CS 450 Assignment K2

1. How To Operate Program

The .elf artifact is committed in the git repository. See section 4 for a link to the code repsoitory. Alternatively see section 5 for instructions to build from source. Once the artifact is obtained, load it into redboot and run `go`.

2. Group Member Names

Xuanji Li

Lennox Fei

3. Kernel Structure

Map Data Structure

A generic (polymorphic) key-value store data structure is provided in `map.h`. This supports inserting a value at a given key, deleting an item, and searching for the value at a given key. The map is implemented using an AVL tree. The global COMM data structure uses maps to store tasks blocked at some point in the send-receive-reply cycle so that when processing a potentially-unblocking task (e.g. `Send`) the scheduler can quickly search if a task will be unblocked (in this case, if any tasks had previously called `Receive`). In addition, the nameserver uses a Map to store the mapping of hashed name to TID. Note that these two cases use different type parameters for Map, i.e., conceptually we can think of them as using `Map<int, Task*>` and `Map<int, int>` (note that we use runtime typecasting to support this, there's no compiler-checked parametric polymorphism or templates of any kind).

Queue Data Structure

The Queue data structure from the last submission has been made polymorphic as well and is used in various places in the kernel. For instance the scheduler now has a queue of free tasks (`freeQueue`) it pops from when new tasks are needed. This way, when `Destroy` is implemented, we can efficiently store and update the list of useable task descriptors (those that have never been used before, or have been used and then destroyed) without scanning the entire list of task descriptors. This "free queue" pattern is also used by COMM.

COMM

The `COMM` data structure (`sendReceiveReply.h`) stores TDs that are blocked in the SRR cycle. We enforce the invariant that every TD is either in

1. scheduler current task
2. scheduler ready queue
3. tracked in COMM

In this way, the scheduler and COMM work very closely together, are mutually recursive, and could conceivably be merged together.

The tracking in COMM is more involved than in scheduler. The `Receiver` struct represents a task blocked on a `Receive` syscall and hence contains the arguments to the syscall. Since memory must be allocated for this we use a free queue for these structs. Similarly the `Sender` struct represents tasks blocked on `Send` (i.e. waiting for a `Reply`). In both cases there is a `Map` in COMM to store these structs; we add to the map we block the task, search in the map when checking if a potentially-unblocking syscall unblocks anyone, and remove it from the map if it does.

Note: when tracked by COMM, the TDs are still allocated in the scheduler, we just store the TID in COMM. However since the scheduler does not iterate through the TID space, when a task is tracked in COMM (i.e., not in the scheduler's current task or ready queue), it is effectively "lost" to the scheduler and can only be rescheduled by COMM. See the section "Kernel Loop" for implications of this.

Nameserver

The nameserver stores a map of hashed name to TID. The hash has codomain `int` so the conceptual type of the map is `Map<int, int>`. For the hash function (`alphaNumericHash`) we use the base-67 value of the input string, where each character is treated as a "digit". This has the advantage that as long as the length of the input, when treated as a number, does not exceed what can be represented as an `int`, we have no collisions. This disadvantage is that this is not designed to be a collision-resistant hash function family, hence if we allow the input to exceed this constraint, it is trivial to find collisions. For now, since we control all the input to the nameserver in this assignment, we are safe.

To bootstrap the name resolution, we communicate the nameserver TID via the `nsTid` global variable (hence violating the memory separation principle), which is written to when the nameserver starts, and read from by `RegisterAs` and `WhoIs`. By doing this instead of setting the name server at a fixed TID, we can start any number of tasks before the name server, as long as those tasks don't use the nameserver before it is started. This is useful for debugging.

The `RegisterAs` and `WhoIs` functions both send a message to the nameserver; the first character specifies which function it is, and the string from the third character on is the function argument. Hence the message looks a lot like an RPC. Correspondingly, the nameserver loops, `Receive`s messages, parses them and `Replies` with the return value.

