Computer Systems Principles

Bank Simulation

Contents

Bank Simulation Project Assignment

Overview

This project assignment will exercise your understanding of process manipulation in C and how to build robust programs. You must complete all the exercises below using a text editor of your choice. Make sure you follow the instructions exactly. The actual code your write is fairly short,

however, the details are quite precise. Programming errors often result from slight differences that are hard to detect. So be careful and understand exactly what the exercises are asking you to do.

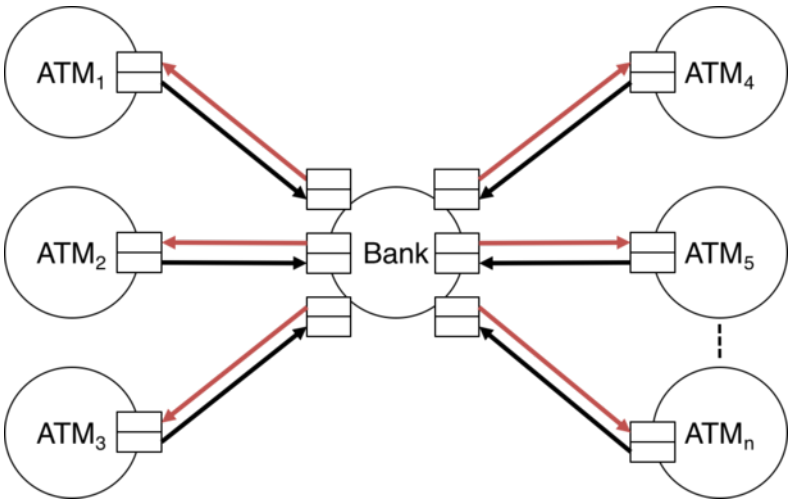

Figure 1: Bank Simulator Architecture

The goal of this project is to complete the implementation of a bank/atm simulator. The bank/atm simulator will fork a number of processes that represent a bank and n ATM terminals that can respond to several different transactions provided by an ATM user as shown in Figure 1. The ATM transactions are *simulated* through a trace file containing a sequence of ATM transactions that each ATM will read and send to a bank process over a "pipe". The bank process will receive commands from the ATMs and will perform the proper transaction received from the ATMs. The bank must then respond back to the ATM indicating the success or failure of the transaction.

Here is an example of running the bank simulator program:

```
$ ./banksim test/4_2_10.trace
not enough funds, retry transaction
Account 0: 4692
Account 1: 5120
```

This shows a failure in one of the transactions trying to perform a withdrawal or transfer on an account with insufficient funds. It then shows the final balance of each of the accounts. You can run the bank simulator in debug mode to show the transactions between the ATMs and the bank:

```
$ BANKSIM_DEBUG=1 ./banksim test/4_2_10.trace
ATM/SEND[3] CONNECT 3 -1 -1 -1
BANK/RECV[0] CONNECT 3 -1 -1 -1
BANK/SEND[0] CONNECT 3 -1 -1 -1
ATM/SEND[2] CONNECT 2 -1 -1 -1
BANK/RECV[0] CONNECT 2 -1 -1 -1
```

```
BANK/SEND[0] CONNECT 2 -1 -1 -1
ATM/RECV[2] OK 0 -1 -1 -1
ATM/RECV[3] OK 0 -1 -1 -1
ATM/SEND[3] DEPOSIT 3 -1 0 5000
BANK/RECV[0] DEPOSIT 3 -1 0 5000
BANK/SEND[0] DEPOSIT 3 -1 0 5000
ATM/RECV[3] OK 0 -1 0 5000
ATM/SEND[3] TRANSFER 3 1 1 39
ATM/SEND[1] CONNECT 1 -1 -1 -1
BANK/RECV[0] TRANSFER 3 1 1 39
ATM/RECV[3] NOFUNDS 0 1 -1 39
not enough funds, retry transaction
ATM/SEND[3] TRANSFER 3 0 1 95
...
Account 0: 4692
Account 1: 5120
```

We elaborate on the debug output generated by this program in our discussion of the modules later. You can generate new trace files using the provided utility program called `twriter`. You can generate a trace for 10 ATMs, 5 accounts, and 100 transactions like this:

```
$ ./twriter 10 5 100
```

It will generate a new trace file named `10_5_100.trace`. You can then run the bank simulator on the generated trace file. If you want to see the ATM transactions generated by `twriter` you can use the other utility program, `treader`, which will read a trace file and print the transactions to standard output. You must use these two executables to interact with trace files because trace files are saved in a binary format. If you are interested in how we implemented `twriter` and `treader` take a look at `twriter.c` and `treader.c`.

