Introduction

The goal of this assignment is to become familiar with low-level POSIX threads, multi-threading safety, concurrency guarantees, and networking. The overall objective is to implement a server that simulates the behavior of a Private Branch Exchange (PBX) telephone system. As you will probably find this somewhat difficult, to grease the way I have provided you with a design for the server, as well as binary object files for almost all the modules. This means that you can build a functioning server without initially facing too much complexity. In each step of the assignment, you will replace one of my binary modules with one built from your own source code. If you succeed in replacing all of my modules, you will have completed your own version of the server.

For this assignment, there are four modules to work on:

- Server initialization (`main.c`)
- Server module (`server.c`)
- PBX module (`pbx.c`)
- TU (telephone unit) module (`tu.c`)

It is probably best if you work on the modules in the order listed. You should turn in whatever modules you have worked on. Though the exact details are not set at this time, I expect that your code will be compiled and tested in the following configurations:

1. Blackbox tests using your `main.c` and your `server.c` (if implemented, otherwise my `server.o`), and my `pbx.o` and `tu.o`.
2. Blackbox tests using the modules you implemented, with mine for the rest.
3. Unit tests on your `pbx.c` in isolation.
4. Unit tests on your `tu.c` in isolation.

For each configuration, you will receive points for whatever test cases in that configuration are passed. If you are able to achieve some reasonable functionality in a module, it will probably be beneficial to submit it. It is probably not a good strategy to submit modules that are completely broken. It is definitely not a good strategy to submit code that does not compile, as this might prevent you from getting any points at all!

Takeaways

After completing this homework, you should:

- Have a basic understanding of socket programming
- Understand thread execution, mutexes, and semaphores
- Have an understanding of POSIX threads
- Have some insight into the design of concurrent data structures
- Have enhanced your C programming abilities

Hints and Tips

- We strongly recommend you check the return codes of all system calls. This will help you catch errors.
- **BEAT UP YOUR OWN CODE!** Exercise your modules with rapid and concurrent calls to ensure that they do not crash or deadlock. Ideally, besides basic sequential tests, you would write multi-threaded test drivers to achieve this. We will use tests of this nature in grading.
- Your code should **NEVER** crash. We will be deducting points for each time your program crashes during grading. Make sure your code handles invalid usage gracefully.
- You should make use of the macros in `debug.h`. In non-debugging, "production" use, your code should basically be silent, except perhaps to emit a one-line error message just before terminating in case a fatal error situation requires an abort. **FOLLOW THIS GUIDELINE!** `make debug` is your friend.

> *:scream: **DO NOT** modify any of the header files provided to you in the base code. These have to remain unmodified so that the modules can interoperate correctly. We will replace these header files with the original versions during grading. You are of course welcome to create your own header files that contain anything you wish.*

Helpful Resources

Textbook Readings

You should make sure that you understand the material covered in chapters **11.4** and **12** of **Computer Systems: A Programmer's Perspective 3rd Edition** before starting this assignment. These chapters cover networking and concurrency in great detail and will be an invaluable resource for this assignment.

pthread Man Pages

The pthread man pages can be easily accessed through your terminal. However, this opengroup.org site (http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html) provides a list of all the available functions. The same list is also available for semaphores (http://pubs.opengroup.org/onlinepubs/7908799/xsh/semaphore.h.html).

Getting Started

Fetch and merge the base code for `hw5` as described in `hw0`. You can find it at this link: https://gitlab02.cs.stonybrook.edu/cse320/hw5 (https://gitlab02.cs.stonybrook.edu/cse320/hw5). Remember to use the `--stategy-option=theirs` flag for the `git merge` command to avoid merge conflicts in the Gitlab CI file.

The Base Code

Here is the structure of the base code:

```
.
├── demo
│   └── pbx
├── hw5.sublime-project
├── include
│   ├── debug.h
│   ├── pbx.h
│   ├── server.h
│   └── tu.h
├── lib
│   └── pbx.a
├── Makefile
├── src
│   ├── globals.c
│   ├── main.c
│   ├── pbx.c
│   ├── server.c
│   └── tu.c
└── tests
    ├── basecode_tests.c
    ├── script_tester.c
    └── __test_includes.h
```

The base code consists of header files that define module interfaces, a library `pbx.a` containing binary object code for my implementations of the modules, a source code file `globals.c` that contains definitions of some global variables, and a source code file `main.c` that contains a stub for function `main()`. The `Makefile` is designed to compile any existing source code files and then link them against the provided library. The result is that any modules for which you provide source code will be included in the final executable, but modules for which no source code is provided will be pulled in from the library. The `pbx.a` library was compiled without `-DDEBUG`, so it does not produce any debugging printout. Also provided is a demonstration binary `demo/pbx`. This executable is a complete implementation of the PBX server, which is intended to help you understand the specifications from a behavioral point of view. It also does not produce any debugging printout, however.