Kernel Loop

We add a new feature to the kernel loop, a warning if there are no active tasks but there are blocked tasks. (Easy way to test this: create a single task that calls `Receive`). The decision to make this a warning was a conscious design choice. Clearly, within the scope of the syscalls implemented for K2, in the situation where the warning is emitted, no further progress can be made and the kernel should quit. Often, this indicates a bug (e.g. if a wrongly implemented RPS server exited while some clients are `Send`-blocked) and we should know about it. However, in the future as more features are added it might be simpler to write things like the nameserver as "loop-forever" tasks that rely on the kernel exit to terminate instead of shutting down cleanly.

4. Code repository

https://git.uwaterloo.ca/f5fei/chos

5. Build and Execution

When you are in the root directory of the repo (chos/), run the following commands

make

make install

Make should build everything, and make install will move it to `ARM/[uwid]`, this allows for parallel testing. In case you get issues with missing linker libraries, do the following to regenerate the symbol link

cd lib ./rebuildLib.sh

6. Output and explanation

```
[rpsServer 3]   7 = Receive(=4, =_signup)
[rpsServer 3]   7 = Receive(=5, =_signup)
[rpsClient 5]   5 = Send(3, _signup, =first)
[rpsClient 4]   5 = Send(3, _signup, =first)
[rpsServer 3]   4 = Receive(=5, =rock)
[rpsServer 3]   4 = Receive(=4, =rock)
[rpsClient 4]   4 = Send(3, rock, =draw)
[rpsClient 5]   4 = Send(3, rock, =draw)
[rpsServer 3]   5 = Receive(=4, =paper)
[rpsServer 3]   8 = Receive(=5, =scissors)
[rpsClient 4]   4 = Send(3, paper, =draw)
[rpsClient 5]   4 = Send(3, scissors, =draw)
[rpsServer 3]   6 = Receive(=4, =i_quit)
[rpsServer 3]   4 = Receive(=5, =rock)
[rpsClient 5]   9 = Send(3, rock, =they_quit)
[rpsClient 4]   9 = Send(3, i_quit, =they_quit)
[rpsServer 3]   7 = Receive(=6, =_signup)
[rpsServer 3]   7 = Receive(=7, =_signup)
[rpsClient 6]   5 = Send(3, _signup, =first)
[rpsClient 7]   5 = Send(3, _signup, =first)
[rpsServer 3]   5 = Receive(=6, =paper)
[rpsServer 3]   5 = Receive(=7, =paper)
[rpsClient 6]   4 = Send(3, paper, =draw)
[rpsClient 7]   4 = Send(3, paper, =draw)
[rpsServer 3]   5 = Receive(=6, =paper)
[rpsServer 3]   8 = Receive(=7, =scissors)
[rpsClient 6]   4 = Send(3, paper, =draw)
[rpsClient 7]   4 = Send(3, scissors, =draw)
[rpsServer 3]   8 = Receive(=6, =scissors)
[rpsServer 3]   5 = Receive(=7, =paper)
[rpsClient 6]   4 = Send(3, scissors, =draw)
[rpsClient 7]   4 = Send(3, paper, =draw)
[rpsServer 3]   5 = Receive(=6, =paper)
[rpsServer 3]   4 = Receive(=7, =rock)
```

```
[rpsClient 6]   4 = Send(3, paper, =draw)
[rpsClient 7]   4 = Send(3, rock, =draw)
[rpsServer 3]   6 = Receive(=6, =i_quit)
[rpsServer 3]   8 = Receive(=7, =scissors)
[rpsServer 3]   7 = Receive(=8, =_signup)