Part 0: Project Startup

Please download the project startup tarball from the course web page. If you do not know what a "tarball" is you should read up on it. To do this, open a web browser by clicking on the left-most menu button followed by "Internet". We have installed Firefox for you to use. You should do this within your virtual machine so you can download the tarball to your virtual machine disk. You should download the tarball to your home directory.

Once you have the tarball in your home directory you can execute the following command:

```
$ tar xzvf bank-proj-student.tgz
```

This will *unarchive* the contents of the tarball and you will see the `bank-proj-student` directory. You can then go into that directory from the terminal using `cd` (change directory):

```
$ cd bank-proj-student
```

Part 1: Understand The Code

We provide you with starter code for this assignment. Your first task is to **read** through each of the provided source files in detail so you understand the structure of the code. This is particularly important for this assignment because of the number of source files involved. The `banksim` program depends on a number of "modules" for its implementation. The following is a description of each of these modules and their corresponding source files.

Command Module

The Command module consists of the following source files:

- `command.h`
- `command.c`

Let us first take a look at the `command.h` file. The command modules provides support for creating and manipulating transaction commands that can be sent between ATMs and the bank. A transaction command "buffer" is a sequence of bytes with a specific format as depicted in Figure 2. In particular, `c` is a byte representing the command (i.e., saying which command this is), `iiii` is a 4-byte integer representing the ID of the sender, `ffff` is a 4-byte integer representing the *from* bank account, `tttt` is a 4-byte integer representing the *to* bank account, and `aaaa` is a 4-byte integer representing the amount of the transaction. The total number of bytes in a command buffer is 17. Not all bytes in the command buffer will be used for every command. For example, a deposit command requires only a *to* account and an *amount*, but not a *from* account. Fields in the command buffer that are not used receive the value of `-1`.

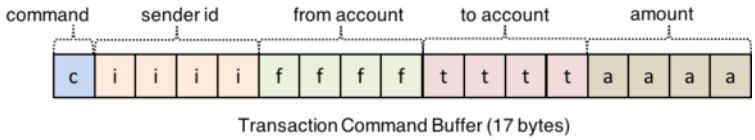

Figure 2: Bank Simulator Command

The first item of interest is the struct defining this representation of a command:

```
typedef struct command {
  byte cmd [1];
  byte id  [4];
  byte from[4];
  byte to  [4];
  byte amt [4];
} Command;
```

Notice that each part is a small array of `byte`, where `byte` has been defined as `unsigned char`. The reason for using an array of bytes for each part of a command is to get the parts to pack

together into the form presented above. If arrays were not used then the compiler would insert padding to align `int` values, etc. The next item of interest is `MESSAGE_SIZE`, which is used when defining "buffers" to hold commands.

The next set of definitions indicate the available commands and their values. These are the possible message types that can be sent between ATMs and the bank. Each message type has a fixed value representing the command. For example, a `TRANSFER` command is represented by the value `5`. Following these are several macro definitions that simplify the construction of messages. We did not cover macros in great detail. However, they should be easy to understand. If you want to know more about C macros you can read this article. They are be invoked like a function:

```
MSG_DEPOSIT(cmd, 0, 1, 200)
```

This creates a new deposit command into the command buffer `cmd` where the ATM id is 0, the account is 1, and the deposit amount is 200. Following the macros you will see a type definition for the command type `cmd_t`.

