently running process continue running (so, all else being equal, avoiding a context switch).

   For round robin, it's possible for there to be a tie between a process entering the ready queue and another process being preempted and returning to the ready queue (we saw this in our in-class example). If this happens, put the preempted process behind the entering process in the queue (so the newly arrived process will get to run earlier).

   You may draw your Gantt charts by hand and scan them in if you'd like. Just make sure your drawing is clear enough to read easily. Alternatively, you may draw them using the drawing program of your choice. Either way, submit your charts as a PDF file called **charts.pdf**. If you'd like to report waiting

and turnaround time in your charts.pdf file, that's fine. Or, if you'd like to just write this part up in a text file, you can submit it as **times.txt**. Be sure to include labels in your `times.txt` file, so we can tell what times you're reporting.

2. (40 pts) For this problem, you're going write a multi-threaded Java program, `Zerosum.java`, that does the same job as the `zerosum.c` problem from homework assignment 1. Like the previous program, your `Zerosum.java` will take a command-line argument giving the number of workers to use to find ranges of the input sequence that add up to zero. It will also take an optional argument, `report`, that tells it to report each zero-sum range as it discovers it (just like in the previous assignment). It will read the input sequence from standard input, just like in the previous assignment.

   This time we'll use threads as workers instead of separate processes. This will simplify our implementation a bit, since we won't have to use message passing to communicate results from the workers. Threads automatically share memory, so we can let the main thread[1] get each worker's result by just leaving it in a field of the worker's Thread object.

   After reading the input sequence, the main thread will create the requested number of worker threads. The workers will work in parallel to look for ranges of values that add up to zero, with each worker examining just some of the possible zero-sum ranges. You can divide up the work however you want, but you should divide it evenly enough that you get speedup with multiple workers on a multi-core system. You may not get perfect linear speedup, but you should be able to see some speedup with more workers, until you have a worker for every CPU core.

   The main thread will probably need to pass a value to each worker, so it will know what part of the problem it's supposed to work on. Then, after starting all the workers, the main thread will wait for them to terminate and then add up their results to report a total. To do this , it will need to get some result value from each of the worker threads. You can handle passing values to the new threads and getting their results like in the ThreadArguments.java example from class. Your workers can be subclasses of java.lang.Thread, with fields for giving the worker any needed values at start-up and other fields to store computed values to be collected after the worker finishes.

   Once you complete your program, you should be able to run it as follows. You can test your solution with the same input files we used on assignment 1. Just like with the previous assignment, since workers report ranges as they find them, you may get different orders from execution to execution; that's OK.

```
$ java Zerosum 5 report < input-3.txt
0 .. 5
27 .. 30
26 .. 30
7 .. 9
6 .. 9
0 .. 9
13 .. 14
19 .. 19
... lots of output lines omitted ...
37 .. 95
29 .. 95
1 .. 95
32 .. 86
86 .. 91
68 .. 91
```

[1] There's not really a "main" thread. Here, I just mean the thread that started running in main, the one that's not a worker thread.

```
42 .. 91
14 .. 96
2 .. 96
Total: 225
```

Since this program is written in Java, feel free to try it out on an EOS Linux machine or on your own system. If you own a system with lots of cores, you may be able to see lots of speedup. Before you turn in your work, be sure to try it out on an EOS Linux system, without any special compiler flags. Although your code will probably run on just about any system, we'll still be testing it on an EOS Linux machine.

When you're done, turn in an electronic copy of your **Zerosum.java** using the assignment_2 submission locker on our Moodle page.

3. (40 pts) For this problem, you're going provide similar functionality to the client/server program from homework assignment 1. This time, instead of using a server and interprocess communication to maintain the account balances, you will store the accounts and balances in a shared memory segment. A program that needs to access the account list can get and attach the shared memory segment. Then it can look at or modify the balance in any account just like any other data structure that's part of its memory.