[rpsClient 7]   9 = Send(3, scissors, =they_quit)
[rpsClient 6]   9 = Send(3, i_quit, =they_quit)
[rpsServer 3]   7 = Receive(=9, =_signup)
[rpsClient 8]   5 = Send(3, _signup, =first)
[rpsClient 9]   5 = Send(3, _signup, =first)
[rpsServer 3]   8 = Receive(=8, =scissors)
[rpsServer 3]   8 = Receive(=9, =scissors)
[rpsClient 8]   4 = Send(3, scissors, =draw)
[rpsClient 9]   4 = Send(3, scissors, =draw)
[rpsServer 3]   5 = Receive(=8, =paper)
[rpsServer 3]   6 = Receive(=9, =i_quit)
[rpsClient 8]   9 = Send(3, paper, =they_quit)
[rpsClient 9]   9 = Send(3, i_quit, =they_quit)
[rpsServer 3]   7 = Receive(=10, =_signup)
[rpsServer 3]   7 = Receive(=11, =_signup)
[rpsClient 10]  5 = Send(3, _signup, =first)
[rpsClient 11]  5 = Send(3, _signup, =first)
[rpsServer 3]   4 = Receive(=10, =rock)
[rpsServer 3]   4 = Receive(=11, =rock)
[rpsClient 10]  4 = Send(3, rock, =draw)
[rpsClient 11]  4 = Send(3, rock, =draw)
[rpsServer 3]   5 = Receive(=10, =paper)
[rpsServer 3]   8 = Receive(=11, =scissors)
[rpsClient 10]  4 = Send(3, paper, =draw)
[rpsClient 11]  4 = Send(3, scissors, =draw)
[rpsServer 3]   5 = Receive(=10, =paper)
[rpsServer 3]   4 = Receive(=11, =rock)
[rpsClient 10]  4 = Send(3, paper, =draw)
[rpsClient 11]  4 = Send(3, rock, =draw)
[rpsServer 3]   4 = Receive(=10, =rock)
[rpsServer 3]   6 = Receive(=11, =i_quit)
[rpsClient 10]  9 = Send(3, rock, =they_quit)
[rpsClient 11]  9 = Send(3, i_quit, =they_quit)
[rpsServer 3]   7 = Receive(=12, =_signup)
[rpsServer 3]   7 = Receive(=13, =_signup)
[rpsClient 12]  5 = Send(3, _signup, =first)
[rpsClient 13]  5 = Send(3, _signup, =first)
[rpsServer 3]   6 = Receive(=12, =i_quit)
[rpsServer 3]   5 = Receive(=13, =paper)
[rpsServer 3]   7 = Receive(=14, =_signup)
[rpsClient 13]  9 = Send(3, paper, =they_quit)
[rpsClient 12]  9 = Send(3, i_quit, =they_quit)
[rpsServer 3]   7 = Receive(=15, =_signup)
[rpsClient 14]  5 = Send(3, _signup, =first)
[rpsClient 15]  5 = Send(3, _signup, =first)
[rpsServer 3]   8 = Receive(=14, =scissors)
[rpsServer 3]   8 = Receive(=15, =scissors)
[rpsClient 14]  4 = Send(3, scissors, =draw)
[rpsClient 15]  4 = Send(3, scissors, =draw)
[rpsServer 3]   5 = Receive(=14, =paper)
[rpsServer 3]   4 = Receive(=15, =rock)
[rpsClient 14]  4 = Send(3, paper, =draw)
[rpsClient 15]  4 = Send(3, rock, =draw)
```

```
[rpsServer 3]   6 = Receive(=14, =i_quit)
[rpsServer 3]   8 = Receive(=15, =scissors)
[rpsServer 3]   7 = Receive(=16, =_signup)
[rpsClient 15]  9 = Send(3, scissors, =they_quit)
[rpsClient 14]  9 = Send(3, i_quit, =they_quit)
[rpsServer 3]   7 = Receive(=17, =_signup)
[rpsClient 16]  5 = Send(3, _signup, =first)
[rpsClient 17]  5 = Send(3, _signup, =first)
[rpsServer 3]   4 = Receive(=16, =rock)
[rpsServer 3]   4 = Receive(=17, =rock)
[rpsClient 16]  4 = Send(3, rock, =draw)
[rpsClient 17]  4 = Send(3, rock, =draw)