Lastly, we provide functions for packing a command into a command buffer and unpacking a command from a command buffer. The `cmd_pack` function receives a command buffer as its first argument followed by each of the important parts of the command. These parts are described in the source documentation. The `cmd_unpack` function takes a filled command buffer and pulls out the corresponding parts. Again, the description of the parts are described in the source documentation. The macros use the `cmd_pack` function to pack specific commands. Here is an example:

```
Command cmd;
cmd_pack(cmd, DEPOSIT, 0, -1, 1, 200);
cmd_t c;
int i, f, t, a;
cmd_unpack(cmd, &c, &i, &f, &t, &a);
```

This example packs a deposit command into the command buffer `cmd`. That is, it takes each of its arguments and "stuffs" them into the 17 byte command buffer `cmd`. Although we could use the `cmd_pack` function directly, as we show in the above example, you should prefer to use the macros instead (`MSG_DEPOSIT`). You can then use the `cmd_unpack` function to retrieve the values from the command buffer `cmd`. Each command requires a command type (e.g., `DEPOSIT`), an ID (which ATM/bank is sending the message), a from/to account, and an amount. If a command does not use a parameter then that parameter should bet set to -1, as we did in the above example for a deposit message since it does not have a *from* account.

Lastly, we include the `cmd_dump` function that will dump out a message to standard output. The function will print the debug information only if you set the `BANKSIM_DEBUG` environment variable. See the function documentation for an example. We recommend putting in calls to `cmd_dump` after reading each command and before writing each one, in both `atm.c` and `bank.c`. We used the strings `ATM/SEND`, `ATM/RECV`, `BANK/SEND`, and `BANK/RECV` to mark the different cases in the output.

Errors Module

The Errors module consists of the following source files:

- `errors.h`
- `errors.c`

This module defines success/errors that can occur in the bank simulator. You will see a list of definitions at the start of `errors.h` which define each of the possible errors that can be encountered. You will use these error codes when you complete the implementation of both the bank and ATM. You set an error code with the `error_msg` function. After calling this function you can retrieve the error code using `error_type` and the error message using `error_msg_str`. You can also print the errors message using the `error_print` function. The implementation is straightforward - you should take a look to see how it works.

Trace Module

The Trace module consists of the following source files:

- `trace.h`
- `trace.c`

This module implements the reading and writing of trace files. A trace file contains a list of commands that the ATM processes will read and send to banks. Each ATM will send a command only if that command matches its ID. To open a trace file you use the `trace_open` function with a given path to a trace file. When you are done reading the trace you must call `trace_close`. The first 8 bytes of the trace file record the number of ATMs and the number of accounts. You can retrieve these values (after the trace file has been opened) using the functions `trace_atm_count` and `trace_account_count` which will return the ATM count and account count, respectively. The `trace_read_cmd` function will fill in the command buffer that is passed to it with the next command found in the trace file. This function will return 0 when there are no more commands to read.

ATM Module

The ATM module consists of the following source files:

- `atm.h`
- `atm.c`

This module exports only a single function called `atm_run`. This is the main entry point that simulates an ATM terminal. The function has 4 parameters. The first parameter, `trace`, is the path to the trace file from which the ATM will read commands. The second parameter, `bank_out_fd`, is the file descriptor that the ATM will use to write commands to the bank. This file descriptor represents the write part of a pipe opened using the `pipe` system call. The third parameter, `atm_in_fd`, is the ATM's input file descriptor. This is where the ATM will read commands

(responses) coming from the bank process. It is the read end of a pipe. The last parameter is the ID of the ATM. The `atm_run` function is called from `main` in the `banksim.c` file after the ATM processes are created.

We provide the implementation of `atm_run`. Briefly, this function opens the trace file and reads each message contained in the trace file, passing it to the `atm` function for processing. You will note in this function the heavy error checking that we perform to make the program more robust. In particular, this function returns with an error code if the trace file can't be opened, prints a message if the account is unknown, prints a message if there were no funds, and returns the error code if the return value from the `atm` function is not successful.

The `atm` function is responsible for processing the command it receives from the trace file. In particular, it will need to send the following commands to a bank when they are read from the trace file:

- CONNECT
- EXIT
- DEPOSIT
- WITHDRAW
- TRANSFER
- BALANCE

You must implement this function using the modules we have provided for you. More details on what you must do are given below.