Most of the detailed specifications for the various modules and functions that you are to implement are provided in comments that precede stubs for these functions in the various source files. In the interests of brevity and avoiding redundancy, those specifications are not reproduced in this document. Nevertheless, the information they contain is very important, and constitutes the authoritative specification of what you are to implement.

> *:scream: The various functions and variables defined in the header files constitute the **entirety** of the interfaces between the modules in this program. Use these functions and variables as described and **do not** introduce any additional functions or global variables as "back door" communication paths between the modules. If you do, the modules you implement will not interoperate properly with my implementations, and it will also likely negatively impact our ability to test your code.*

The function stubs that appear in the various source files have been commented out in the basecode. This is important, because when the linker does not see any definitions for these functions, it will link in binaries from the `pbx.a` library. If you choose to implement a function in one of the modules, you must uncomment the stubs for **all** the other functions in that module, otherwise the linker will still link the library version of that module, which will result in "multiply defined" errors at link time.

The PBX Server: Overview

"PBX" is a simple implementation of a server that simulates a PBX telephone system. A PBX is a private telephone exchange that is used within a business or other organization to allow calls to be placed between telephone units (TUs) attached to the system, without having to route those calls over the public telephone network. We will use the familiar term "extension" to refer to one of the TUs attached to a PBX.

The PBX system provides the following basic capabilities:

- *Register* a TU as an extension in the system.
- *Unregister* a previously registered extension.

Once a TU has been registered, the following operations are available to perform on it:

- *Pick up* the handset of a registered TU. If the TU was ringing, then a connection is established with a calling TU. If the TU was not ringing, then the user hears a dial tone over the receiver.
- *Hang up* the handset of a registered TU. Any call in progress is disconnected.
- *Dial* an extension on a registered TU. If the dialed extension is currently "on hook" (*i.e.* the telephone handset is on the switchhook), then the dialed extension starts to ring, indicating the presence of an incoming call, and a "ring back" notification is played over the receiver of the calling extension. Otherwise, if the dialed extension is "off hook", then a "busy signal" notification is played over the receiver of the calling extension.
- *Chat* over the connection made when one TU has dialed an extension and the called extension has picked up.

The basic idea of these operations should be simple and familiar, since I am sure that everyone has used a telephone system :wink:. However, we will need a rather more detailed and complete specification than just this simple overview.

The PBX Server: Details

Our PBX system will be implemented as a multi-threaded network server. When the server is started, a **master server** thread sets up a socket on which to listen for connections from clients (*i.e.* the TUs). When a network connection is accepted, a **client service thread** is started to handle requests sent by the client over that connection. The client service thread registers the client with the PBX system and is assigned an extension number. The client service thread then executes a service loop in which it repeatedly receives a **message** sent by the client, performs some operation determined by the message, and sends one or more messages in response. The server will also send messages to a client asynchronously as a result of actions performed on behalf of other clients. For example, if one client sends a "dial" message to dial another extension, then if that extension is currently on-hook it will receive an asynchronous "ring" message, indicating that the ringer is to be activated.

Messages from a client to a server represent commands to be performed. Except for messages containing "chat" sent from a connected TU, every message from the server to a client will consist of a notification of the current state of that client, as it is currently understood by the server. Usually these messages will inform the client of a state change that has occurred as a result of a command the client sent, or of an asynchronous state change that has occurred as a result of a command sent by some other client. If a command sent by a client does not result in any state change, then the response sent by the server will simply report the unchanged state.

> *:nerd: One of the basic tenets of network programming is that a network connection can be broken at any time and the parties using such a connection must be able to handle this situation. In the present context, the client's connection to the PBX server may be broken at any time, either as a result of explicit action by the client or for other reasons. When disconnection of the client is noticed by the client service thread, the server acts as though the client had sent an explicit **hangup** command, the client is then unregistered from the PBX, and the client service thread terminates.*

The PBX system maintains the set of registered clients in the form of a mapping from assigned extension numbers to clients. It also maintains, for each registered client, information about the current state of the TU for that client. The following are the possible states of a TU:

- **On hook**: The TU handset is on the switchhook and the TU is idle.
- **Ringing**: The TU handset is on the switchhook and the TU ringer is active, indicating the presence of an incoming call.
- **Dial tone**: The TU handset is off the switchhook and a dial tone is being played over the TU receiver.
- **Ring back**: The TU handset is off the switchhook and a "ring back" signal is being played over the TU receiver.
- **Busy signal**: The TU handset is off the switchhook and a "busy" signal is being played over the TU receiver.
- **Connected**: The TU handset is off the switchhook and a connection has been established between this TU and the TU at another extension. In this state it is possible for users at the two TUs to "chat" with each other over the connection.
- **Error**: The TU handset is off the switchhook and an "error" signal is being played over the TU receiver.