[rpsServer 3]   5 = Receive(=16, =paper)
[rpsServer 3]   4 = Receive(=17, =rock)
[rpsClient 16]  4 = Send(3, paper, =draw)
[rpsClient 17]  4 = Send(3, rock, =draw)
[rpsServer 3]   5 = Receive(=16, =paper)
[rpsServer 3]   4 = Receive(=17, =rock)
[rpsClient 16]  4 = Send(3, paper, =draw)
[rpsClient 17]  4 = Send(3, rock, =draw)
[rpsServer 3]   8 = Receive(=16, =scissors)
[rpsServer 3]   8 = Receive(=17, =scissors)
[rpsClient 16]  4 = Send(3, scissors, =draw)
[rpsClient 17]  4 = Send(3, scissors, =draw)
[rpsServer 3]   6 = Receive(=16, =i_quit)
[rpsServer 3]   5 = Receive(=17, =paper)
[rpsServer 3]   7 = Receive(=18, =_signup)
[rpsClient 17]  9 = Send(3, paper, =they_quit)
[rpsClient 16]  9 = Send(3, i_quit, =they_quit)
[rpsServer 3]   7 = Receive(=19, =_signup)
[rpsClient 18]  5 = Send(3, _signup, =first)
[rpsClient 19]  5 = Send(3, _signup, =first)
[rpsServer 3]   5 = Receive(=18, =paper)
[rpsServer 3]   5 = Receive(=19, =paper)
[rpsClient 18]  4 = Send(3, paper, =draw)
[rpsClient 19]  4 = Send(3, paper, =draw)
[rpsServer 3]   5 = Receive(=18, =paper)
[rpsServer 3]   6 = Receive(=19, =i_quit)
[rpsClient 18]  9 = Send(3, paper, =they_quit)
[rpsClient 19]  9 = Send(3, i_quit, =they_quit)
[rpsServer 3]   7 = Receive(=20, =_signup)
[rpsServer 3]   7 = Receive(=21, =_signup)
[rpsClient 20]  5 = Send(3, _signup, =first)
[rpsClient 21]  5 = Send(3, _signup, =first)
[rpsServer 3]   8 = Receive(=20, =scissors)
[rpsServer 3]   4 = Receive(=21, =rock)
[rpsClient 20]  4 = Send(3, scissors, =draw)
[rpsClient 21]  4 = Send(3, rock, =draw)
[rpsServer 3]   5 = Receive(=20, =paper)
[rpsServer 3]   4 = Receive(=21, =rock)
[rpsClient 20]  4 = Send(3, paper, =draw)
[rpsClient 21]  4 = Send(3, rock, =draw)
[rpsServer 3]   4 = Receive(=20, =rock)
[rpsServer 3]   8 = Receive(=21, =scissors)
[rpsClient 20]  4 = Send(3, rock, =draw)
[rpsClient 21]  4 = Send(3, scissors, =draw)
[rpsServer 3]   8 = Receive(=20, =scissors)
[rpsServer 3]   8 = Receive(=21, =scissors)
```

```
[rpsClient 20]  4 = Send(3, scissors, =draw)
[rpsClient 21]  4 = Send(3, scissors, =draw)
[rpsServer 3]   4 = Receive(=20, =rock)
[rpsServer 3]   4 = Receive(=21, =rock)
[rpsClient 20]  4 = Send(3, rock, =draw)
[rpsClient 21]  4 = Send(3, rock, =draw)
[rpsServer 3]   5 = Receive(=20, =paper)
[rpsServer 3]   6 = Receive(=21, =i_quit)
[rpsClient 20]  9 = Send(3, paper, =they_quit)
[rpsClient 21]  9 = Send(3, i_quit, =they_quit)
[rpsServer 3]   7 = Receive(=22, =_signup)
[rpsServer 3]   7 = Receive(=23, =_signup)
[rpsClient 23]  5 = Send(3, _signup, =first)
[rpsServer 3]   5 = Receive(=23, =paper)
[rpsClient 22]  5 = Send(3, _signup, =first)
[rpsServer 3]   6 = Receive(=22, =i_quit)
[rpsClient 23]  9 = Send(3, paper, =they_quit)
[rpsClient 22]  9 = Send(3, i_quit, =they_quit)
[rpsServer 3]   3 = Receive(=24, =kys)
[killer 24]     2 = Send(3, kys, =ok)
[killer 24]     2 = Send(2, kys, =ok)
```