Bank Module

The bank module consists of the following source files:

- `bank.h`
- `bank.c`

This module exports the following functions:

- `bank_open`
- `bank_close`
- `bank_dump`
- `run_bank`

The `bank_open` and `bank_close` functions open and close a bank respectively. This is required because the bank creates a data structure to represent accounts. The `bank_dump` function prints out the accounts and their balances. We will use this to check your solution against ours to make sure that your implememtation implements the transactions correctly. The `run_bank` function is the main entry point that simulates a bank. The function has 2 parameters. The first parameter, `bank_in_fd`, is an array of file descriptors. There is on file descriptor for each ATM - this is where the bank will read messages from ATMs. The output end of each of these pipes is provided to a particular ATM so that it can write messages to the bank. The second parameter is an array of file descriptors that are the output ends of the pipes associated with each

ATM. To write to an ATM with ID *i* you simply index `atm_out_fd` with *i* and write the message to that pipe. We provide you the implementation of `run_bank`.

The `run_bank` function is simple. It loops until the number of ATMs has reached 0. If the number of ATMs is 0 it means we have received `EXIT` messages from all the ATMs and so the bank can exit. The main loop in this function first determines an ATM that appears to have input, using `find_ready_atm`, which itself uses the `poll` system call. We invite you to read the man page for `poll`. It deals with *sets* of file descriptors and allows you to wait until any one of a selected set of file descriptors is ready for I/O (or has an error condition). Here we are checking only for input, from the currently open pipes from the ATMs. We said "appears to be ready" because if the pipe has been closed on the ATM side, the bank end will appear ready since it is trying to indicate EOF (`POLLHUP`).

`bank_run` then tries to read a command from the corresponding `bank_in_fd` file descriptor using the `checked_read`. We invite you to understand `checked_read` since it deals with important cases of reading from a file descriptore with the `read` system call, including EOF, reads that deliver only a part of the data expected, and errors. To learn more about `read` use the `man` command. It takes three arguments. The first argument is the file descriptor to read from, the second is a buffer (block of storage, generally an array, in this case an array of bytes) to read the bytes into, and the third is the number of bytes to read.

Next, we call the `bank` function passing it the output file descriptors for each ATM, the command buffer we just read, and a pointer to the number of ATMs (which we adjust inside the `bank` function). You need to complete the implementation of the `bank` function. The details of what you must do are given below.

Bank Simulator

This is the main entry point into the program. It consists of the `banksim.c` source file. We have provided part of the implementation of the `main` function. The goal of `main` is to create the processes for the ATMs and the bank. The main process will wait for the other processes to complete before it completes. In addition, this is where we need to create the pipes that are used for interprocess communication between ATMs and the bank. You are required to complete the implementation of this function by including the appropriate calls to `fork` and calling the `atm_run` and `run_bank` functions. You will notice two `TODO` entries in this function. You must write your solution exactly where the comments tell you to. The first part of the implementation is forking the ATM processes, the second is forking the bank process. More information about what you must do is given below.

The code for this part is very small, but *very* specific. Make sure you follow the instructions in the `banksim.c` file exactly to provide a successful implementation.

The sbanksim Program

The `sbanksim` program is our solution for this assignment. You are welcome to look at the guts (assembly) of this file. If you are able to figure out what we did - well, then you probably deserve a good score. You can run this in the same fashion as the executable that is generated for your solution.

Part 2: Compiling the Project

To compile the project you need to use the following command:

```
$ make
```

This will produce several *object files* and two *executable files*. There are three executable files:

- `banksim`
- `twriter`
- `treader`

The `banksim` executable is your solution to this assignment. You can also remove all the generated binary files using make:

```
$ make clean
```

Part 3: Testing

We have provided tests that you can run to see if you are on the right track toward a solution. Each submission will also be assessed by peers, so your code will be inspected and commented upon. Make sure you review the academic honesty policy on the course website and what is included as part of this project's documentation toward the end.

The `test/public-test.c` file contains tests using the check C unit testing framework. You do not need to know every detail of the check framework to use it. Each test is defined by using special macros that auto-generate test functions. If you want to know more about C macros you can read this article. You can run the tests with the following command:

```
$ make test
```

You are welcomed and **encouraged** to introduce additional tests. To add a new test you should copy one of the existing tests and add your own. After you add your own test you need to add it to the test suite. To do this you simply extend the `tester_suite` function to include an additional `tcase_add_test(tc_inc, <your test name>);`

NOTE: The tests for this project take some time to complete. You should give the testing framework at least 15-20 seconds to complete before pulling the plug.