Transitions between TU states occur in response to messages received from the associated client, and sometimes also asynchronously in conjunction with state transitions of other TUs. The list below specifies all the possible transitions between states that a TU can perform. The arrival of any message other than those explicitly listed for each particular state does not cause any transition between states to take place.

- When in the **on hook** state:
    - A **pickup** message from the client will cause a transition to the **dial tone** state.
    - An asynchronous transition to the **ringing** state is also possible from this state.
- When in the **ringing** state:
    - A **pickup** message from the client will cause a transition to the **connected** state. Simultaneously, the calling TU will also make an asynchronous transition from the **ring back** state to the **connected** state. The PBX will establish a connection between these two TUs, which we will refer to as *peers* as long as the connection remains established.
    - An asynchronous transition to the **on hook** state is also possible from this state. This would occur if the calling TU hangs up before the call is answered.
- When in the **dial tone** state:
    - A **hangup** message from the client will cause a transition to the **on hook** state.
    - A **dial** message from the client will cause a transition to either the **error**, **busy signal**, or **ring back** states, depending firstly on whether or not a TU is currently registered at the dialed extension, and secondly, whether the TU at the dialed extension is currently in the **on hook** state or in some other state. If there is no TU registered at the dialed extension, the transition will be to the **error** state. If there is a TU registered at the dialed extension, then if that extension is currently not in the **on hook** state, then the transition will be to the **busy signal** state, otherwise the transition will be to the **ring back** state. In the latter case, the TU at the dialed extension makes an asynchronous transition to the **ringing** state, simultaneously with the transition of the calling TU to the **ring back** state.
- When in the **ring back** state:
    - A **hangup** message from the client will cause a transition to the **on hook** state. Simultaneously, the called TU will make an asynchronous transition from the **ringing** state to the **on hook** state.
    - An asynchronous transition to the **connected** state (if the called TU picks up) or to the **dial tone** state (if the called TU unregisters) is also possible from this state.
- When in the **busy signal** state:
    - A **hangup** message from the client will cause a transition to the **on hook** state.
- When in the **connected** state:
    - A **hangup** message from the client will cause a transition to the **on hook** state. Simultaneously, the peer TU will make a transition from the **connected** state to the **dial tone** state.
    - An asynchronous transition to the **dial tone** state is also possible from this state. This would occur if the peer TU were to hang up.
- When in the **error** state:
    - A **hangup** message from the client will cause a transition to the **on hook** state.

Messages are sent between the client and server in a text-based format, in which each message consists of a single line of text, the end of which is indicated by the two-byte line termination sequence `"\r\n"`. There is no *a priori* limitation on the length of the line of text that is sent in a message. The possible messages that can be sent are listed below. The initial keywords in each message are case-sensitive.

- Commands from Client to Server
  - **pickup**
  - **hangup**
  - **dial** #, where # is the number of the extension to be dialed.
  - **chat** ...arbitrary text...
- Responses from Server to Client
  - **ON HOOK** #, where # reports the extension number of the client.
  - **RINGING**
  - **DIAL TONE**
  - **RING BACK**
  - **CONNECTED** #, where # is the number of the extension to which the connection exists.
  - **ERROR**
  - **CHAT** ...arbitrary text...

Demonstration Server

To help you understand what the PBX server is supposed to do, I have provided a complete implementation for demonstration purposes. Run it by typing the following command:

```
$ demo/pbx -p 3333
```

You may replace `3333` by any port number 1024 or above (port numbers below 1024 are generally reserved for use as "well-known" ports for particular services, and require "root" privilege to be used). The server should report that it has been initialized and is listening on the specified port. From another terminal window, use `telnet` to connect to the server as follows:

```
$ telnet localhost 3333
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
ON HOOK 4
```

You can now issue commands to the server:

```
pickup
DIAL TONE
dial 5
ERROR
hangup
ON HOOK 4
```

If you make a second connection to the server from yet another terminal window, you can make calls.

Note that the server will silently ignore syntactically incorrect commands, such as "dial" without an extension number. If commands are issued when the TU at the server side is in an inappropriate state (for example, if "dial 5" is sent when the TU is "on hook"), then a response will be sent by the server that just repeats the state of the TU, which does not change.