Output Generation

This output was generated with PRNG seed `0xdeadbeef` and `bwputc` turned off and then copied. We ran it through `expand -t8` to convert tabs to spaces to preserve column alignment, since LaTeX's verbatim enviroment does not do column alignment for tabs.

Output Format

Output format is inspired by `strace`. The `Send` and `Receive` system calls are "instrumented" in the rps client, rps server and killer tasks, by calling them through a wrapper function. Each line contains: `[<symbolic name> <TID>] <ret> = <Send/Receive>(...args)` where symbolic name is passed in manually. For args, an `=` preceding an argument means that it is an output argument (e.g. the second argument to Send, '`const char *msg`' in the specs), and we print the output. All length arguments are omitted from the trace

For e.g.: look at the last line. This means that the "killer" task (`killServer` in `k2.c`) did a `Send` to TID 2 with message `"kys"` and received a reply `"ok"` (as an output parameter, i.e. `"ok"` was copied to the buffer passed as the 4th argument to Send).

We don't instrument `Reply` calls as they are all quite boring; by design of the user program, all `Reply` calls are printed by a returning `Send` call

Killer Task

To check that the tasks exit cleanly, we modify the nameserver and the rps server to accept a `kys` message to exit their loop and call `Exit`. We create a low-priority task (labelled `killer` here) to send the kys message at the end. This explains why the last two lines are as they are. More importantly, because we don't see the blocked tasks warning (see the Kernel Loop section) we know that each client exits cleanly.

Client

Each rps client randomly decides on a number of rounds to play, then `Send`s that number of random moves. Of course, they send the signup message and waits for the "first choice" reply, and exit early if they get a notification that their counterparty has quit.

Note: while a client is not matched, it is blocked. You can test this by creating just one client and observing the warning.

Server

The RPS server keeps track of an active matched game as well as a queue of unmatched clients. The main loop of the server does a `Receive` every iteration and is then divided into handling the receive (modifying local server state) and handling matches (reading local server state to see if a new event, e.g. matchup, both players have played, someone quit etc) to `Reply` to the clients. The exception is that the receive handler immediately replies if the message is from someone whose counterparty has quit. We also use potentially unneeded quit replies to make sure no clients are blocked while simplifying our code, relying on the fact that `Reply` does not block the sender if the target does not exist / is not blocked on `Send`.

Priorities

The nameserver is created with highest priority, then the RPS server, then all the clients at the same priority, then the killer. The RPS server runs at a higher priority than any client so that its `Receive` is always called first (otherwise some client `Send`s to the server could fail, requiring potentially ugly code to recover from). The relative priorities of the clients don't really matter; you can make them random if you want.

Sample subtrace

The clients with TIDs 3 and 4 got matched to each other. In the first four lines we see the server's `Receive` logs first, and then the corresponding clients' send logs (that caused the `Receive`) to return. The reason for this order is that the server has higher priority. Similarly the next four lines show them both choosing rock and being notified that they drew the round (also in the order server, then client). After a while we see that client 4 sent the `i_quit` command, and client 3 getting the `they_quit` reply. Client 4 also gets that reply (the reply value to a quit message doesn't matter, it is just for unblocking). We also see the client 5 game starting before the quit messages because the clients run at the same priority, so the client printing happens a long time after the quit reply gets received.

PRNG

A Linear congruential generator is implemented for randomness. We made the design deterministic for debuggability. Each task that requires randomness allocates a seed on its stack; the seed is derived from the "root seed" (current set to `0xdeadbeef`) added to the task TID.