Task 1: Complete ATM

Your first task is to implement the `atm` function in `atm.c`. The `atm` function is passed the following arguments:

- `int bank_out_fd`
- `int atm_in_fd`
- `int atm_id`
- `Command *cmd`

The *bank_out_fd* argument is the output pipe that you must use to communicate to the bank server. The *atm_in_fd* is this ATM's input pipe file descriptor from which the ATM will read in order to receive messages from the bank process. The *atm_id* is the ID of this ATM, used to uniquely identify the ATM to the bank process. The *cmd* is the command buffer that contains the next command coming from the trace file.

The general idea is that the ATM process must identify which commands coming from the trace file pertain to this ATM and handle them appropriately by sending the bank process those commands. Remember, the trace file contains commands for *all* the ATMs, so we need to determine if the command is associated with this ATM by checking the ID contained in the command. We provide the following code that gives you some starting definitions that you must use in your implementation:

```
byte c;
int i, f, t, a;
Command atmcmd;
```

The variables *c*, *i*, *f*, *t*, and *a* are used to extract the individual parts of a command. The *atmcmd* buffer is used to build messages that are to be sent and received to and from the bank server. We unpack the command coming from the trace file for you:

```
cmd_unpack(cmd, &c, &i, &f, &t, &a);
```

After the *cmd_unpack* function is called, the given arguments will be filled in with the values contained in the trace file command. We also define a *status* variable:

```
int status = SUCCESS;
```

that you should use in your implementation to return the correct error code if there is a problem. We initialize this to *SUCCESS*. This function should return the *status* variable in the end. You must then perform the following todos in the order described:

TODO 1: First, you must check to make sure that the ID in the unpacked command is the same as this ATM's ID. If it is not the same ATM ID then you must return the ERR_UNKNOWN_ATM error code. This will be used by the `atm_run` function to skip over that trace command as it is not to be processed by this ATM.

TODO 2: Next, you need to handle the logic for each of the commands: CONNECT, EXIT, DEPOSIT, WITHDRAW, TRANSFER, and BALANCE. Each of these commands is handled the

same way by the ATM. You will need to use the *checked_write* function to send the *cmd* received from the trace file to the bank process. The general format of the call to write looks like this:

```
status = checked_write(bank_out_fd, cmd, MESSAGE_SIZE);
```

This will write *MESSAGE_SIZE* bytes from the *cmd* command buffer to the *bank_out_fd* output pipe connected to the bank process. This call will return a status code from *check_write*. If that code is not *SUCCESS*, you should return the status code, otherwise continue.

After you write the command to the bank and it is successful, you must then read the response from the bank using the *checked_read* function, into the *atmcmd* command buffer. In this case you want to read a command in from the bank. The bank will be writing to the specific ATM's pipe file descriptor, *atm_in_fd*. The general format for the read looks like this:

```
status = checked_read(atm_in_fd, atmcmd, MESSAGE_SIZE);
```

This call will block until the bank has written a response to the ATM. It will return a status code. Again, if it is not *SUCCESS*, you should return the status code, and otherwise continue.

checked_write and *checked_read* are "wrappers" for the write and read system calls, all of which you should read about (*man* pages are good!). In particular, the ATM versions of these functions deal with the possibility of read/write calls that do not transfer all the requested bytes in one go. While partial transfers are unlikly for messages this small, they are theoretically possible and it is best if the code handles them. These functions also detects errors (negative values returned by the underlying system calls).

If you have successfully received a response from the bank then you need to use *cmd_unpack* to unpack the command into its individual parts using the variables declared at the beginning of this function. You will need to check the command type in the response for the following cases:

- *OK*: Everything was successful and the bank processed the transaction correctly. If so, then you should set *status* to SUCCESS.
- *NOFUNDS*: The bank could not complete the transaction because of insufficient funds. In this case, you should set *status* to *ERR_NOFUNDS*.
- *ACCUNKN*: The bank could not complete the transaction because an account that was used is unknown (invalid account).