   You will still need two different programs, but they won't be doing the same jobs as before. One of your programs will be called `reset.c`. Its job will be to read command-line arguments just like the server did in the previous assignment and create a shared memory segment containing the list of accounts and balances for each account. The `reset.c` program will just exit after creating the shared memory segment and filling it in with all the initial account balances.

   The other program will be called `account.c`. It will interpret a user command given in the command-line arguments and make requested changes to the account balances. You can also use a header file named `common.h` that can be included by both of your programs.

   Your `account.c` program will be used kind of like the client program from the last assignment. The user will run it with command-line arguments that say what to do with the account list. But, instead of having to send requests to a separate server program, it will just directly access and modify the contents of the account list stored in the shared memory segment. Then, other copies of the account.c program run on the same host will automatically see the changes to the balances.

   You should be able to run the account.c program in the following ways (just like you could run the client in homework 1):

   - `./account credit` *acct v*
     Running the account program like this should increase the balance in the account with the given name by the given non-negative value, *v*. As with the previous version of the command, the value is give as a floating point number and 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 in the account list or if the resulting balance would be larger than ten million. The program should print a line saying "success" if the command is successful, or "error" if the command is invalid.
   - `./account debit` *acct v*
     This is like the credit command, but it deducts the given non-negative amount from the balance in the given account. The value, *v* can be any floating point value, but it should be rounded to the nearest integer cent before the account balance is changed. The program should print a line saying "error" if the given name isn't the name of an account, if the value v isn't a non-negative floating point value or if the resulting balance is negative. Otherwise, it should print out a line with the word "success".

- `./account query` *acct*
  When the user enters this command, the program should print out the current balance of the account with the given name. Print it in dollars, with two digits to the right of the decimal point representing cents. If the given name doesn't match any of the account names, print a line with the word, "error".

Starter Files

The course website has starter files for your three source files. They are mostly empty, but they include a few functions for reporting errors and probably all of the includes you will need.

Error Conditions

The reset.c program should error check its command-line arguments just like the server from assignment 1. The command line arguments should describe between 1 and 10 accounts. If the given accounts aren't valid (e.g., duplicate account names, names that are too long, balance less than 0.00 or greater than 10,000,000.00), the program should terminate unsuccessfully and print the following usage message to standard error:

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

Error handling in the account.c program is simple. If the user enters command-line arguments that don't match one of the commands shown above (e.g., not enough arguments or a first argument that isn't query, credit or debit), the program should print a line with the word "error" to standard output.

For both programs, if an error is encountered in one of the system calls, (like not being able to create or attach its shared memory), you should print out a meaningful error message of your choice and then exit. These messages will also help you diagnose problems while you're developing your program.

Creating the Shared Memory

For this problem, you'll need to organize your account list into a single struct, with all the account names and balances stored in fields of the struct. We'll call this the AccountList sturct. Having the whole representation in this one struct is going make it easy to store in shared memory. To create the shared memory, we just need to ask for a segment that's the size of an instance of AccountList.

When you attach the shared memory, you can just cast the pointer you get back from shmat() to a pointer to AccountList. Then, you can easily access the contents of the shared memory via this pointer, as if the shared memory was just an instance of AccountList. Notice, the way we're using shared memory is a lot like how we use malloc(); we tell shmget() how much memory we need to store an AccountList structure, then we use the pointer returned by shmat() as a pointer to a new instance of AccountList.

The shared memory segment will persist across multiple executions of our programs. The first time you run reset.c, it will create the shared memory and fill it in based on the command-line arguments. After this, running account.c will get and attach that shared memory segment (without creating a new one). It will be able to look at the AccountList struct created by reset.c and change the account balances based on the command-line arguments.

The shared memory segment will let multiple programs access the list at the same time[2]. Like with our client server, we should be able to open multiple terminals on the same host and access the same account list from any of the terminals.