Task I: Server Initialization

When the base code is compiled and run, it will print out a message saying that the server will not function until `main()` is implemented. This is your first task. The `main()` function will need to do the following things:

- Obtain the port number to be used by the server from the command-line arguments. The port number is to be supplied by the required option `-p <port>`.

- Install a `SIGHUP` handler so that clean termination of the server can be achieved by sending it a `SIGHUP`. Note that you need to use `sigaction()` rather than `signal()`, as the behavior of the latter is not well-defined in a multithreaded context.

- Set up the server socket and enter a loop to accept connections on this socket. For each connection, a thread should be started to run function `pbx_client_service()`.

These things should be relatively straightforward to accomplish, given the information presented in class and in the textbook. If you do them properly, the server should function and accept connections on the specified port, and you should be able to connect to the server using the test client.

Task II: Server Module

In this part of the assignment, you are to implement the server module, which provides the function `pbx_client_service()` that is invoked when a client connects to the server. You should implement this function in the `src/server.c` file.

The `pbx_client_service` function is invoked as the **thread function** for a thread that is created (using `pthread_create()`) to service a client connection. The argument is a pointer to the integer file descriptor to be used to communicate with the client. Once this file descriptor has been retrieved, the storage it occupied needs to be freed. The thread must then become detached, so that it does not have to be explicitly reaped, it must initialize a new TU with that file descriptor, and it must register the TU with the PBX module under a particular extension number. The demo program uses the file descriptor as the extension number, but you may choose a different scheme if you wish. Finally, the thread should enter a service loop in which it repeatedly receives a message sent by the client, parses the message, and carries out the specified command. The actual work involved in carrying out the command is performed by calling the functions provided by the PBX module. These functions will also send the required response back to the client (each syntactically correct command will elicit a single response that contains the resulting state of the TU) so the server module need not be directly concerned with that.

Task III: PBX Module

The PBX module is the central module in the implementation of the server. It provides the functions listed below, for which more detailed specifications are given in the source file `pbx.c`.

- `PBX *pbx_init()`: Initialize a new PBX.
- `void pbx_shutdown(PBX *pbx)`: Shut down a PBX.
- `int pbx_register(PBX *pbx, TU *tu, int ext)`: Register a TU with a PBX.
- `int pbx_unregister(PBX *pbx, TU *tu)`: Unregister a TU from a PBX.
- `int pbx_dial(PBX *pbx, TU *tu, int ext)`: Dial an extension.

The PBX module will need to maintain a registry of connected clients and manage the TU objects associated with these clients. The PBX will need to be able to map each extension number to the associated TU object. As the PBX object will be accessed concurrently by multiple threads, it will need to provide appropriate synchronization (for example, using mutexes and/or semaphores) to ensure correct and reliable operation. Finally, the `pbx_shutdown()` function is required to shut down the network connections to all registered clients (the `shutdown(2)` function can be used to shut down a socket for reading, writing, or both, without closing the associated file descriptor) and it is then required to wait for all the client service threads to unregister the associated TUs before returning. Consider using a semaphore, possibly in conjunction with additional bookkeeping variables, for this purpose.

Task IV: TU Module

The TU module implements objects that simulate a telephone unit. The functions provided by this module are shown below. More detailed specifications can be found in the source file `tu.c`.

- `TU *tu_init(int fd)`: Initialize a new TU with the file descriptor for a client.
- `void tu_ref(TU *tu, char *reason)`: Increase the reference count (see below) of a TU by one.
- `void tu_unref(TU *tu, char *reason)`: Decrease the reference count of a TU by one, freeing the TU and associated resources if the reference count reaches zero.
- `int tu_fileno(TU *tu)`: Get the file descriptor for the network connection underlying a TU.
- `int tu_extension(TU *tu)`: Get the extension number for a TU.
- `int tu_set_extension(TU *tu, int ext)`: Set the extension number for a TU.
- `int tu_pickup(TU *tu)`: Take a TU receiver off-hook (i.e. pick up the handset).
- `int tu_hangup(TU *tu)`: Hang up a TU (i.e. replace the handset on the switchhook).
- `int tu_dial(TU *tu, int ext)`: Use a TU to originate a call to a specified extension.
- `int tu_chat(TU *tu, char *msg)`: "Chat" during a call to another TU.