If you receive any other command from the bank then you must use the *error_msg* function (defined in *errors.h*) to indicate that you received an unknown command (*ERR_UNKNOWN_CMD*) with an appropriate error message. You should then set the status to *ERR_UNKNOWN_CMD*.

TODO 3: Lastly, if you receive a command from the trace file that is not any of the command types mentioned above you must use the *error_msg* function (defined in *errors.h*) to indicate that you received an unknown command (*ERR_UNKNOWN_CMD*) with an appropriate error message. You should then set the status to *ERR_UNKNOWN_CMD*.

At the end of this function you should return the *status*.

Task 2: Complete Bank

Your second task is to implement the `bank` function in `bank.c`. The `bank` function is passed the following arguments:

- `int atm_out_fd[]`
- `Command * cmd`
- `int *atms_remaining`

The *atm_out_fd* is an array of all the ATM output pipe file descriptors. The bank will use these to communicate to each of the connected ATMs. This array is indexed by the ATM's unique ID. The *cmd* command buffer contains the command that was received from an ATM. The *atm_count* is the number of ATMs that are still running.

The general idea is that the bank will receive commands coming in from each of the ATMs that are available. The bank will then process the command and execute the appropriate transaction (e.g., deposit, transfer). The bank must then send to the ATM that sent the command a response that will allow the ATM to proceed or report an error if there was a problem. We provide the following code that gives you some starting definitions that you must use in your implementation:

```
cmd_t c;
int i, f, t,a;
Command bankcmd;
cmd_unpack(cmd, &c, &i, &f, &t, &a);
int result = SUCCESS;
```

The *c*, *i*, *f*, *t*, and *a* variables are then used for unpacking the command received from the ATM (see documentation for the *cmd_unpack* function in *command.h* for more details). The *bankcmd* command buffer is used to build command messages that are sent to an ATM. We then unpack the command using the *cmd_unpack* function. Then, we initialize the *result* status with SUCCESS. The *result* will be returned at the end of this function. You must then perform the following todos in the order described:

TODO 1: First, you need to check that the ATM ID unpacked from the received command is a valid ATM. You can easily do this using the *check_valid_atm* function. If the return value of this function is not *SUCCESS*, then you should return *ERR_UNKNOWN_ATM* - the result from the *check_valid_atm* function. There is nothing more that can be done as we can't respond back to the ATM as it is an invalid ID - in the real world the ATM should be shutdown and a technician will need to service it (bad ATM!).

TODO 2: Next, you will need to perform different actions for each of the different command types: CONNECT, EXIT, DEPOSIT, WITHDRAW, TRANSFER, and BALANCE.

CONNECT: Each ATM will send a CONNECT message to the bank before performing any other transactions. If the bank receives this message, the response back to the ATM is simple. It should create an *OK* message type using the *MSG_OK* macro (see *command.h* for details). The ID of the bank can simply be *0*. The message can simply use the *f*, *t*, and *a* variables received from the ATM message. You should use the *check_write* function to write to the correct ATM (using the ID of the

ATM received in the incoming message - in variable *i*), and save its returned value in *result*.

EXIT: Each ATM will send an EXIT message to the bank when it has completed all the transactions in the trace file. For this case you must decrement the *atms_remaining* passed in to the `bank` function and write an *OK* message back to the ATM performing the same checks as you did in the CONNECT case. (Be sure to decrement the int at the other end of the pointer, not the pointer itself!) When *atms_remaining* reaches 0 it indicates that the bank process has received an EXIT for all ATMs and should quit. You do not need to implement that part of the logic as it is done for you in the *run_bank* function.

DEPOSIT: If a DEPOSIT command is received you must check the *to* account using the *check_valid_account* function to make sure that it is a valid account. If it is not you need to send a *ACCUNKN* message back to the ATM (using the *MSG_ACCUNKN* macro) to indicate that the account was invalid. Use *checked_write* as in the previous cases. If the account is valid then you perform the update on the account by incrementing the account with the amount: `accounts[t] += a;`. The *accounts* variable is the array containing the balance of each of the accounts. Lastly, send an *OK* message back to the ATM using *checked_write* as in the previous cases.

