Assignment 3
EEL4732/5733 Advanced System Programming

In this assignment, you are going to gain practice on 1) designing a deadlock-free solution for resource sharing, 2) signal handling, and 3) using mmap for inter-process data sharing.

You are going to simulate electronic fund transfers (EFT) between bank accounts. We will assume that there is just one bank and several accounts. Your program will take four command line arguments:

```
$ ./eft <concurrencymode>  <numworkers> <signalmode> input.txt
```

There are two possible values the concurrencymode argument can take: multi_process and single_process. If the mode is specified as multi_process (single_process) then each worker will be created as a separate process (thread). In the case of a multi_process mode, you need to use the **mmap** system call to set up the shared memory region. *We require that you use anonymous mapping (no backing file)*. Also, you will need to initialize the synchronization primitives to be process shared. In the case of a single_process mode, you are not going to use mmap and will not need to set the synchronization primitives to be process shared. However, the expected output will be the same for both modes as we explain below.

The second argument, numworkers, will be a positive integer. If the mode is single_process (multi_process) then it will determine the number of worker threads (processes) to be created.

The third argument, signalmode, can take two values: none and sigint. If the specified value is node then the application will not handle sigint, i.e., default disposition will be in place. The output will be displayed on standard output if the application is not terminated by SIGINT.
If the specified value for this mode is sigint then the application will handle SIGINT by defining a signal handler and installing it using the signal or sigaction system calls. In the sigint mode the application will not display any output on standard output unless it receives a SIGINT at some point. If the application receives SIGINT before generating the whole output, it should continue execution until all the data is output and then terminate. If the application does not receive any SIGINT by the time all the output has been generated then it should block waiting for a SIGINIT using the pause system call until it receives it.

The fourth argument is an input file in the form of:

```
AccountNo1 <space> initialBalance1
AccountNo2 <space> initialBalance2
..
AccountNoN <space> initialBalanceN
Transfer <space> accountNoFrom1 <space> accountNoTo1 <space> Amount1
Transfer <space> accountNoFrom1 <space> accountNoTo2 <space> Amount2
…
Transfer <space> accountNoFrom1 <space> accountNoTo1 <space> Amount1
```

which first lists the accounts in the system along with the initial balances and then lists the transfers between accounts. You can assume that all transfers refer to the existing accounts and all initial balances and the transfer amounts are nonnegative integers.

The <numworkers> argument will determine the number of worker threads/processes that will run in parallel. Note that the main thread will initialize the accounts, read the input file and assign work (EFTs) to worker threads/processes in a round-robin fashion until all transfers are processed. It is possible that an account may be overdrawn and get a negative value while processing. Your program should output on the standard output the amount in each account (in the order specified in the input file) once all transfers are completed.

So, as an example, assuming the input file is as below

```
1 1000
2 50
3 400
4 150
Transfer 1 2 200
Transfer 1 4 50
Transfer 2 3 100
```

your program should produce the following output regardless of the number of worker threads/processes specified (assuming either none is specified for the signalmode or a SIGINT has been delivered to the application):

```
1 750
2 150
3 500
4 200
```

To get full credit, your solution should maximize concurrency, be free of race conditions and deadlocks, and produce the correct output. Note the following solutions will not be accepted:

Unacceptable solution 1: The problem with this solution is that it does not maximize concurrency as each transfer acquires a global mutex that is held during the transfer and blocks other transfers even if they involve different accounts.

```
Transfer(A1,A2,amount):
lock(mutex)
A1.value -= amount
A2.value += amount
unlock(mutex)
```

Unacceptable solution 2: The problem with this solution is that it will leave the global state of accounts in an inconsistent state, e.g., if somebody wants to display all accounts between two transfers, the transferred amount will not be accounted for. Also, this is similar to letting a dining philosopher eat with a single fork 😊

```
Transfer(A1,A2,amount):
lock(A1.mutex)
A1.value -= amount
unlock(A1.mutex)
lock(A2.mutex)
A2.value += amount
unlock(A2.mutex)
```

You should devise a solution that prevents deadlocks by either breaking circularity or by allowing both accounts to be acquired at once before performing the transfer.
You can use mutexes and condition variables and/or semaphores for synchronizing your threads.
Please submit your files (source, README, and a Makefile) on CANVAS by the due date.