Each TU object will contain the file descriptor of an underlying network connection to a client, as well as a representation of the current state of the TU. It will need to use the file descriptor to issue responses to the client, as well as any required asynchronous notifications, whenever any of the `tu_xxx` functions is called. Since the TU objects will be accessed concurrently by multiple threads, the TU module will need to provide appropriate synchronization. Changes to the state of a TU will require exclusive access to the TU. Some operations require simultaneous changes of state of two TUs; these will require exclusive access to both TUs at the same time in order to ensure the simultaneity of the state changes. Care must be taken to avoid deadlock when obtaining exclusive access to two TUs at the same time. Sending responses and notifications to the network client managed by a TU will also require exclusive access to the TU, in order to ensure that messages sent by separate threads are serialized over the network connection, rather than possibly intermingled.

In order for TU objects to exist independently of a PBX object, yet still avoid the possibility of having "dangling pointers" to TU objects that have unexpectedly been freed, the TU module uses a *reference counting* scheme. The basic idea of reference counts is that they are an integer field stored in an object that keeps track of the number of references (*i.e.* pointers) that exist to that object. When a pointer to a TU object is copied, the reference count on that object must be increased in order to account for the additional pointer that now exists. When a pointer to a TU object is discarded, the reference count on that object must be decreased in order to account for the pointer that no longer exists. When the reference count on a TU object has been decremented to zero, that means there are no longer any pointers by which that object can be accessed and therefore the object can safely be freed. Incrementing and decrementing the reference count of a TU is performed by calling the `tu_ref()` and `tu_unref()` functions. These functions take as an argument the TU to be manipulated, but in addition they take a second argument `msg`. This is for debugging purposes: when you increment or decrement a TU you should supply as the `msg` argument a string giving the reason why the reference count is being changed. If, when you implement the TU module, you have the `tu_ref()` and `tu_unref()` functions announce when they are called, along with the reason why, it will make it easier to debug reference count issues.

Test Exerciser

The `tests` directory in the basecode contains a test driver with a few selected tests to get you started. These tests are coded using Criterion as usual, but note that it is important that when you run the tests you supply the `-j1` argument to `pbx_tests`. This is because each test starts its own server instance and if you run them all concurrently only one server instance will be able to bind the port number that is being used and the other server instances will fail.

The file `basecode_tests.c` contains the Criterion portion of the tests and the file `script_tester.c` contains the test driver that they use. The file `__test_includes.h` contains some macros and declarations that are shared between the two files. The overall idea of the tests are that they are table-driven. Each test has a "script", which defines an array of `TEST_STEP` structures. The `TEST_STEP` structure is defined in `__test_includes.h`. Each step contains an `id`, which identifies a particular TU that should execute the step, a `command` which is the command to be sent to the TU, an `id_to_dial` field which specifies the extension to be dialed in case the `command` is TU_DIAL, a `response` field, which specifies the TU state that it is expected will be sent in the response from the server, and a `timeout` field, which can be used to place a limit on how long the tester will wait for a response from the server. The `timeout` values themselves have type `struct timeval` (see `man 2 setitimer` for a definition). There are several predefined timeout values of various lengths defined in `__test_includes.h`. The idea is that if the expected response from the server does not arrive before the timeout expires, then the test script fails. A timeout value of `ZERO_SEC` means the test driver will wait indefinitely for the response from the server.

To use the full capabilities of the test driver is probably somewhat complicated, since if you get multiple TUs sending commands in a concurrent fashion you have to start worrying about asynchronous notifications from the server "crossing in front of" the expected response to a command. But you can probably follow the basic pattern in the scripts that I have provided to create other similar scripts that test the ability of your server process other series of commands issued in rapid succession.

Debugging Multi-threaded Code with `gdb`

The `gdb` debugger has features for debugging multi-threaded code. In particular, it is aware of the presence of multiple threads and it provides commands for you to switch the focus of debugging from one thread to another. Use the `info threads` command to get a list of the existing threads. Use the command `thread nnn` (replace `nnn` by the thread number) to switch the focus of debugging to that thread. Once you have done, this, the `gdb` commands such as `bt` that examine the stack will be executed with respect to the selected thread. Threads can be stopped and started independently using `gdb`, as well.

Submission Instructions

Make sure your hw5 directory looks similarly to the way it did initially and that your homework compiles (be sure to try compiling both with and without "debug"). Note that you should omit any source files for modules that you did not complete, and that you might have some source and header files in addition to those shown. You are also, of course, encouraged to create Criterion tests for your code. Note that Criterion tests can themselves be multi-threaded; a single test case can create multiple threads to operate concurrently on the module being tested. Of course, some effort is required to design and implement such tests.

It would definitely be a good idea to use `valgrind` to check your program for memory and file descriptor leaks. Keeping track of allocated objects and making sure to free them is potentially one of the more challenging aspects of this assignment.

To submit, run `git submit hw5`.