WITHDRAW: If a WITHDRAW command is received you should perform the same general logic as you did for DEPOSIT except you will need to check that the *from* account is correct and verify that the amount requested for withdrawal is available in the account. If it is not you will need to send the ATM a command indicating that there is insufficient funds (*NOFUNDS*). If everything is successful you must respond to the ATM with an *OK* message. Make sure you set the *result* variable correctly.

TRANSFER: If a TRANSFER command is received you must perform the same general logic as you did for DEPOSIT and WITHDRAW except that you need to check *both* accounts for validity and that the amount being transferred is available in the *from* account. You will need to respond appropriately for unknown accounts and insufficient funds. Send an *OK* message to the ATM if all is successful and make sure you set the *result* variable correctly.

BALANCE: If a BALANCE command is received you must check the *from* account to make sure it exists. If not, respond to the ATM with a command that indicates that the account is unknown. If successful, you should send the ATM an *OK* message filling in the *amount* with the balance in the associated account. Again, set the *result* variable appropriately.

Lastly, if the command received does not match any of the expected commands above, then you must use the *error_msg* function to indicate an `ERR_UNKNOWN_CMD` with an appropriate error message. The *result* variable should be set with `ERR_UNKNOWN_CMD`.

Return the *result* variable at the end of this function.

Task 3: Complete BankSim

Your final task is to use the fork system call to fork ATM processes and a bank process. We provide some starter code in the *main* function in this file. In particular, we check to ensure that the arguments passed in on the command line are correct. We declare some initial variables to hold

the status result of the function calls, the number of ATMs, and the number of accounts. We then open the trace file to read in the number of ATMs and accounts. Lastly, we declare two arrays, *atm_out_fd* and *bank_in_fd*, that will hold the file descriptors for pipes out to the ATMs and in to the bank, for communication with each ATM. These will be passed to the bank process.

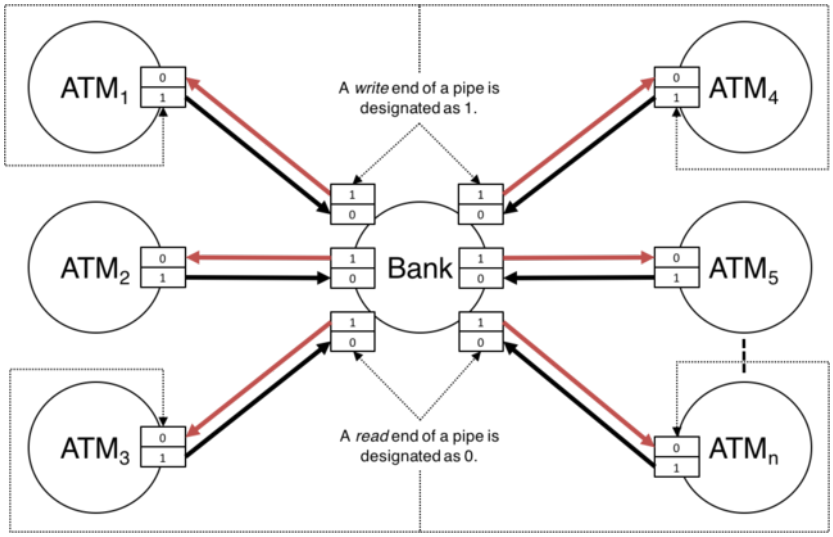

Figure 3: Bank Simulator Pipes

TODO 1: Your first todo is to create the ATM processes. We have labeled the start and end placement of your code in the provided source file. You should put your implementation between these comments. The body of this part of the implementation is a `for` loop that iterates on *i* from 0 to *atm_count*-1. Each iteration in the loop will declare an integer array of size 2 for the ATM's to-the-ATM pipe, called *to_atmfd*. You should call pipe on this array to create the pipe and store in the ith entry of *atm_out_fd* the to-the-ATM's output file descriptor, *to_atmfd[1]*. You should also declare an integer array of size 2 for the ATM's into-the-bank pipe, called *to_bankfd*. Call pipe on that and store *to_bankfd[0]*, the input file descriptor for the pipe, into the ith entry of *bank_in_fd*.