Sample Execution

Once your program is working, you should be able to run it as demonstrated below. Here, I'm showing how you can access the shared account list from multiple terminal running on the same host. The commands on the left are from one terminal session and those on the right are from a different terminal. I've ordered the commands top-to-bottom to show the order I ran them. For example, when we query the balance for the cash account on the right, we see the 100.00 balance set when this account was initialized on the left. The shell comments mixed in with the commands just give some explanation of what this example is doing (you don't need to enter them).

```
# Make an initial list of accounts.
$ ./reset cash 100.00 supplies 37.50 flowers 0.00 snacks 45.92 bribes 18 rainy-day 240.00


                        # From elsewhere on the host, we can see the
                        $ ./account query cash
                        100.00
                        $ ./account query supplies
                        37.50

# Back on the original terminal, try
# the credit command.
$ ./account credit snacks 14.08
success
$ ./account query snacks
60.00
$ ./account credit cash 0.12
success
$ ./account query cash
100.12

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

# Try some invalid arguments for the
# account program
$ ./account query abcdefg
error
$ ./account credit flowers abc.123
```

[2]If you really ran two copies of account.c at the same time, there's the potential for a race condition. If they both tried to modify the contents of shared memory at the same time, they might interfere with each other and leave it in an unknown state. We'll just ignore this potential problem for this assignment. Maybe we'll fix it on assignment 3.

```
error
$ ./account debit cash 50.00
error

# Then some invalid arguments for the
# reset program.
$ ./reset a 5.0 b 10.0 c
usage: reset (<account-name> <balance>)+

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

$ ./reset a 1.00 b 1.00 a 1.00
usage: reset (<account-name> <balance>)+
```

Naming Your Shared Memory

You will need your own key (number) to create a shared memory segment. For our in-class example, I used a hard-coded key I just made up, 9876, but we can't all use this key if some of us might be developing on the same, shared systems. Instead, let's use the ftok() function to assign different keys to each member of the class. This function maps a pathname in the filesystem to a unique key. Just use the name of your AFS home directory as the path, and you should get a key that's unique to you.[3]

Resetting Your Shared Memory

Your account.c program will use the same segment of shared memory every time you run it. If the contents of this memory get corrupted (say, because of a bug while you're developing your program), you will need to reset the contents of this memory. Your reset.c program will make this easy. You should be able to just re-run reset whenever you need to erase the old account list and overwrite it with a new one.

If you need to completely delete your shared memory segment so reset.c can make a new one, there are a couple of command-line utilities to help you out. The `ipcs` shell command will list all your shared memory segments (along with some other IPC objects). You can delete a shared memory segment by running `ipcrm` with the `-m` option followed by the id of the segment you want to delete. `ipcs` reports the ids for each segment, along with the segment size. If you own multiple shared memory segments (maybe created by other programs begin run by you), you can use the size to help you decide which one to delete. When I tried this, the shared memory segment for my account list was much smaller than the other segments owned by me.

Accessing Shared Memory

Like the message queues we used on assignment 1, shared memory only works for programs running on the same host. If you access the shared memory from two different terminals (like in the example) you need to make sure terminals are logged in on the same host. If you're using remote.eos.ncsu.edu, use the hostname command to figure out what machine you're logged in on and login on that particular

[3]It's important that you get this path right, or the TA may not be able to grade your work. If you're not sure what your home directory is in AFS, you can enter `echo $HOME` on an EOS Linux machine. Also, for ftok(), there's sometimes some confusion about what to do with the proj_id parameter. It's just there to let you get multiple unique keys based on the same pathname. Since you just need one key, you can use any constant you want for this parameter, but be sure to use the same constant every time in order to get the same key.

host for any other terminals you're using. Or, if you use a Linux desktop machine, it'll be easy to open as many terminals as you want on the host you're sitting in front of.

Submitting Your Work

When you're done with your program, submit three source files. This should include your common header file, **common.h**, the implementation file for the reset program, **reset.c** and the implementation file for the account program, **account.c**. Submit all of these to the `assignment_2` assignment in Moodle.