Next, you should call fork. If you are in the child process (the ATM process) you need to call close on the ATM's output pipe file descriptor, *atmfd[1]* (we do not need this in the ATM process) and the bank's input pipe file descriptor, *bankfd[0]* (we do not need this in the ATM process either). See Figure 3 to understand how the pipes are connected and their directionality. Next, you must call the *atm_run* function defined in *atm.c* with the file name of the trace file, the to-the-bank output pipe file descriptor, the to-the-ATM input pipe file descriptor, and the ATM's ID as the last argument, which is *i*. If the return value of *atm_run* is not `SUCCESS` then you should call *error_print* defined in *errors.h*. Lastly, call *exit(0)*.

If you are not in the child process you should simply close the to-the-ATM input pipe file descriptor and the to-the-bank output file descriptor as they are not necessary in the bank process.

TODO 2: Your second todo is to create the bank process. We have labeled the start and end

placement of your code in the provided source file. You should put your implementation between these comments. You should first call fork. See Figure 3 to understand how the pipes are connected and their directionality. Next, call the *bank_open* function, providing it the *atm_count* and the *account_count*. After the bank is open you should call the *run_bank* function, passing it the bank's input pipe file descriptor array and ATM output pipe file descriptor array. If the return status from this function is not `SUCCESS`, then use the *error_print* function to print the error. Lastly, call *bank_dump* to dump the account balances, followed by *bank_close* to close the bank, then finally *exit(0)* to exit the bank process.

We have provided the code at the end of the *main* routine to wait for the child processes to complete using the *wait* system call. You do not need to make any additional changes here.

BankSim Running and Testing

In addition to running the unit tests you should also verify your implementation against ours. Create several trace files and run them with your implementation and ours. If your implementation produces the same result as our implementation then you are in good shape. It is important to try runs using `banksim` directly in addition to doing `make test`!

Debugging Hints

First, see our notes above about the `cmd_dump` function. It is very useful in the context of this assignment.

More generally C programs, especially ones that fork child processes, can be tricky to debug. You may use the `valgrind` tool to identify pointer-related problems in your code and any memory leaks (allocating memory without freeing it). To run `valgrind` from the command line you do this:

```
$ valgrind ./banksim test/4_2_10.trace
```

This will report any invalid access to memory and any memory that had been allocated and not freed before the program terminated. If your program tries to access memory in a way it is not allowed to do, you will likely see `segmentation violation (core dump)` as the only output after running your program. Make sure you use `valgrind` to help better understand where things went wrong. Here is a list of how to go about debugging your C code:

1. Use `printf` to output debugging information. Do not underestimate the usefulness of this simple method of debugging! (See also `cmd_dump`!)
2. Use `valgrind` to narrow the scope of where your problem is (which function), then use the `printf` method. This is probably useful only on the top process, not on child processes.
3. Use `gdb` if you are really stuck and need to step through your program one line at a time. There are lots of resources online that walk you through finding segfault errors. (This will not work easily for debugging forked child processes – the `printf` and `cmd_dump` approach is better.)

Assessors will check your programs to make sure that you do not have any memory leaks.

Because this is a very complicated assignment involving several different processes, you should use `printf` to understand better what your processes are doing. In addition, you should start simple by generating trace files that contain a *single* ATM and a *single* account. This will help you isolate bugs in the simplest case. Once you have that working correctly you should move on the larger trace files.

Submission Instructions

You must submit your assignment as a *tarball*. After you complete the assignment you need to run the following command from your `bank-proj-student` directory:

```
$ make tar
```

This will create the file `bank-submit.tgz` which you need to upload to the assignment activity in Moodle. **Make sure you add the submission tar, not the original one!** Checking that you submitted the correct tarball is part of the assignment. You should **double check** that you submitted the correct tarball by downloading what you submitted to Moodle and extract the contents. Please submit your assignment to Moodle by the assigned due date. Please make sure you have followed all the instructions described in this assignment. Failure to follow these instructions precisely will likely lead to considerable point deductions and possibly failure for the assignment.